Normalization and Partial Evaluation of Functional Logic Programs

M. Sc. Björn Peemöller

Dissertation
zur Erlangung des akademischen Grades
Doktor der Ingenieurwissenschaften
(Dr.-Ing.)
der Technischen Fakultät
der Christian-Albrechts-Universität zu Kiel
eingereicht im Jahr 2016

Kiel Computer Science Series (KCSS) 2017/1 dated 2017-06-11

URN:NBN urn:nbn:de:gbv:8:1-zs-00000328-a6
ISSN 2193-6781 (print version)
ISSN 2194-6639 (electronic version)

Electronic version, updates, errata available via https://www.informatik.uni-kiel.de/kcss

The author can be contacted via bjoern.peemoeller@gmail.com

Published by the Department of Computer Science, Kiel University

Programming Languages and Compiler Construction

Please cite as:

▻ Björn Peemöller. *Normalization and Partial Evaluation of Functional Logic Programs*. Number 2017/1 in Kiel Computer Science Series. Department of Computer Science, 2017. Dissertation, Faculty of Engineering, Kiel University.

```
@book{Peemoeller2017,
  author    = {Bj{\"o}rn Peem{\"o}ller},
  title     = {Normalization and Partial Evaluation of Functional Logic Programs},
  publisher = {Department of Computer Science, Kiel University},
  year      = {2017},
  number    = {2017/1},
  isbn      = {978-3-7448-2192-6}
  series    = {Kiel Computer Science Series},
  note      = {Dissertation, Faculty of Engineering, Kiel University}
}
```


Herstellung und Verlag: BoD — Books on Demand, Norderstedt

ISBN: 978-3-7448-2192-6

About this Series

The Kiel Computer Science Series (KCSS) covers dissertations, habilitation theses, lecture notes, textbooks, surveys, collections, handbooks, etc. written at the Department of Computer Science at Kiel University. It was initiated in 2011 to support authors in the dissemination of their work in electronic and printed form, without restricting their rights to their work. The series provides a unified appearance and aims at high-quality typography. The KCSS is an open access series; all series titles are electronically available free of charge at the department's website. In addition, authors are encouraged to make printed copies available at a reasonable price, typically with a print-on-demand service.

Please visit `http://www.informatik.uni-kiel.de/kcss` for more information, for instructions how to publish in the KCSS, and for access to all existing publications.

1. Gutachter:	Prof. Dr. Michael Hanus Christian-Albrechts-Universität zu Kiel
2. Gutachter:	Priv.-Doz. Dr. Frank Huch Christian-Albrechts-Universität zu Kiel

Datum der mündlichen Prüfung: 25. Juli 2016

Zusammenfassung

Diese Arbeit befasst sich mit der Normalisierung und der partiellen Auswertung von Programmen in der funktional-logischen Programmiersprache Curry. Das Paradigma der *funktional-logischen Programmierung* kombiniert die beiden wichtigsten Bereiche der deklarativen Programmierung, die funktionale und die logische Programmierung. Während funktionale Sprachen Konzepte wie algebraische Datentypen, Funktionen höherer Ordnung oder eine bedarfsgesteuerte Auswertung bereitstellen, bieten logische Sprachen eine nicht-deterministische Auswertung sowie eine eingebaute Suche nach Ergebnissen. Diese beiden Paradigmen werden in funktional-logischen Sprachen kombiniert, sodass eine Vielzahl an Sprachkonstrukten und Konzepten zur Verfügung steht um kompakte und ausdrucksstarke Programme zu entwickeln. Jedoch erschweren die Vielzahl syntaktischer Konstrukte und der hohe Abstraktionsgrad auch die Übersetzung in effiziente Zielprogramme.

Zur Reduktion der syntaktischen Komplexität beinhalten gängige Kompilierungsschemata üblicherweise eine *Normalisierungsphase*, in der komplexe Konstrukte schrittweise durch einfachere ersetzt werden, bis ein Programm schließlich in einer minimalen Teilsprache ausgedrückt werden kann. Obwohl einzelne Transformationen vergleichsweise einfach sind, muss auch deren korrekte Kombination sichergestellt sein, damit die Konstrukte ihre ursprüngliche Bedeutung behalten.

Die Effizienz normalisierter Programme kann dann mittels verschiedener Optimierungstechniken verbessert werden. Eine sehr mächtige Technik ist hierbei die *partielle Auswertung*, bei der zur Kompilierungszeit die Ausführung bestimmter Programmteile simuliert wird, um so ein semantisch äquivalentes Programm zu berechnen, das zur Laufzeit üblicherweise effizienter ausgeführt wird. Da die partielle Auswertung eine vollautomatische Optimierung darstellt, kann sie zudem in bestehende Kompilierungsschemata eingebunden werden. Allerdings erfordert dies auch die Terminierung des Optimierungsprozesses, was eine wesentliche Herausforderung neben der semantischen Äquivalenz darstellt.

Diese Arbeit betrachtet die Sprache Curry als Repräsentanten für das funktional-logische Programmierparadigma. Zunächst wird eine formale Darstellung der Normalisierung von Curry-Programmen entwickelt, um die korrekte Transformation unterschiedlicher Sprachkonstrukte festzulegen. Für die resultierende Kernsprache wird dann deren dynamische Semantik definiert, um darauf aufbauend ein grundlegendes Schema zur partiellen Auswertung zu entwickeln und dessen Korrektheit und Terminierung zu zeigen. Aufgrund der Normalisierung eignet sich dieses Schema bereits zur partiellen Auswertung beliebiger Curry-Programme. Des Weiteren wird die Implementierung eines praktischen partiellen Auswerters skizziert und dessen Anwendbarkeit und Vorteilhaftigkeit anhand einiger typischer Beispiele demonstriert.

Abstract

This thesis deals with the development of a normalization scheme and a partial evaluator for the functional logic programming language Curry. The *functional logic programming* paradigm combines the two most important fields of declarative programming, namely functional and logic programming. While functional languages provide concepts such as algebraic data types, higher-order functions or demand-driven evaluation, logic languages usually support a non-deterministic evaluation and a built-in search for results. Functional logic languages finally combine these two paradigms in an integrated way, hence providing multiple syntactic constructs and concepts to facilitate the concise notation of high-level programs. However, both the variety of syntactic constructs and the high degree of abstraction complicate the translation into efficient target programs.

To reduce the syntactic complexity of functional logic languages, a typical compilation scheme incorporates a *normalization* phase to subsequently replace complex constructs by simpler ones until a minimal language subset is reached. While the individual transformations are usually simple, they also have to be correctly combined to make the syntactic constructs interact in the intended way.

The efficiency of normalized programs can then be improved by means of different optimization techniques. A very powerful optimization technique is the *partial evaluation* of programs. Partial evaluation basically anticipates the execution of certain program fragments at compile time and computes a semantically equivalent program, which is usually more efficient at run time. Since partial evaluation is a fully automatic optimization technique, it can also be incorporated into the normal compilation scheme of programs. Nevertheless, this also requires termination of the optimization process, which establishes one of the main challenges for partial evaluation besides semantic equivalence.

In this work we consider the language Curry as a representative of the functional logic programming paradigm. We develop a formal representation of the normalization process of Curry programs into a kernel language, while respecting the interference of different language constructs. We then define the dynamic semantics of this kernel language, before we subsequently develop a partial evaluation scheme and show its correctness and termination. Due to the previously described normalization process, this scheme is then directly applicable to arbitrary Curry programs. Furthermore, the implementation of a practical partial evaluator is sketched based on the partial evaluation scheme, and its applicability and usefulness is documented by a variety of typical partial evaluation examples.

Preface by Prof. Dr. Michael Hanus

Declarative programming languages support the development of reliable software in a high-level manner. Due to their abstraction from a concrete computer architecture, the implementation of declarative programming languages is demanding. This thesis is an interesting contribution to this problem. Björn Peemöller developed an optimization technique for functional logic programs which is applied to the source level. Since declarative programs often contain generic and re-usable operations that are applied in various contexts, it is advantageous to specialize the various uses of such operations and generate new specialized operations that are usually more efficient. This transformation process is called "partial evaluation" and it has been successfully applied to different classes of programming languages. In fact, partial evaluation for functional logic languages has been pioneered by the Technical University of Valencia more than fifteen years ago. Since the structure and practice of functional logic programming has been advanced since that time, the original partial evaluation techniques are not applicable to modern functional logic languages like Curry.

Björn Peemöller solved this gap in his thesis. Since partial evaluation is a complex process, he develops a partial evaluator for a simplified intermediate language, called FlatCurry. In order to apply this partial evaluator to practical Curry programs, the first part of the thesis specifies a transformation of Curry programs into FlatCurry programs. Although this transformation is implemented in contemporary Curry systems, the precise and abstract specification of this task is an important contribution which can be used as a standard reference in future work. The main part of this thesis contains the specification and correctness proof of the partial evaluator. In order to develop a practical tool, a number of smaller transformations are added which improve the efficiency of the generated programs. The final benchmarks demonstrate the success of his approach. The developed partial evaluator produces similar efficient code as the original partial evaluator, but it is applicable to a larger class of programs. In particular, it is able to optimize programs containing functional patterns so that the potential non-determinism is completely eliminated. In this way the partial evaluator can be used as an automatic tool to transform high-level non-deterministic specifications into efficient deterministic programs.

This thesis contains good and original results and I recommend it to anyone interested in source code optimization techniques for declarative programs.

Michael Hanus
Kiel, October 2016

Acknowledgements

A dissertation thesis is the culmination of a long process of constant efforts, and this result would not have been possible without the help of many different people. I therefore like to take the opportunity to express my thanks for their support.

First of all, I like to thank my supervisor *Michael Hanus* for giving me the opportunity to join his research group in Kiel in 2010. Since then, I really enjoyed discovering the research field of functional logic programming languages, a paradigm I haven't been agnostic to beforehand, containing impressive and sometimes delicate features. Furthermore, I had the opportunity to participate in the development of the functional logic language Curry, ranging from the specification of new language features, across its normalization to the kernel language of FlatCurry, and finally to the implementation of the KiCS2 Curry system. I furthermore like to thank Michael Hanus for providing me the freedom to strive for a research topic that caught my interest, and the many illustrative discussions we had.

I want to thank *Fabian Skrlac* for the countless hours we spent together during the implementation of the KiCS2 system, and on the hunt for elusive implementation bugs. Working on the ecosystem of Curry would not have been half as much fun without you! *Jan Tikovsky* always lent his ear to me during my process of writing, let it be spontaneous ideas or dissatisfaction, and I really appreciated his support. *Frank Huch* has a fascinating ability to fathom the ideas of other people and expose any culprits, and I thank him for the insights that appeared to me. Moreover, the atmosphere at the research group has always been very pleasant and encouraging, and I deeply thank *all members of the seventh floor* for their kindness, and the *Technik-Service* for their pleasant company during lunch breaks.

Furthermore, I like to express my thanks to all persons who have provided feedback on earlier drafts of this work, and especially *Jan Tikovsky* and *Heide Rasmus* for their extensive suggestions. I also like to express my thankfulness to *my family* and *my friends*, who assisted me in these ambitious times and helped me to find some regeneration from writing.

Finally, I am deeply grateful to *Janna Rasmus* for encouraging me to apply for this fascinating venture I have undertaken, and her ongoing support all along the way.

Björn Peemöller
Kiel, June 2017

Contents

List of Figures

Chapter 1

Introduction

Abstraction is one of the greatest visionary tools ever invented by human beings to imagine, decipher, and depict the world.

Jerry Saltz

Since the invention of the first computer programming languages, their evolution was constantly influenced by the requirement to find new *abstractions*. Abstractions free the programmer from dealing with specific peculiarities, and allow the construction of programs based on more general structures. *Assembly languages*, for instance, abstract from the concrete machine code by means of mnemonic operations and make the programs readable for human beings instead of computers. Nevertheless, their model of computation still corresponds to the model of the underlying hardware, which is based on registers, memory cells, and processor instructions. *Imperative programming* languages establish another level of abstraction, by providing concepts such as (structured) data types, variables, and structured statements such conditions, loops, and procedures. This abstraction allows the programmer to ignore particular details about the implementation, for instance the exact mechanism how values are copied between memory cells and registers. The programmer can then concentrate on the implementation of the algorithms, which dramatically facilitates the development of larger software systems. *Declarative programming* languages furthermore abstract from the computation steps necessary to solve a given problem, by requiring only a description of the properties of a desired solution. This in turn frees the programmer from the invention of sophisticated algorithms, so that the implementation of the programming language can be used to find a solution according to the specification.

1.1 Functional Logic Programming

The most prominent paradigms in the field of declarative programming are functional and logic programming. *Functional programming* is based upon the composition of simple functions to larger ones, and often provides compelling features such as algebraic data types, higher-order functions, type polymorphism, and lazy evaluation. *Logic programming*, on the other hand, is based on the definition of predicates that

allow the deduction of new knowledge from basic facts via inference rules. In addition, it provides the ability to deal with multiple solutions by non-determinism, as well as with partial information in the form of logic variables. More recent research aims at the seamless integration of both paradigms, leading to the field of integrated *functional logic programming* languages. Although the amalgamation of both paradigms sometimes causes delicate combinations such as lazy non-determinism, it has set the foundations for the advent of new programming patterns and syntactic extensions. In the field of functional logic languages, the language Curry [Han12] was created by an international initiative to provide a common platform for research, teaching, and application. For the rest of this work, we will therefore target the language Curry exclusively, although the presented ideas are applicable to functional logic programming languages in general.

1.2 Program Transformation

The idea of *program transformation*, which goes back to McCarthy [McC61], comprises different techniques that aim to make an existing program more efficient while preserving its semantics. That is, given a program P, the task of program transformation is to obtain a different program P' that solves the same problem as P and computes the same results, but ideally in an improved behavior with respect to some criteria such as elapsed run time or memory consumption. A general approach in the area of program transformation is the *fold/unfold framework* of Burstall and Darlington [BD77], originally proposed for the transformation of functional programs. This framework defines six basic transformations that can be applied to a given program to derive a more efficient resulting program:

define allows the definition of a new function,

instantiate allows an existing function to be specialized with respect to a given set of arguments,

unfold allows the replacement of a function call by the right-hand side of the function definition with the function parameters substituted by the actual arguments,

fold does the reverse by replacing an expression with an equivalent function call,

abstract allows the extraction of common subexpressions which are then shared by a local variable definition, and

laws allows the application of known laws for operations, such as associativity or commutativity.

In principle, this method is capable of achieving super-linear speedups [JGS93], but unrestricted application of the transformations may also lead to misshapen programs with a degraded performance. Consequently, a *strategy* for the application of

transformations is necessary to obtain an automatic method for program transformation. In the literature, there exists a wide range of program transformation techniques which can be seen as instances of the general fold/unfold framework, and typical representatives can be classified as follows.

Symbolic Function Composition A common practice in functional programming is the partition of a program into small functions, each responsible for a specific task, and the construction of more complex functions from simpler ones by means of *function composition*. Although this style of programming leads to modular programs, it may also affect its efficiency, since passing the result of one function as the input of another usually requires some intermediate data structures. By a *symbolic function composition*, previously independent function definitions can be merged using the general folding and unfolding transformations, so that the intermediate data structures effectively vanish.

A popular instance of this approach is Wadler's work on *deforestation* [Wad90], where he presents an algorithm for the composition of functions defined in a specific form called *treeless* to eliminate intermediate data structures. Another approach is the automatic composition of functions which are defined using two functions `fold` and `build`, based on a single fusion rule for merging their invocations [GLP93]. This method is sometimes also referred to as *short-cut deforestation* or *fold/build-fusion*.

Tupling The basic idea of the *tupling* strategy [BD77; Pet77] is to combine the definitions of two functions recursing on a common argument into a single function returning a tuple of the previous results, and thus avoiding a repeated recursion. A common example is the computation of the mean of a list of numbers, which requires both the computation of the sum of the elements and the length of the list. A tupled function can then compute both the sum and the length in parallel, thus saving an additional traversal of the list. Unfortunately, the identification of a suitable tuple, often called the "eureka tuple", requires a complex analysis, which limits its potential for automation [PP94].

Partial Evaluation Partial evaluation [JGS93], sometimes also called partial computation or program specialization, is an optimization technique that takes a program and a function call and derives a more efficient variant of this function which provides the same semantics. The partial evaluation process mainly follows the usual evaluation process of the respective language, and the decision on the transformation steps to be applied may furthermore be guided by some additional analysis.

Supercompilation The technique of *supercompilation* (*super*vised *compilation*) according to Turchin [Tur86] performs the transformations of *driving*, i. e., the unfolding of functions and the propagation of information, and *generalization*, a form of

abstraction enabling folding. Based on the kind of propagated information, supercompilation is sometimes distinguished into supercompilation with *positive* information (assertions on the values of variables) or *negative* information (restrictions on the values of variables). If both positive and negative information is used, this is also referred to as *perfect* supercompilation [SS99]. Due to the propagation of information obtained during driving, supercompilation is strictly more powerful than classical partial evaluation, and can also achieve the results of deforestation. However, the process of driving may also extend the domain of functions, and thus yield a resulting program with a different semantics [JGS93; GS94].

Generalized Partial Computation As an extension to partial evaluation, Futamura and Nogi [FN88] presented the method of *generalized partial computation* which combines classical partial evaluation with the consideration of additional information obtained during evaluation. In contrast to supercompilation, this information is represented as predicates (logical formulas), and the predicates are considered by means of an external theorem prover to achieve a better level of specialization.

Partial Deduction Partial evaluation has also been successfully employed for logic languages, where it is usually referred to as *partial deduction* [LS91]. Due to the process of unification applied in partial deduction, this method also provides a way of information propagation comparable to positive supercompilation.

In our work, we are interested in the development of a program transformation that is both generally applicable to arbitrary source programs, as well as fully automatic, i. e., it should not require any dynamic interaction with the user. Furthermore, it should be reasonably powerful, and preserve the semantics of the original program. Since the partial evaluation of functional logic programs can be seen as a generalization of partial deduction [AFV98] and thus is capable of achieving the same results as positive supercompilation [GS94], we will concentrate on the technique of partial evaluation in the following.

1.3 Partial Evaluation

Partial evaluation anticipates at compile time (or partial evaluation time) some parts of the computations that a program would normally perform at run time. From a technical point of view, partial evaluation takes as its arguments a program and parts of the program's input, and produces a new (specialized) program as the result. The program variables are classified as either *static* or *dynamic*, and the program is then executed with respect to the input to compute values for those parts that only depend on the static variables, and a *residual* program for those parts that also depend on the dynamic variables. The notion of input is by no means restricted to run time arguments, but may also denote concrete arguments of a specific function.

Example 1.1 (Partial Evaluation). *Consider the following implementation of a exponentiation function and its application to raise a number to the fourth power.*

```
square  x = x * x
even    x = (mod x 2) == 0
power n x = if n == 0
              then 1
              else if even n
                     then power (div n 2) (square x)
                     else x * (power (n - 1) x)
power4  x = power 4 x
```

In the expression `power 4 x`*, the value of* `x` *is unknown (dynamic), while the value* `n = 4` *is known (static). Based on this information, the process of partial evaluation can optimize the expression to yield the following residual (specialized) program:*

```
power4' x = let y = (x * x) in (y * y)
```

Since the conditionals and the recursion have been removed, the resulting program can be expected to run reasonably faster. Furthermore, the local binding for `y` *ensures that the same number of multiplications is performed as in the original program.*

Besides improvements in program efficiency, the concept of partial evaluation is also of theoretical importance in the field of compiler construction. For instance, by using a suitable partial evaluator, it is possible to construct a compiler by partial evaluation of an interpreter, a surprising effect first discovered by Futamura [Fut71]. Although we are not interested in the automatic construction of compilers in this work, we take this fact as an impressive evidence for the power of partial evaluation.

1.3.1 Online vs. Offline Partial Evaluation

A partial evaluator can be seen as a mixture of a compiler and an interpreter, since it both emits parts of the residual program and evaluates parts of the original program. Given a particular expression, the decision whether to emit a residual program fragment or to evaluate the expression requires some decision strategy. Regarding this strategy, partial evaluators can be classified as either *offline* or *online* partial evaluators.

Offline partial evaluators obtain information about the static data from a separate static analysis phase called *binding-time analysis*, where variables are classified as either *static* or *dynamic*. Based on this distinction, the partial evaluator decides whether to evaluate a variable and take its value into account (for static variables), or to ignore the value of a variable and produce residual code fragments (for dynamic variables).

Online partial evaluators obtain this information on the fly based on the initial static input, and propagate this information during the partial evaluation process. The decision whether to evaluate an expression or create a residual expression then depends on the current values of the involved variables.

The main advantage of *online* partial evaluation is its ability to exploit more static information compared to *offline* partial evaluation, which may lead to a more efficient residual program. On the other hand, offline partial evaluators can use a simpler specialization algorithm, and the additional guidance of the binding-time analysis may avoid useless specializations. Usually, offline partial evaluation is employed to approach *self-applicable* partial evaluators that allow the generation of compilers and compiler generators. For this objective, the offline approach provides more possibilities to reach effective specializations. To summarize, both approaches have their advantages and disadvantages, and even hybrid approaches exist [JGS93].

1.3.2 Control Restructuring

Another well-known dimension for the characterization of partial evaluators and comparable program transformations is their ability to restructure the control flow between the original program and its specialization. In this dimension, the relationship between program points (function definitions in our context) of the original and the specialized program are expressed by means of the following terminology [GS96].

Monovariant: Any program point in the subject program gives rise to zero or one program point in the residual program.

Polyvariant: Any program point in the subject program can give rise to one or more program points in the residual program.

Monogenetic: Any program point in the residual program is produced from a single program point of the subject program.

Polygenetic: Any program point in the residual program may be produced from one or more program points of the subject program.

In consequence, polyvariance allows more powerful specializations since the same function can be specialized to different instantiations, and polygenetic specializations reduce the size of the resulting program compared to monogenetic specializations.

1.4 Partial Evaluation of Functional Logic Languages

The technique of partial evaluation has been successfully applied to languages of different programming paradigms. In fact, a kind of partial evaluation is regularly

applied in the optimization of functional programs by creating specialized functions for specific arguments, and partial evaluation has also also been applied to logic programming under the name of partial deduction [LS91]. Not surprisingly, more recent research has been devoted to partial evaluation of functional logic languages in general, and Curry programs in particular. For instance, Alpuente, Falaschi, and Vidal [AFV98] proposed a generic partial evaluation scheme suitable for partial evaluation of functional logic languages based on narrowing as the underlying model of computation, and a practical implementation has been developed for the partial evaluation of Curry programs [AHV02].

However, at those times the development of Curry was in a considerably earlier stage, and many important features that are now contained in the language specification were not considered back then. For instance, the concept of non-deterministic operations, which may evaluate to more than one value for the same set of arguments, is nowadays deeply integrated but was not included back then. This prevents the application of the partial evaluator to contemporary Curry programs like the following.

Example 1.2 (Partial Evaluation of Functional Logic Program)**.** *Consider the following Curry program introducing a type* `Nat` *for natural numbers in their Peano representation as well as some operations on these numbers:*

```
 --- Natural numbers defined in Peano representation.
data Nat = Z | S Nat

--- Addition on natural numbers.
add Z     Z     = Z
add Z     (S n) = S n
add (S m) Z     = S m
add (S m) (S n) = S (S (add m n))

--- Is the given natural number even?
even Z         = True
even (S Z)     = False
even (S (S n)) = even n

--- double a natural number
double x = add x x

--- even or odd number close to input
eo x = x ? S x

-- main expression
main n = even (double (eo n))
```

Note that the operation `eo` *calls the non-deterministic operator "*`?`*", such that* `eo` *is non-deterministic itself and returns it argument either unchanged or incremented by one. Furthermore, the argument* `x` *of* `double` *occurs twice in the right-hand side, and we therefore expect both occurrences of* `x` *to evaluate to the same value. We now consider the expression* `even (double (eo n))`*, where* `n` *might be an arbitrary natural number. In this expression,* `n` *is either left unchanged and then doubled, or it is incremented by one and afterwards doubled. In consequence, we expect the value of* `double (eo n)` *to be even in both cases, and thus the expression* `main` *to return the result* `True` *for any input number* `n`*. If we apply the partial evaluator developed in this thesis, we obtain the following specialized program:*

```
main' :: Nat -> Bool
main' Z     = True ? True
main' (S n) = main' n
```

In the specialized program, the intermediate function calls have successfully been removed, and the operation `main'` *directly traverses its argument, resulting in a more efficient evaluation. Furthermore, since the resulting program is equivalent to the original program, it can also be seen as a proof for our expectation that the expression* `main` *will return* `True` *for any argument number. In this respect, the process of partial evaluation may also be used as a simple proof assistant.*

Since the above example uses non-deterministic operations, the demonstrated result can only be achieved by the partial evaluator developed in this thesis, but not the original partial evaluator of Albert, Hanus, and Vidal [AHV02]. In this case, their partial evaluator produces a specialized program which returns both `False` and `True` for any argument of `main`, and thus changes the semantics of the initial program. The main objective of this work therefore is the development of a partial evaluator for the functional-logic language Curry which supports the entire range of language constructs and is able to correctly consider non-deterministic operations.

1.5 Contributions

The superior objective of this work is the development of a partial evaluator for programs written in the functional logic language Curry, covering the entire range of language constructs. Since we are most interested in the *correctness* and *termination* of the partial evaluation process, we will follow the *online* partial evaluation approach. Note that this does not impede self-applicability in general, but there are little guarantees regarding the effectiveness of the specialization. The partial evaluator will furthermore be capable of producing both polyvariant and polygenetic specializations, enabling powerful and compact residual programs.

The development of a partial evaluator for a real-world language is a complex and challenging task, which is why we separate the development into several smaller steps that form the main contributions of this thesis.

Normalization Modern programming languages in general and Curry in particular provide a rich set of syntactic constructs to facilitate the notation of concise and expressive programs. During compilation, this complexity is usually reduced by means of a series of transformation steps. For many parts of the syntax, the applied transformation steps are (semi-)formally described in the language report of Curry [Han12]. However, the description therein considers each construct in isolation, and thus does not cover the intricacies of the combination of certain constructs. Furthermore, more recent language features are not considered at all. We therefore provide a formal, comprehensive description of the applied transformations, comprising all language features as well as the pitfalls of some of their combinations.

Operational Semantics A partial evaluator can also be regarded as a non-standard interpreter, in that it does not only compute values but also residual program fragments. It is therefore indispensable to have a clear notion of the dynamic semantics of Curry programs, and we present its operational semantics based on existing work. Usually, the operational semantics is defined for a subclass of programs that result from the elimination of certain kinds of nested expressions. Although this elimination preserves the semantics of programs, it sometimes impedes the readability, and we generalize the operational semantics to overcome this restriction. Furthermore, we extend the semantics to cover those primitive operations necessary for the implementation of advanced language constructs.

Residualizing Semantics Based on the extended semantics, we develop an adaptation thereof suitable for partial evaluation of Curry programs. This variant is capable of yielding residual code fragments in addition to normal values as a special kind of result, and also handles the occurrence of unknown (dynamic) input. We formally show that partial evaluation using this residualizing semantics does not change the semantics of expressions, although it anticipates some computation steps.

Partial Evaluation Scheme We then adapt the general partial evaluation scheme proposed by Alpuente, Falaschi, and Vidal [AFV98] for the partial evaluation of Curry programs and instantiate this scheme with the residualizing semantics. This instantiation is applicable for the partial evaluation of arbitrary Curry programs, and we formally prove its correctness as well as its termination.

Practical Implementation We accompany the development of the partial evaluation scheme with the implementation of a practical partial evaluator. While the partial evaluation scheme provides all necessary concepts, it is not sufficient to achieve a good level of specialization for realistic programs in general. Therefore, the implementation is extended to cover selected primitive functions, and to perform additional optimizations to achieve reasonably efficient specializations. Since the

partial evaluator itself is implemented in Curry and thus in the same language as the source programs to be specialized, this opens the door for possible self-applications, i. e., the partial evaluation of the partial evaluator itself, which might be an interesting topic for future research.

The residualizing semantics as well as some earlier results of the practical implementation have been previously published as "A Partial Evaluator for Curry" in the *Proceedings of the 23rd International Workshop on Functional and (Constraint) Logic Programming (WFLP 2014)* [HP14]. Their presentation have been thoroughly revised and accompanied by correctness and termination proofs in this thesis.

In summary, the central result of this thesis is the development of a practical, online partial evaluator for the functional logic language Curry with the following characteristics:

- It features the entire range of syntactic constructs of standard Curry, and parts of the extension of functional patterns [AH05].
- It correctly considers non-deterministic operations and sharing of arguments.
- The core algorithm is formally proven to be correct (i. e., the original and the specialized program are equivalent) and terminating (i. e., the algorithm terminates for any input program).
- The implementation is optimized to create small and efficient specializations.
- The architecture of the partial evaluator allows its easy integration into standard compilation chains for Curry programs.

1.6 Outline

This thesis is structured into three main parts, where the first part introduces the necessary foundations for the remainder of the thesis. The second part is then concerned with the normalization and operational semantics of Curry programs, while the third part covers the topic of partial evaluation.

Foundations In Chapter 2 we introduce to the reader the general concepts incorporated into the programming language Curry. We begin with those concepts originating from the field of functional programming, before we describe the additional concepts borrowed from logic programming, and discuss their integration.

We continue with a brief recapitulation of the topic of term rewriting in Chapter 3, and discuss the concept of narrowing and its applicability as the model of computation for functional logic programs.

Normalization and Operational Semantics The normalization process applied to Curry programs is described in Chapter 4, which discusses in detail the transformation steps applied to obtain a reduced representation of programs. The operational semantics of this representation is the topic of Chapter 5.

Partial Evaluation In Chapter 6 we present the preceding work of Alpuente, Falaschi, and Vidal [AFV98], who developed a partial evaluation framework usable with arbitrary narrowing strategies, and the results of a concrete instance for the evaluation of a deterministic subset of Curry programs.

Following their ideas, we continue with the development of a partial evaluation scheme suitable for the partial evaluation of Curry programs in Chapter 7, and provide theoretical results regarding its correctness and termination behavior.

We then instantiate this scheme to a practical implementation in Chapter 8, where we discuss different optimizations necessary to obtain reasonable efficient specializations. We accompany the implementation with a series of benchmarks documenting the usefulness of the developed partial evaluator.

We conclude the thesis in Chapter 9, where we summarize our achievements and give directions regarding possible improvements and further research topics. This work is completed by a reference to the context-free grammar of Curry in Appendix A, and by proofs for the claims stated in the main part in Appendix B.

Part I

Foundations

Chapter 2

Functional Logic Programming

The whole is greater than the sum of its parts.

Aristotle

Programming languages can be classified into different paradigms, which determine the fundamental style of how a program is written, structured and executed. Traditional programming languages such as C or Pascal follow the *imperative programming* paradigm, where a program can be considered as a sequence of instructions operating on a mutable state. In this paradigm, variables denote memory cells whose content can be changed using assignments, and typical abstraction mechanisms are procedures and loops. Therefore, a program contains a detailed description of *how* a problem should be solved by supplying all individual steps to reach the solution.

A popular extension of the imperative programming paradigm is the *object-oriented programming* paradigm, with well-known representatives such as Java or C++. Object-orientation extends imperative programming with the possibility to combine certain portions of data with the procedures operating upon them as single units called *objects*, where the structure of uniform objects is represented by their *class*. Furthermore, the mechanism of *inheritance* allows the refinement of both data structures and operations. The basic model of computation, however, is the same as for imperative programming.

In contrast, the paradigm of *declarative programming* tries to abstract from the steps that have to be performed to find a solution. This is achieved by not describing *how* a solution is to be found, but specifying the problem and *what characterizes a solution*. The task of finding the solution is then handed over to the language implementation. Amongst others, two major paradigms in the field of declarative programming are functional and logic programming, which are both based on a clean mathematical foundation. *Functional programming* with mature languages such as Haskell nowadays attracts increasing interest. Furthermore, fundamental concepts such as lambda expressions have found their way into mainstream languages such as Java 8, and new languages like Scala try to combine the object-oriented and functional programming paradigms in a single language. *Logic programming*, of which the language Prolog can be considered the most prominent representative, is based on a deduction calculus that allows the inference of new knowledge from established facts. Not surprisingly, logic languages are regularly used in the context of artificial intelligence.

While functional and logic programming feature different models of computation, their difference is not substantial, and more recent work on the combination of both paradigms led to the field of *functional logic programming*. In the remainder of this chapter, we describe these two paradigms as well as their integration by means of the functional logic language Curry. We begin in Section 2.1 with an introduction to the concepts of functional programming and their support for expressive and concise programs. We then extend this paradigm by the concepts originating from logic programming in Section 2.2, and discuss the implications of their integration.

2.1 Functional Programming

As we have already mentioned, imperative programs consist of a sequence of instructions that are subsequently executed to mutate a global state. In contrast, a functional program consists of a series of function definitions, and the evaluation of a functional program corresponds to the evaluation of an expression built from the functions previously defined. In this context, functions are considered to be functions in the mathematical sense, i. e., they do not perform any side effect and, thus, always return the same result for the same arguments. Consequently, the order in which the arguments are evaluated does not affect the result, which allows the application of different evaluation strategies. Furthermore, it is also safe to replace a function call with its result, which considerably simplifies the optimization of and reasoning about functional programs.

2.1.1 Algebraic Data Types and Functions

Programs written in the functional logic language Curry [Han12] basically consist of two kinds of declarations. *Data type declarations* introduce new types to the program, while *function declarations* introduce new functions that operate on values of the available types. For instance, a data type declaration for the type of Boolean values, representing the truth values *true* and *false*, is denoted in Curry as

```
data Bool = False | True
```

The declaration of a data type is initiated by the keyword "`data`" and followed by the name of the type, so that this declaration introduces the type `Bool` to the program. The right-hand side of the declaration consists of an enumeration of *data constructors*, which are alternatives to construct values of the declared type. Hence, `False` and `True` are the only values of type `Bool`. We follow the syntactic convention of the functional language Haskell in that we capitalize the names of types and data constructors, while the names of functions will be lowercased.[1] Based on the definition of Boolean

[1] The syntax of Curry is more flexible, since it also allows lowercase data types and uppercase functions, but this increased flexibility may also impede the readability of programs.

values, we can then define the negation function `not` that inverts the truth value of its argument.

```
not :: Bool -> Bool
not False = True
not True  = False
```

The first line of this *function declaration* specifies the *type signature* of the function, where the name and type of the function are separated by two colons. In this example, the signature states that `not` is a function that takes an argument of type `Bool` to produce a result of type `Bool`. The declaration of a type signature is generally optional, since the type of a function (and more generally of an expression) can be automatically deduced by a process called *type inference*.

Below the type signature follows a sequence of *equations* (or *rules*) that define the function, where each rule starts with the name of the function and a *pattern* describing the structure of the arguments to which a rule can be applied. When a function is called with a specific argument, an attempt is made to match the different patterns to the argument's structure (*pattern matching*), and the expression on the right-hand side is evaluated if the pattern on the left matches. For functions defined by a single rule, we will also refer to the expression on the right-hand side as the *function body*.

In Curry, the application of a function to an argument is denoted by juxtaposition, so that the expression `not True` denotes the application of function `not` to the argument `True`. For this function call, the second rule of `not` matches, and the result will be `False`. Since the patterns in the rules of `not` cover all possible values of the type `Bool`, `not` is said to be *totally defined*, in contrast to the following function which is only *partially defined*:

```
isTrue :: Bool -> Bool
isTrue True = True
```

If this function is called with the argument `False`, then the process of pattern matching fails, so that evaluation of the expression `isTrue False` fails as well.

A function that takes multiple arguments consists of rules that take multiple patterns. For instance, the disjunction and conjunction of Boolean values can be defined as follows:

```
(||) :: Bool -> Bool -> Bool
False || x = x
True  || _ = True

(&&) :: Bool -> Bool -> Bool
False && _ = False
True  && x = x
```

Since we use symbols instead of letters for the function names in this definition, they can be written in *infix notation*, and we may write `True && False` for an application. To regain a prefix notation, the syntax `(&&) True False` is also allowed.

In this example, the patterns do not only contain constructor patterns (`True`, `False`) like before, but also a *variable pattern* "`x`" and a *wildcard pattern* "`_`". While a constructor pattern only matches a specific constructor, a variable pattern matches every value, and in addition establishes a binding of the variable to the argument so that the argument can be referenced by the variable in the right-hand side. The wildcard pattern also matches every value but does not establish a binding, so that it denotes an unused argument.

When a function is defined using patterns for more than one argument, the order in which the patterns should be matched need to be determined. The defined strategy for Curry is to match those patterns first that discriminate the arguments the most, so that the following order of pattern matching is achieved:

1. Arguments where all patterns are constructor patterns
2. Arguments where at least one pattern is a constructor pattern
3. Remaining arguments

If several arguments fall into the same category, then the arguments are tried from left to right. Thus, in both the disjunction and the conjunction function, the first argument is evaluated before the second.

In addition to the simple data type declarations as introduced above, the constructors in a data type declaration may also take values of other types as additional arguments. Naturally, the declared type may occur on the right-hand side as well, which allows a recursive data declaration. For example, we can reuse the data type `Bool` to define a type representing simple propositional formulas consisting of Boolean constants, conjunctions and disjunctions:

```
data Formula = Const Bool | And Formula Formula | Or Formula Formula
```

Concrete formulas can then be built using the constructors `Const`, `And`, and `Or`:

```
formula :: Formula
formula = And (Const True) (Or (Const False) (Const True))
```

We can now define a function `eval` that evaluates the truth value of a given formula:

```
eval :: Formula -> Bool
eval (Const b) = b
eval (And f g) = eval f && eval g
eval (Or  f g) = eval f || eval g
```

In the last two rules, we combine the results of the recursive calls to `eval` using `(&&)` and `(||)`, respectively. Since the application of a function to an argument by juxtaposition binds tightest, we can omit the parentheses around the recursive calls.

In addition to algebraic data types, the Curry language also provides predefined types for integer numbers of arbitrary precision (type `Int`), floating point numbers (type `Float`) and characters (type `Char`). They feature the common operations, so that we can define the function `fib` computing the n-th Fibonacci number by

```
fib :: Int -> Int
fib n = if n <= 0 then 0
                  else if n == 1 then n
                                 else fib (n - 1) + fib (n - 2)
```

where the built-in conditional `if-then-else` selects one of the two alternative expressions based on the truth value of the condition. Note that operations testing the (in)equality or ordering of values of the same type are pre-defined for algebraic data types as well as numbers and characters.

Instead of using the conditional, we can also extend the left-hand side with conditions that constrain the applicability of a rule, which are denoted as follows:

```
otherwise :: Bool
otherwise = True

fib :: Int -> Int
fib n | n <= 0    = 0
      | n <= 1    = n
      | otherwise = fib (n - 1) + fib (n - 2)
```

Because the expressions on the right-hand side are only evaluated if the respective condition is satisfied, these conditions are also referred to as *guards*.

2.1.2 Type Polymorphism

Using the data type declarations encountered so far, we can define a data type for a list of Boolean values by the following declaration:

```
data BoolList = EmptyB | ConsB Bool BoolList
```

An empty list is represented by the constructor `EmptyB`, and the list containing the values `False` and `True` is represented as `ConsB False (ConsB True EmptyB)`. Likewise, we could define a similar type for lists of elements of another type, such as lists of numbers. However, this would directly lead to code repetition, both for the definition of the types as well as for some of the functions operating on them. For instance, the computation of the length of a list is independent of the type of its elements, so that it is desirable to define it only once for lists with elements of an arbitrary type.

For this purpose, data declarations can be parametrized by *type variables*, so that a single declaration may summarize several more concrete definitions. For instance, we can generalize the declaration of lists of Boolean values to lists with elements of an arbitrary type:

```
data List a = Empty | Cons a (List a)
```

Concrete list types can be constructed by supplying the type of the elements, for instance `List Bool` for a list of Boolean values. Since `List` requires a type as its argument to construct another type, it does not form a type but a *type constructor*.

Note that the type variable "a" occurs twice at the right-hand side, so that both occurrences will be replaced with the same concrete type. Thus, this definition only allows the construction of lists with elements that share the same type, like `Cons 1 (Cons 2 Empty)` or `Cons True (Cons False Empty)`, but prohibits lists containing elements of different types, like `Cons 1 (Cons True Empty)`.

Because the above declaration summarizes lists with different element types, it is said to be *polymorphic* in the type of the elements, and this type polymorphism [DM82] allows the reuse of common structures both in type and function declarations. For instance, using the generalized definition of lists, we can define the function `null` that computes whether a given list is empty or not as

```
null :: List a -> Bool
null Empty      = True
null (Cons _ _) = False
```

and this definition is applicable for lists with an arbitrary element type. Our previous list of Booleans is now a special case of the more general notion of arbitrary lists, and we can express this fact by specifying the *type synonym*

```
type BoolList = List Bool
```

so that the new name `BoolList` serves as a synonym for the type `List Bool`. Type synonyms and their definitions are perfectly interchangeable, and we can always replace the synonym with its definition and vice versa.

Because lists are so commonly used in functional programs, there exists a special built-in syntax that is much more concise. The empty list is denoted by `[]` while the non-empty list is denoted by the infix constructor `(:)`, so that a pseudo-definition would look like:

```
data [a] = [] | a : [a] -- built-in syntax
```

Furthermore, concrete lists of a finite length can be expressed like `[False, True]`, which is just a shorthand for the more verbose notation `False : (True : [])`. The built-in syntax can also be used in patterns, for instance to define the function `(++)` for list concatenation or the function `length` computing the length of a list:

```
(++) :: [a] -> [a] -> [a]
[]       ++ ys = ys
(x : xs) ++ ys = x : (xs ++ ys)

length :: [a] -> Int
length []       = 0
length (_ : xs) = 1 + length xs
```

Note that strings in Curry are just a synonym for a list of characters, so that we can compute the length of a string as `length "Curry"`.

In addition to lists of elements of a common type, it is often useful to group a fixed number of elements of different types, for which Curry provides a special *tuple*

syntax. For instance, the expression `(1, False)` denotes a tuple of type `(Int, Bool)` with `1` as its first and `False` as its second component, and the same syntax can be used for tuples with more than two components.

2.1.3 Higher-Order Functions

Until now, we have only considered functions that accept data values as their arguments and produce new data values as their results. In addition, it is also possible to define functions that take *functions as arguments* to be applied in their definition, or to yield a function as the result, a concept which is called *higher-order functions* [Hud89]. For instance, the well-known function `map` takes a function to be applied to all elements of a given list:

```
map :: (a -> b) -> [a] -> [b]
map _ []       = []
map f (x : xs) = f x : map f xs
```

The type signature specifies that the first argument must be a function of type `a -> b`, hence it must accept an argument of type `a` and yield a result of type `b`, and the second argument must be a list of elements of type `a`. If the list as the second argument is empty, then the result is the empty list as well. Otherwise, we have a non-empty list with `x` as the first element and `xs` as the remaining list. In this case, we apply the argument function `f` to `x`, proceed recursively for the remaining list `xs`, and combine both results using the list constructor `(:)`. We can then use the function `map` to negate a list of Booleans by:

```
> map not [False, True]
[True, False]
```

In contrast, an application such as `map length [True, False]` is prohibited by the type system, since the argument type `[a]` of the function `length` does not match the element type `Bool` of the list.

Besides the application of an argument function, a higher-order function may also yield a function as its result. Consider, for instance, the function composition `(.)` that resembles the mathematical function composition:

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
f . g = \x -> f (g x)
```

This function takes two functions as its arguments, namely `f` and `g`, and yields a new function as the result. The result function is expressed using a *lambda*[2] expression `\x -> f (g x)`, which denotes an *anonymous function* that takes `x` as its only parameter and then evaluates the expression `f (g x)`. We can use this definition to test whether or not the elements of a list of lists are non-empty:

[2]The term "lambda" refers to the lambda calculus, the underlying computation model of functional languages.

```
> map (not . null) [[], [True], [False]]
[False, True, True]
```

Until now, we have always supplied a function with all arguments at once, but this is not necessary in general. Instead, we can also supply the arguments one after another. For instance, consider the arithmetic addition `(+) :: Int -> Int -> Int` and the following examples:

```
> (+) 1 2
3
> ((+) 1) 2
3
```

While in the first example we provide both arguments at once, we do not in the second. Instead, we apply `(+)` only to its first argument, which yields a new (anonymous) function as the result, taking the second argument to increment it by `1`. The application of an n-ary function to less than n arguments is called a *partial application*, and the conversion of an n-ary function to a new $(n-1)$-ary function is called *Currying*[3]. In fact, a function (or more generally an expression) is *always* applied to a single argument only. Thus, the syntax "$f\ e_1\ e_2$" denotes the application of f to e_1 and the application of $f\ e_1$ to e_2, and therefore the same as $(f\ e_1)\ e_2$. However, since function application associates to the left, the parentheses can be omitted.

Using the concept of partial application, we can implement a function that filters out all negative numbers from a list of numbers as

```
nonNegative :: [Int] -> [Int]
nonNegative xs = filter (0 <) xs
```

where the expression "`(0 <)`" is a shorthand notation for "`(<) 0`", thus the partial application of `(<)` to `0`. This definition also uses the following higher-order function `filter`, which filters all elements from a list that satisfy a given predicate:

```
filter :: (a -> Bool) -> [a] -> [a]
filter p []                   = []
filter p (x : xs) | p x       = x : rest
                  | otherwise = rest
  where rest = filter p xs
```

In addition to the already known constructs, this definition uses a *local declaration* introduced by the keyword "`where`" to allow multiple references to a value using a variable name.

2.1.4 Evaluation Strategy

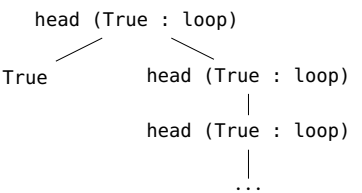
We have previously noted that the result of a function does only depend on the values of its arguments, and that the evaluation of expressions is free of side-effects.

[3]Named after Haskell Brooks Curry, who elaborated this concept, although it was originally discovered by Moses Schönfinkel. However, the term "Schönfinkeling" is less frequently used.

However, we have not discussed so far the order in which expressions are evaluated. This generally depends on the applied *evaluation strategy*, which is used to determine *when* to evaluate the arguments and *how* to pass the arguments to the called function. When a function call shall be evaluated, it is possible to first evaluate its arguments, which corresponds to a *strict evaluation*, or to start with the evaluation of the function itself, corresponding to a *non-strict evaluation*. To give an example, we consider the following functions

```
head :: [a] -> a
head (x : _) = x

loop :: a
loop = loop
```

where `head` selects the first element of a non-empty list, and `loop` denotes a non-terminating function. If we now consider the expression `head (True : loop)`, there are two possibilities how the expression can be evaluated:



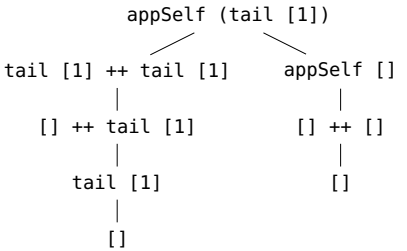
The strategy *call-by-name* selects the leftmost outermost function symbol, which is then evaluated independently of whether or not its arguments have already been evaluated before. This strategy corresponds to the left evaluation path in the tree, and starts evaluation of the initial expression by evaluation of the function `head`, which directly yields the result `True`. In contrast, the strategy *call-by-value* selects the leftmost innermost function symbol of an expression as the subexpression to be evaluated first. This strategy corresponds to the right evaluation path in the tree, where the function `loop` is repeatedly evaluated. Since `loop` evaluates to itself, evaluation fails to terminate in this example.

We can conclude from this example that there exist expressions for which the call-by-value strategy fails to compute a value, whereas the strategy call-by-name succeeds in a finite number of steps. Furthermore, it is well-known that the set of expressions for which the strategy call-by-name can compute a value is a proper superset of the set of expressions for which the strategy call-by-value can [Bar84].

Nevertheless, the strategy call-by-name is not always superior, since it may duplicate some computations and thus require more evaluation steps. Consider, for instance, the following function definitions where `appSelf` concatenates a list with itself and `tail` computes the tail of a list:

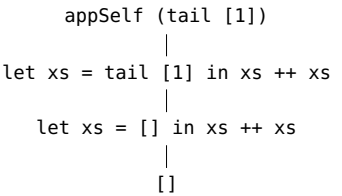
```
appSelf :: [a] -> [a]
appSelf xs = xs ++ xs

tail :: [a] -> [a]
tail (_ : xs) = xs
```

If we now evaluate the expression `appSelf (tail [1])` using call-by-name and call-by-value, we get the following evaluation steps:



We observe that the evaluation using call-by-name needs one additional step, since the subexpression `tail [1]` is evaluated twice. This is caused by the double occurrence of the variable `xs` in the right-hand side of `appSelf`, so that the unevaluated expression is duplicated in this example. While this does not influence the computed result, it may significantly affect the performance in case of more complex expressions.

This disadvantage is solved by the *call-by-need* strategy. This strategy evaluates expressions in the same order as call-by-name, but establishes references to expressions for function arguments, such that only references but not expressions are duplicated. We illustrate this strategy by the evaluation below, where we represent a reference by means of a `let` expression that introduces a local binding:



In this example, the variable `xs` is referenced more than once in the expression, and we thus say that (the binding of) the variable is *shared*. Note that call-by-need and call-by-value evaluation now require the same number of steps, and it is generally true that call-by-need requires at most the number of steps that call-by-value does, while the set of expressions that can be evaluated using call-by-need is the same as for call-by-name [HL91].

Since the call-by-need strategy avoids reevaluation of expressions and only evaluates expressions whose values are necessary to determine the final result, it is also referred to as *lazy evaluation* [Wad71], and has been chosen as the evaluation strategy of Curry. This evaluation strategy does not only influence the efficiency of programs, but also allows new programming patterns such as programming with infinite data structures [Hud89] due to its non-strictness. For instance, we can compute the infinite list of prime numbers using the sieve of Eratosthenes as follows. We first define a function that computes an infinite list of increasing numbers by

```
from :: Int -> [Int]
from n = n : from (n + 1)
```

We can then use this definition to compute all prime numbers by repeatedly removing all multiples of a prime number from the remaining list:

```
primes :: [Int]
primes = sieve (from 2)
  where sieve (p : ps) = p : sieve (filter (\q -> mod q p /= 0) ps)
```

To restrict the number of prime numbers to show, we can furthermore use the function `take :: Int -> [a] -> [a]` that takes at most the first n elements out of a list and ignores any subsequent elements:

```
> take 10 primes
[2,3,5,7,11,13,17,19,23,29]
```

2.1.5 Input/Output

Functional languages that disallow side-effects in the evaluation of expressions are also called *purely functional* languages. However, due to the absence of side-effects, the implementation of input/output operations in such languages is slightly more complicated than in imperative languages. While functions are generally required to yield the same results for the same arguments, this is not applicable for input/output operations, which are expected to return different results based on the current state of the machine a program is executed on. To overcome this problem, Curry uses the concept of monadic IO operations [PW93], which provides a clear separation of side-effects and pure (free of side-effects) computations. In Curry, every function that interacts with the outside world must be a computation of type `IO a`, where `IO` is a primitive type constructor and `a` is the type of the returned value. Such functions are often called *IO actions* to emphasize their difference from pure functions. For instance, there exists a predefined function

```
print :: a -> IO ()
```

that takes a value of an arbitrary type and prints it to the terminal via standard output. Since this action has no meaningful result, it returns a value of type `()`, pronounced as “unit”, whose only value is also `()`. Another action reading the contents of a file is

```
readFile :: String -> IO String
```

which takes the name of a file as its parameter and returns the file's content. IO actions can be sequentially composed by the infix operator

```
(>>=) :: IO a -> (a -> IO b) -> IO b
```

that takes as its parameters an action and a function yielding a second action. If executed, it will first execute the first action, apply the function to its result to get the second action, and execute it afterwards. Thus, we can define a new action that computes the size of a file and prints it to the user as

```
fileSize :: String -> IO ()
fileSize file = readFile file >>= \content -> print (length content)
```

IO actions are also executed lazily just like ordinary expressions, so that `readFile` as specified above will read the file character-wise on demand of the function `length`, and the entire action can be computed in constant memory space. To facilitate the notation of monadic actions, Curry furthermore provides a special syntax, the so-called `do`-notation, that resembles the imperative programming style:

```
fileSize :: String -> IO ()
fileSize file = do
  content <- readFile file
  print (length content)
```

Sometimes, it is also necessary to convert a pure expression into an action that returns its value, which can be achieved using the operation

```
return :: a -> IO a
```

For instance, we could extend our example above by first checking whether the file exists at all and returning a size of zero in case it does not:

```
fileSize :: String -> IO ()
fileSize fileName = do
  exists <- doesFileExist fileName
  if exists
    then do
      content <- readFile fileName
      print (length content)
    else print 0
```

By intention, there is no function to convert an IO action into a pure expression, so that side-effects can *only* occur in IO actions.[4]

[4] This is not entirely true since there exists an operation `unsafePerformIO :: IO a -> a`, but its usage is, as the name suggests, highly unsafe.

2.2 Extensions from Logic Programming

Another well-known branch of declarative programming is logic programming, which provides the concepts of non-deterministic operations, logic variables for unknown values, and a built-in search for solutions of a stated problem. Despite other variants, the integration of lazy functional programming and logic programming has gained notable attraction in research, and the functional logic language Curry was built on the theoretical results thereof. In the following, we will introduce the additional concepts originating from logic programming and discuss their integration into Curry.

2.2.1 Non-Determinism

Until now, we have only considered function declarations where at most one rule was applicable, so that the mechanism of pattern matching could deterministically identify the rule to be applied, if any. In general, there may be more than one applicable rule, and functional languages such as Haskell often choose the first one. In Curry, however, *all* applicable rules are tried non-deterministically, which may lead to multiple results. We consider as an example the operation `insert`, which non-deterministically inserts an element into a list at an arbitrary position.

```
insert :: a -> [a] -> [a]
insert x ys       = x : ys
insert x (y : ys) = y : insert x ys
```

While in Haskell the last rule would be ignored since the first one is always applicable, in Curry both rules are applied, such that the expression `insert 1 [2,3]` will yield the results `[1,2,3]`, `[2,1,3]`, and `[2,3,1]`. All three values are valid results, and in general the order in which the non-deterministic alternatives are evaluated can be arbitrary. Since `insert` yields multiple results for the same set of arguments, we will from now on use the term *operation* instead of function to emphasize that non-determinism is involved. Note that in general there also may exist non-deterministic alternatives that fail to yield a value, which can be explicitly represented by the primitive operation `failed`. Since `failed` does not establish a value, only the remaining alternatives are then further considered.

Whenever more than one rule is applicable for the same argument, we say that these applicable rules *overlap*. The main advantage of non-determinism by overlapping rules is its seamless integration into the language, since it does not require any additional data structures or combinator functions. We can, for instance, reuse the above definition to non-deterministically compute a permutation of a list.

```
permute :: [a] -> [a]
permute []       = []
permute (x : xs) = insert x (permute xs)
```

If we evaluate the expression `permute [1,2,3]`, we will get all six permutations non-deterministically as the result. In addition to the specification of overlapping rules, a non-deterministic operation can also be defined by the choice operator `(?)`, which is defined as

```
(?) :: a -> a -> a
x ? _ = x
_ ? y = y
```

This operator can then be used to define arbitrary non-deterministic expressions. For instance, we could use it to define the operation `coin` resembling the flipping of a coin:

```
coin :: Int
coin = 0 ? 1
```

Thus, every time we evaluate `coin`, we will non-deterministically get one of the results `0` and `1`.

2.2.2 Logic Variables and Narrowing

Another important concept in logic programming is the declaration of logic variables. While in functional languages variables can only be bound to expressions, Curry allows the declaration of logic (or free) variables by the keyword `free`. Conceptually, a logic variable represents all possible values (expressions built from constructors and literals) of a certain type. For instance, in the declaration

```
aBool :: Bool
aBool = x where x free
```

the logic variable `x` represents both Boolean values `False` and `True`. Logic variables are evaluated on demand, and can furthermore be instantiated during pattern matching. That is, if a logic variable is matched against a pattern, it is *instantiated* to a value representing the pattern, where variables in the pattern are again treated as logic variables. Thus, if we consider the function

```
head :: [a] -> a
head (x : _) = x
```

and evaluate the expression `let l free in head l`, then `l` will be instantiated to a non-empty list `(x : xs)`, where both `x` and `xs` are logic variables. Since this instantiation of a logic variable narrows the set of values it represents, this mechanism is called *narrowing* [Sla74] accordingly.

Because both logic variables and overlapping rules can lead to non-determinism, the question may arise how these concepts are related, and it turns out that they both can be expressed in terms of each other. Firstly, it is possible to transform every Curry program that utilizes overlapping rules to express non-determinism into an equivalent

program without overlapping rules, but with the use of a non-deterministic operator defined using logic variables [Ant01]:

```
(?) :: a -> a -> a
x ? y = ifThenElse b x y where b free

ifThenElse :: Bool -> a -> a -> a
ifThenElse False _ y = y
ifThenElse True  x _ = x
```

A later result [AH06] showed that the opposite is also possible, i. e., logic variables can be expressed by means of non-determinism with the help of generator functions. More precisely, every logic variable can be replaced by a generator function that non-deterministically evaluates to all values of the respective type. For instance, we can declare generator functions for Booleans as well as lists of Booleans as

```
aBool :: Bool
aBool = False ? True

aBoolList :: [Bool]
aBoolList = [] ? (aBool : aBoolList)
```

Note that the replacement of logic variables by generator functions requires a non-strict evaluation strategy, since otherwise the enumeration of Boolean lists would fail to terminate.

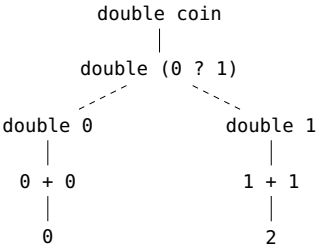
While the instantiation of logic variables by pattern matching is the default behavior in Curry, its application is not always reasonable, since there may exist operations that should be applied to deterministic values only, such as IO actions. For this purpose, there exists another approach for the handling of logic variables, namely *residuation*. The key idea of residuation is to suspend the evaluation of an expression that depends on the value of a logic variable, in hope that the variable might be instantiated by a concurrent evaluation. This behavior can, for example, be achieved by using the primitive operation `ensureNotFree :: a -> a`, which evaluates its argument and then suspends if the value of the argument is a logic variable.

2.2.3 Call-Time Choice

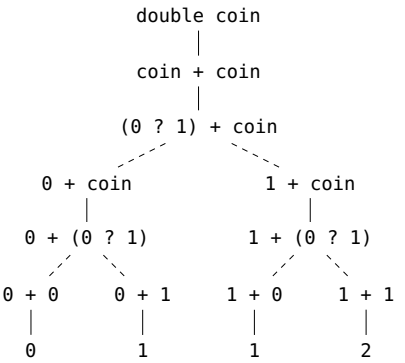
In the section about evaluation strategies, we have noticed that the strategies differ only in their efficiency, but not in the computed results (unless they fail to terminate). This, however, is no longer true in the context of non-deterministic expressions. Consider, for instance, the function

```
double :: Int -> Int
double x = x + x
```

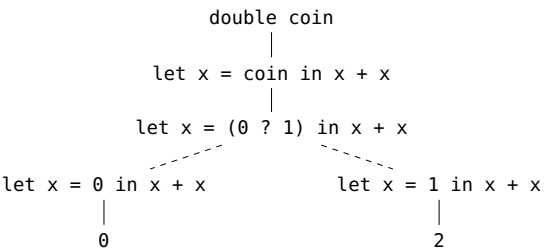
that doubles its argument, and the expression `double coin`. If we evaluate this expression using call-by-value, we obtain the following evaluations (we denote non-deterministic alternatives using dashed lines):



Thus, we obtain two non-deterministic results, `0` and `2`. If we evaluate the same expression using call-by-name, we get the following evaluations:



In contrast to the evaluation using call-by-value, we now obtain the additional result `1` twice. For the call-by-need strategy, we get the following evaluations:



This strategy again yields only two results, just like the call-by-value strategy.

The reason for the different results is obviously caused by the duplication of the non-deterministic expression `coin` in the call-by-name strategy, which is eliminated for the call-by-need strategy due to sharing. In consequence, the question arises at which point the implicit non-determinism in the variable `x` should be evaluated. The notion of *call-time choice* [HA77; Hus92] specifies that non-determinism in arguments should be evaluated when the function is called before evaluation of its body, so that the variables can be thought of as values instead of (unevaluated) expressions. Thus, call-time choice can be implemented by a call-by-value or call-by-need strategy. In contrast, the notion of *run-time choice* [HA77; Hus92] specifies that the non-determinism of arguments is independently evaluated when the function body gets evaluated, such that variables denote (possibly non-deterministic) expressions. As we have seen, this notion is implemented by the call-by-name strategy. The consensus in current functional logic languages such as $\mathcal{TOY}$ [LS99] or Curry is to follow the notion of *call-time choice*, since it implements the *principle of least astonishment*. This decision supports the more familiar concept of a variable denoting a value instead of an unevaluated expression, and thus simplifies the reasoning about programs.

One popular example that requires the ability to share the value of a non-deterministic expressions is the implementation of permutation sort, a very plausible but inefficient sorting algorithm. Permutation sort computes all permutations of a given list, and then returns the sorted permutations. Thus, using the `permute` operation defined above, we can define permutation sort as follows:

```
isSorted :: [Int] -> Bool
isSorted []           = True
isSorted [_]          = True
isSorted (x : y : ys) = x <= y && isSorted (y : ys)

permsort :: [Int] -> [Int]
permsort xs | isSorted p = p
  where p = permute xs
```

In this example, we expect the variable `p` to denote the *same* permutation of `xs` in both occurrences, which is only true under the call-time choice semantics.

2.2.4 Constraints

In addition to the narrowing of logic variables using pattern matching, logic variables can also be instantiated by means of *constraints*, which effectively constrain the set of possible values of a logic variable. For this purpose, Curry supports a structural equality constraint in the form of the strict unification operator:

```
(=:=) :: a -> a -> Success
```

Since we are only interested in satisfied constraints, the result of the unification operator is of the abstract type `Success` with the only value `success`. The strict

unification tries to instantiate both arguments to equal values, and either succeeds with the value `success` or fails otherwise. If none of the two arguments contains logic variables, then both arguments are evaluated and checked for equivalence. If one of the arguments, however, is a logic variable, then the variable is instantiated to the value of the other argument. Thus, we can constrain a logic variable `x` to the constructor `True` by the simple constraint `x =:= True`. Finally, if we want to ensure the equivalence of *two* logic variables `x` and `y` using the constraint `x =:= y`, then the strict unification may directly bind one of the variables to the other one instead of enumerating all possible bindings for `x` and `y`, which greatly improves the efficiency of the operation [BHP+13].

Multiple constraints can be combined using the concurrent constraint conjunction, which ensures that both constraints are satisfied:

```
(&) :: Success -> Success -> Success
```

Since the constraints are solved concurrently, this operator is regularly used in combination with residuation to implement a limited form of concurrency. Furthermore, an expression can be constrained to be valid only under a specified condition, which can be expressed using the conditional operator

```
(&>) :: Success -> a -> a
```

Thus, the expression `(x =:= 1 & y =:= 2) &> x + y` will first try to instantiate `x` to `1` and `y` to `2` in any order, and only if both unifications succeed then the sum of the variables is evaluated.

The combination of narrowing and constraints allows the application of interesting programming patterns, such as the *generate-and-test* principle. This principle tries to compute a solution for a problem by first computing the superset of possible solutions, and afterwards constraining the set to valid solutions. This allows the easy formulation of operations based on their properties instead of implementing a decent algorithm. For instance, to compute the last element of a given list, we could recursively traverse the list until we reach the final element:

```
lastRec :: [a] -> a
lastRec [x]          = x
lastRec (_ : y : ys) = lastRec (y : ys)
```

Alternatively, we can use the property that a value `x` is the last element of a list `l` if there exists a list `xs` such that the concatenation of `xs` and `[x]` yields `l`. This directly leads us to an implementation using an equality constraint:

```
lastUni :: [a] -> a
lastUni l | xs ++ [x] =:= l = x
  where xs, x free
```

In this example, we non-deterministically compute all non-empty lists by `xs ++ [x]`, and constrain them to those lists that equal the input list `l`. Due to the constraint, the variable `x` will be bound to the last element of `l`, which establishes our result.

2.2.5 Functional and Non-Linear Patterns

Although the above definition of the function `lastUni` using strict unification is rather concise, it has a serious drawback: it is unnecessarily strict, since it evaluates all elements of the list. The reason for this behavior is the usage of strict unification, which binds a variable to the *value* of an expression, so that the expression has to be evaluated first. Thus, we can compute the last element of lists such as `[1,2,3]`, but evaluation fails if it fails for any list element:

```
> lastUni [1,2,3]
3
> lastUni [failed, 0]
*** No value found!
```

This is an unsatisfactory situation, since we have traded conciseness against reduced applicability. Fortunately, there exists a solution for this problem in the form of *functional patterns* [AH05]. A functional pattern allows the matching of an argument against the structure of a value that originates from the evaluation of a function call. For instance, we can reformulate the function `last` using a functional pattern to the following definition:

```
lastFP :: [a] -> a
lastFP (xs ++ [x]) = x
```

In this example, we have moved the shape-defining expression `xs ++ [x]` to the left-hand side. Conceptually, a functional pattern abbreviates a possibly infinite set of patterns, where the patterns are obtained from evaluating the functional pattern as an expression with the variables in the function call considered as logic variables. Since the expression `xs ++ [x]` will evaluate to the set of all non-empty lists, the definition can thus be seen as an abbreviation for the following infinite set of equations:

```
lastFP :: [a] -> a
lastFP [x]        = x
lastFP [x1, x]    = x
lastFP [x1, x2, x] = x
⋮
```

The main difference to the implementation of `lastUni` is that the variables originating from a functional pattern now behave like normal pattern variables, and thus can be bound against expressions instead of values like in strict unification. In consequence, in `lastFP` the evaluation of the argument is now limited to the list structure, so that we can compute the last element even in case of undefined list elements:

```
> lastFP [failed, 0]
0
```

Another extension of pattern matching is the specification of *non-linear patterns*, so that a single variable may occur more than once in the left-hand side of a function rule. After the normal process of pattern matching, the arguments bound by the same variable are then checked for equivalence before evaluation continues with the right-hand side. For instance, the function

```
sumSame :: (Int, Int) -> Int
sumSame (x, x) = x + x
```

computes the sum of the components of a tuple only if both components contain the same value, and fails otherwise. Functional and non-linear patterns can be elegantly combined to express very concise function declarations. Consider, for instance, an operation `lookup` that non-deterministically searches for a value in a list of key-value pairs using a given key. Traditionally, this can be expressed by a combination of pattern matching and Boolean equality:

```
lookup :: k -> [(k, v)] -> v
lookup k ((k', v) : kvs) | k == k'   = v ? lookup k kvs
                         | otherwise = lookup k kvs
```

Using functional and non-linear patterns, we can instead express this operation by

```
lookup :: k -> [(k, v)] -> v
lookup k (_ ++ [(k, v)] ++ _) = v
```

where the comparison of the keys is expressed by the double occurrence of `k`, and the traversal of the list is expressed in the functional pattern.

2.2.6 Encapsulated Search

We deliberately applied the built-in non-determinism of Curry to encode different solutions for the same expression, and then used the interactive run time system to list the individual solutions. In logic languages such as Prolog, the different solutions are searched for using a depth-first strategy. However, although this strategy can be efficiently implemented, it is known to be incomplete since it may be stuck in infinite search branches. The Curry system KiCS2 has overcome this drawback by supplying multiple search strategies, for instance iterative deepening or breadth-first-search [HPR12]. Nevertheless, the top-level search is not sufficient to gather information about non-deterministic solutions *inside* the program, for instance to answer question like

- How many solutions exist for a given problem?
- What is the best solution for a given problem?

Without further language support, such questions could not be answered in a single program, which is why Curry supports the concept of *encapsulated search*.

Encapsulated search allows the user to search for all solutions of a given expression *inside* a Curry program and to inspect the solutions thereafter, for instance as a list of results. An important requirement for encapsulated search is that the result must be independent of the evaluation order, since otherwise the processing of the result may affect its value. Unfortunately, it appeared to be surprisingly hard to meet this requirement, since the order of the solutions naturally depends on the search strategy. Current Curry systems therefore provide two different concepts for encapsulated search.

The solution proposed by Braßel, Hanus, and Huch [BHH04] introduces a primitive operation `getAllValues :: a -> IO [a]` which returns a list of solutions inside the `IO` monad, since it does not guarantee any order of the solutions. Furthermore, it is possible to retrieve the abstract search tree corresponding to a non-deterministic evaluation in order to traverse it with the help of a user-defined search strategy.

The second solution is the concept of *set functions* [AH09], which accompanies every operation f with an associated set function f_S. The set function f_S then encapsulates the non-determinism introduced by the operation f itself, but not the non-determinism introduced by the arguments. This behavior is referred to as *weak encapsulation*, in contrast to *strong encapsulation* where non-determinism is encapsulated independent of its origin. The weak encapsulation mechanism ensures that each non-deterministic set of results does not depend on the order in which f_S and its arguments are evaluated. Furthermore, the solutions are returned as an abstract value that does not allow the inspection of their order, so that the result is guaranteed to be independent of the chosen search strategy.

Generally, the concept of encapsulated search can be seen as a way of meta-programming, since it allows the programmer to reason about the non-deterministic results of an expression. In this thesis, we will not further discuss the possibilities of encapsulated search and concentrate on the fundamentals of the language Curry, but nonetheless wanted to illustrate the potential of the language.

2.3 Summary

In this chapter we have discussed the concepts of both functional and logic programming, as well as their tight integration in the multi-paradigm language Curry. The history and development of today's functional logic programming languages is described in detail in a survey article [Han13], with a focus on the multi-paradigm language Curry.

The concepts originating from functional programming enable the programmer to write compact, modular, and reusable code. Algebraic data types allow the concise definition of new data structures that can be easily processed by functions using pattern matching. Type polymorphism allows to abstract from concrete types to enable the re-usage of functions, and higher-order functions further allow us to invent

new abstractions for common control patterns. Lazy evaluation builds the basis both for efficient computations as well as the new programming pattern of infinite data structures. Finally, the concept of monadic IO helps to cleanly separate pure functions and actions performing side-effects, which facilitates equational reasoning and program transformations. The importance of the functional programming paradigm has already been advocated in 1989 in the article "Why Functional Programming Matters" of John Hughes [Hug89]. Since then, the idea has been increasingly spread and adopted, and the success of functional programming has just recently been described in a subsequent article [HHW15].

The concepts of logic programming, namely computation with unknown values and the coexistence of multiple solutions by non-determinism, further contribute the ability to abstract from concrete algorithms by only specifying the properties a solution should fulfill. Additional concepts such as equational constraints or functional patterns once again enlarge the repertoire of instruments to develop concise programs. Although the combination of the functional and logic paradigms can be quite delicate in detail, it increases the flexibility in programming and the expressiveness of programs.

Chapter 3

Term Rewriting and Narrowing

Formalism is music that people don't understand at first hearing.

Sergei Prokofiev

In this chapter we present the basic notions necessary for the theoretical developments in the remainder of this thesis. We start with the concepts and notations of signatures, terms and substitutions to establish a precise terminology, and continue with a presentation of term rewriting and narrowing. We then provide a detailed description of the *needed narrowing* strategy, and discuss its applicability and limitations in the context of functional logic languages. In our formal presentation, we follow the usual notations [BN98; DJ90; Klo92].

3.1 Signature and Terms

A (many-sorted) *signature* Σ is a set of function (or operation) symbols, where each $f \in \Sigma$ is associated with a non-negative integer n, called the *arity* of f. For $n \geqslant 0$, we denote the set of all n-ary elements of Σ by $\Sigma^{(n)}$, and we define the auxiliary operation arity to yield the arity of a function symbol, i. e., $\mathsf{arity}(f) = n$ if and only if $f \in \Sigma^{(n)}$. A *constructor-based signature* Σ is a disjoint union of two sets of symbols $\mathcal{C} \cup \mathcal{F}$, the set $\mathcal{C}$ of constructor symbols and the set $\mathcal{F}$ of (defined) function or operator symbols. By $\mathcal{C}^{(n)}$ (resp. $\mathcal{F}^{(n)}$), we will denote the subset of all n-ary elements of $\mathcal{C}$ (resp. $\mathcal{F}$). We will use $c, d, c_1, \ldots, c_n$ for constructor symbols, $f, g, f_1, \ldots, f_n$ for function symbols, and $\phi, \psi, \phi_1, \ldots, \phi_n$ for arbitrary symbols in Σ. In addition, we will refer to $\phi \in \Sigma^{(0)}$ as a *constant*.

In general, if we refer to a *sequence of objects* $o_1, \ldots, o_n$, we will use the notation $\overline{o_n}$, and may omit the index if the exact length of the sequence is arbitrary, i. e., we may write $\overline{o}$ in this case. Furthermore, we will write $o \in \overline{o_n}$ to denote the test whether o is contained in the sequence $\overline{o_n}$. The empty sequence will be denoted by ε, and by $\overline{o_m} \cdot \overline{o'_n}$ we denote the composition $o_1, \ldots, o_m, o'_1, \ldots, o'_n$ of the two sequences $\overline{o_m}$ and $\overline{o'_n}$. Furthermore, we will use the notation $o \cdot \overline{o_n}$ for the (de)construction of the sequence $o, o_1, \ldots o_n$, i. e., the addition or removal of the first element of a sequence. Finally, we will write $\bigcup \overline{os_n}$ to denote the composition $os_1 \cdot \ldots \cdot os_n$ of multiple sequences.

In the following, we assume a fixed set $\mathcal{V}$ of *variables* with $\mathcal{V} \cap \Sigma = \emptyset$, and we denote variables by $x, y, z, x_1, \ldots, x_n, y_1, \ldots, y_n$. The set of *terms*, denoted by $\mathcal{T}(\Sigma, \mathcal{V})$, is inductively defined as the smallest set satisfying the following conditions:

- $\mathcal{V} \subseteq \mathcal{T}(\Sigma, \mathcal{V})$ (every variable is a term),
- for all $n \geqslant 0$, all $\phi \in \Sigma^{(n)}$, and all $t_1, \ldots, t_n \in \mathcal{T}(\Sigma, \mathcal{V})$, we have $\phi(\overline{t_n}) \in \mathcal{T}(\Sigma, \mathcal{V})$ (the application of an n-ary symbol to n terms yields a term).

The set of *constructor terms*, i. e., the set of terms only constructed from variables and constructor symbols, is denoted by $\mathcal{T}(\mathcal{C}, \mathcal{V})$ and defined accordingly. In addition, we will omit the parentheses for constants, i. e., we will simply write ϕ instead of $\phi()$. The set of variables occurring in a term t, denoted by $\mathcal{V}ar(t)$, is defined as

$$\mathcal{V}ar(t) = \begin{cases} \{t\} & \text{if } t \in \mathcal{V} \\ \bigcup_{i=1}^{n} \mathcal{V}ar(t_i) & \text{if } t = \phi(\overline{t_n}) \end{cases}$$

and we call a term t *ground* if $\mathcal{V}ar(t) = \emptyset$. Furthermore, a term t is called *linear* if it does not contain multiple occurrences of the same variable. A *pattern* is a term of the form $f(\overline{t_n})$, where $f \in \mathcal{F}^{(n)}$ and $t_1, \ldots, t_n \in \mathcal{T}(\mathcal{C}, \mathcal{V})$. The *root symbol* of a term refers to the outermost symbol of a term. A term is *operation-rooted* (*constructor-rooted*), if it has an operation symbol (constructor symbol) as the root symbol.

A *position* p in a term t is represented by a sequence of natural numbers, where the empty sequence ε refers to the root position of the term. For a given term t, the set $\mathcal{P}os(t)$ of all positions in t is inductively defined as follows:

$$\mathcal{P}os(t) = \begin{cases} \{\varepsilon\} & \text{if } t \in \mathcal{V} \\ \{\varepsilon\} \cup \bigcup_{i=1}^{n} \{i \cdot p \mid p \in \mathcal{P}os(t_i)\} & \text{if } t = \phi(\overline{t_n}) \end{cases}$$

Positions are ordered by a *prefix ordering* defined as

$$p \leqslant q \quad \text{if and only if there exists a sequence } p' \text{ such that } p \cdot p' = q$$

and we say that p is *over* q if $p \leqslant q$. Two positions p and q are called *disjoint* if neither $p \leqslant q$ nor $q \leqslant p$, and a position p is *left of* (*right of*) q if $p = p_0 \cdot i \cdot p_1$ and $q = p_0 \cdot j \cdot q_1$ with $i < j$ ($i > j$). For a term t and a position $p \in \mathcal{P}os(t)$, the *subterm* of t at the position p, denoted by $t|_p$, is inductively defined as

$$t|_\varepsilon = t$$
$$\phi(\overline{t_n})|_{i \cdot p} = t_i|_p$$

Furthermore, for a term t and a position $p \in \mathcal{P}os(t)$, the *replacement* of the subterm at position p by a term s, denoted by $t[s]_p$, is defined as

$$t[s]_\varepsilon = s$$
$$\phi(\overline{t_n})[s]_{i \cdot p} = \phi(t_1, \ldots, t_{i-1}, t_i[s]_p, t_{i+1}, \ldots, t_n)$$

We extend both notations to sequences of terms, where $\overline{t_n}|_{i \cdot p} = t_i|_p$ and $\overline{t_n}[s]_{i \cdot p} = t_1, \ldots, t_{i-1}, t_i[s]_p, t_{i+1}, \ldots, t_n$. A Σ-*context* is a term $t \in \mathcal{T}(\Sigma, \mathcal{V} \cup \{\Box\})$ possibly containing occurrences of the special symbol $\Box \notin \Sigma \cup \mathcal{V}$, which denotes a *hole* (empty place) in the term. A context is generally denoted by C, and if $\{p_1, \ldots, p_n\} = \{p \in \mathcal{P}os(C) \mid C|_p = \Box\}$ where p_i is left of p_{i+1} for all $i \in \{1, \ldots, n-1\}$, then $C[t_1, \ldots, t_n]$ denotes the term $C[t_1]_{p_1} \ldots [t_n]_{p_n}$.

3.2 Substitutions

A *substitution* $\sigma : \mathcal{V} \to \mathcal{T}(\Sigma, \mathcal{V})$ is a mapping from a (countably infinite) set of variables to terms such that only for a finite subset of variables we have $\sigma(x) \neq x$. We will denote a concrete substitution by giving the explicit mappings for the variables. For instance, $\sigma = \{\overline{x_n \mapsto t_n}\}$ with $x_i \neq x_j$ for $i \neq j$ denotes a substitution such that $\sigma(x_i) = t_i$ for all $i \in \{1, \ldots, n\}$ and $\sigma(x) = x$ for all $x \notin \overline{x_n}$. We denote by $\mathcal{D}om(\sigma) := \{x \mid \sigma(x) \neq x\}$ the *domain* of σ, and by $\mathcal{R}an(\sigma) := \{\sigma(x) \mid \sigma(x) \neq x\}$ the *range* of σ. We say that a substitution σ *instantiates* a variable x if $x \in \mathcal{D}om(\sigma)$, and we will use the name *id* for the identity substitution.

A substitution σ can be extended to a mapping $\sigma^* : \mathcal{T}(\Sigma, \mathcal{V}) \to \mathcal{T}(\Sigma, \mathcal{V})$ from terms to terms as follows:

$$\sigma^*(x) = \sigma(x)$$
$$\sigma^*(\phi(\overline{t_n})) = \phi(\overline{\sigma^*(t_n)})$$

If it is clear from the context, we will call the extension σ^* of σ also a substitution and denote it by σ as well. The *composition* of two substitutions σ and θ, denoted by $\sigma \circ \theta$, is defined as

$$\sigma \circ \theta(x) = \sigma^*(\theta(x))$$

A substitution σ is called a *variable substitution* if $\sigma(x) \in \mathcal{V}$ for all $x \in \mathcal{D}om(\sigma)$, and an injective variable substitution is called a *variable renaming*. Furthermore, a substitution is called (ground) constructor if $\sigma(x)$ is a (ground) constructor term for all $x \in \mathcal{D}om(\sigma)$. The *restriction* of the domain of a substitution σ to a set of variables $V \subseteq \mathcal{V}$ is denoted by $\sigma_{\restriction V}$. We say that two substitutions θ and σ are equivalent with respect to a set of variables V, denoted by $\theta = \sigma[V]$, if $\theta_{\restriction V} = \sigma_{\restriction V}$. A substitution σ is called *more general* than a substitution σ', denoted by $\sigma \leqslant \sigma'[V]$, if there exists a substitution γ such that $\gamma \circ \sigma = \sigma'[V]$. If $\sigma \leqslant \sigma'[V]$ for $V = \mathcal{V}$, i. e., the set of all variables, then we simply write $\sigma \leqslant \sigma'$.

A term t' is called an *instance* of a term t if there exists a substitution σ such that $t' = \sigma(t)$. This leads to a *subsumption ordering* on terms that is defined as

$$t \leqslant t' \quad \text{if and only if } t' \text{ is an instance of } t$$

Furthermore, if $t \leqslant t'$, we say that t is *more general* than t'. A term t is called a *variant* of a term t' if there exists a variable renaming σ such that $t = \sigma(t')$. A *unifier* of two

terms s and t is a substitution σ such that $\sigma(s) = \sigma(t)$. A *most general unifier* (*mgu*) for two terms is a unifier σ such that $\sigma \leqslant \sigma'$ for every other unifier σ', and a mgu for two terms is unique up to variable renaming [LMM88].

3.3 Term Rewriting

A *rewrite rule* $l \to r$ is an ordered pair of two terms l and r such that l is not a variable and $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l)$. The terms l and r are called the *left-hand side* and *right-hand side* of the rule, respectively. A *term rewriting system* (*TRS*) is a set of rewrite rules. A TRS $\mathcal{R}$ is called *left-linear* if the left-hand side l of every rule $l \to r \in \mathcal{R}$ is linear, and a TRS is *constructor-based* if every left-hand side l is a pattern. In the following, we will only consider finite, left-linear and constructor-based term rewriting systems.

Two (possibly renamed) rewrite rules $l \to r$ and $l' \to r'$ are said to *overlap* if there exists a non-variable position $p \in \mathcal{P}os(l)$ and a most general unifier σ such that $\sigma(l|_p) = \sigma(l')$. In this case, the pair $\langle \sigma(r), \sigma(l)[\sigma(r')]_p \rangle$ is called a *critical pair*. A critical pair $\langle t, t' \rangle$ is *trivial* if $t = t'$. A left-linear TRS without critical pairs is called *orthogonal*, and if it only has trivial critical pairs, it is called *weakly orthogonal*.

A *rewrite step* (*reduction step*) is an application of a rewrite rule to a term t, denoted by $t \to_{p,R} t'$, if there exist a position p in t, a rewrite rule $R = l \to r$ and a substitution σ with $t|_p = \sigma(l)$ and $t' = t[\sigma(r)]_p$. We will usually omit the position p and rewrite rule R in the notation when appropriate. By $\to^+$ we denote the transitive closure of $\to$, and by $\to^*$ its reflexive and transitive closure. The subterm $t|_p$ that matches the instantiated left-hand side $\sigma(l)$ is called a *reducible expression*, or a *redex* for short. A term t is called *root-stable* if it cannot be rewritten to a redex. A *constructor root-stable* term is either a variable or a constructor-rooted term, and we will abuse the notion of a *head normal form* to denote a constructor root-stable term. A term t is *irreducible* or in *normal form* if no term t' such that $t \to t'$ exists.

3.4 Narrowing

The process of narrowing [Sla74] can be seen as a generalization of term rewriting where the matching process is replaced by unification, such that the term to be rewritten gets instantiated. Formally, $t \leadsto_{p,R,\sigma} t'$ is a *narrowing step* if p is a non-variable position in t, $R = l \to r$ is a rewrite rule, σ is a mgu for $t|_p$ and l so that $\sigma(t|_p) = \sigma(l)$, and $t' = \sigma(t[r]_p)$. If the applied rewrite rule (and the position) are irrelevant, we may also write $t \leadsto_{p,\sigma} t'$ or $t \leadsto_\sigma t'$. In consequence, the substitution σ instantiates some of the variables in t such that the rewrite rule R becomes applicable, and the concept of narrowing generally combines the reduction of a term with an instantiation of (some of) its variables.

We denote by $t_0 \leadsto^*_\sigma t_n$ a sequence of narrowing steps $t_o \leadsto_{\sigma_1} \ldots \leadsto_{\sigma_n} t_n$ with $\sigma = \sigma_n \circ \ldots \circ \sigma_1$ and $\sigma = id$ for $n = 0$. In the process of narrowing, we are gener-

ally interested in the computation of *values* (constructor terms) as well as *answers* (substitutions), and we say that $t \leadsto_\sigma v$ computes the value v with answer σ if v is a constructor term. The set of narrowing sequences (or derivations) originating from a single term t can be represented by a (possibly infinitely) branching tree rooted by t.

Definition 3.1 (Narrowing Tree)**.** *Given a term rewriting system $\mathcal{R}$ and a term t, a narrowing tree T for t in $\mathcal{R}$ is a (possibly infinite) directed rooted node- and edge-labeled graph built as follows:*

- *The root node is labeled with t.*
- *If C is a node labeled with t' and $t' \leadsto_{p_1,R_1,\sigma_1} t'_1, \ldots, t' \leadsto_{p_n,R_n,\sigma_n} t'_n$ are all possible narrowing steps for t', then either*
  - *C has no child nodes (incomplete path), or*
  - *C has an edge labeled with (p_i, R_i, σ_i) to a child node C_i labeled with t'_i for all $i \in \{1, \ldots, n\}$.*
- *If a node C is labeled with a term t' and there is no narrowing steps $t' \leadsto_{p'',R'',\sigma''} t''$, then C has no children.*

The set of narrowing derivations in a narrowing tree T is the set of paths starting at the root that do not end at an inner node.

A single derivation can potentially be infinite, failed, successful or incomplete. A *failing derivation* ends with a term that is neither a constructor term nor can it be narrowed any further, while a successful (incomplete) derivation ends with a value (term that can be further narrowed).

The definition of narrowing is highly non-deterministic in general, since it allows arbitrary variable instantiations in the term to be evaluated. Therefore, in principle all possible instantiations must be tried in order to compute all possible values and answers. This advocates the usage of a *narrowing strategy*, selecting the position of a single non-variable subterm for the next narrowing step.

Definition 3.2 (Narrowing Strategy)**.** *A narrowing strategy is a function from terms to sets of triples. If $\mathcal{S}$ is a narrowing strategy, t a term and $(p, R, \sigma) \in \mathcal{S}(t)$, then $t \leadsto_{p,R,\sigma} \sigma(t[r]_p)$ is a narrowing step for the position p of t, rewrite rule $R = l \to r$, and substitution σ.*

Naturally, narrowing with respect to a certain narrowing strategy should at least be *sound*, i. e., it should compute only correct solutions, as well as *complete*, i. e., it should compute all solutions or more general representations of the solutions. There exists a multitude of narrowing strategies (see [Han94] for a survey), each requiring specific conditions on the considered rewriting systems. Of these strategies, *(outermost) needed narrowing* is a very promising strategy due to its optimality results, and it is furthermore suitable for the evaluation of functional logic languages.

3.4.1 Needed Narrowing

The basic idea of *(outermost) needed narrowing* [AEH00] is to perform a narrowing step only if this step is necessary to compute a result, so that needed narrowing implements a *demand-driven* narrowing strategy. To evaluate a term, it analyzes the left-hand sides of the rewrite rules of the outermost operation in the term. If there exists an argument position at which all left-hand sides are constructor-rooted, the corresponding actual argument must also be constructor-rooted in order to apply a rewrite step. Thus, the corresponding argument is evaluated to its head normal form and if this head normal form is a variable, it is non-deterministically instantiated with some constructor applied to fresh variables.

Since not all TRSs allow this kind of analysis of left-hand sides, needed narrowing is defined only for the subclass of *inductively sequential* TRSs, which can be characterized by *definitional trees*. Intuitively, a definitional tree is a hierarchically ordered set of patterns, representing the sequential matching of a specific pattern.

Definition 3.3 (Partial Definitional Tree [Ant92])**.** *$\mathcal{T}$ is a partial definitional tree, or pdt, if and only if one of the following cases holds:*

- *$\mathcal{T} = branch(\pi, p, \overline{\mathcal{T}_k})$ where π is a pattern, p is the position of a variable of π, $\overline{\mathcal{T}_k}$ is a sequence of pdts, and for all $i \in 1, \ldots, k$ the pdt $\mathcal{T}_i$ has a pattern $\pi[c_i(\overline{x_{n_i}})]_p$, where n_i is the arity of c_i, $\overline{x_{n_i}}$ are new distinct variables and $\overline{c_k}$ are distinct constructors.*
- *$\mathcal{T} = rule(\pi, l \to r)$ where π is a pattern and $l \to r$ is a rewrite rule such that $l = \pi$ (modulo variable renaming).*
- *$\mathcal{T} = exempt(\pi)$ where π is a pattern.*

In this definition, branch, leaf, and exempt are uninterpreted functions classifying the tree nodes.

Definition 3.4 (Definitional Tree [Ant92])**.** *$\mathcal{T}$ is a definitional tree of an operation f if and only if $\mathcal{T}$ is a pdt with $f(\overline{x_n})$ as the pattern argument, where n is the arity of f and $\overline{x_n}$ are new variables.*

Note that this definition allows the construction of infinite definitional trees, containing an infinite number of *exempt* nodes. We therefore follow a common convention and require definitional trees to not contain *branch* nodes with *exempt* nodes as their only children.

Definition 3.5 (Inductively Sequential (according to [Ant92]))**.** *We call an operation f of a term rewrite system $\mathcal{R}$ inductively sequential if and only if there exists a definitional tree $\mathcal{T}$ of f such that the rules contained in $\mathcal{T}$ are all and only the rules defining f in $\mathcal{R}$. We call a term rewrite system $\mathcal{R}$ inductively sequential if and only if any operation of $\mathcal{R}$ is inductively sequential.*

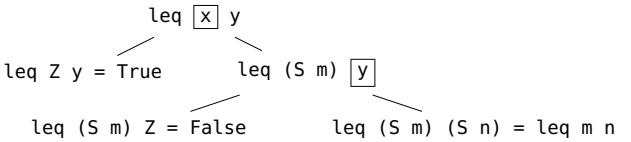


Figure 3.1. Definitional Tree for the Operation `leq`

In a definitional tree, a *branch* node represents a discrimination between different patterns based on the constructor at a certain position, a *leaf* node represents the application of a rule after pattern matching, and an *exempt* node denotes missing patterns in case of incompletely defined operations. Note that every inductively sequential TRS is also constructor-based and orthogonal [Ant92], but not vice versa.

To give an example, we consider the following term rewriting system defining an operation "less than or equal to" based on a Peano representation of natural numbers. For simplicity, we denote concrete term rewriting systems in the syntax of Curry.

```
leq Z     y     = True
leq (S m) Z     = False
leq (S m) (S n) = leq m n
```

We show a graphical representation of the corresponding definitional tree in Figure 3.1, where leaves are marked with a variant of the corresponding rule, branches are marked with the corresponding pattern, and the inductive position is surrounded by a box. Note that there can exist more than one definitional tree for a single function if there exist more than one inductive position, and we assume that the leftmost inductive position is taken in this case.

Based on the notion of definitional trees, we proceed with an informal description of the needed narrowing strategy [Han13], before we provide its formal definition. To evaluate a term t rooted by an operation f, the definitional tree $\mathcal{T}$ for the operation f (applied to fresh variables) is considered. This tree is recursively traversed from the root downwards to find the maximal pattern unifiable with t, where at each level of the tree the following decision is made based on the tree node and the structure of the term t:

- If $\mathcal{T} = \mathit{rule}(\pi, l \to r)$, then we apply the corresponding rule.
- If $\mathcal{T} = \mathit{exempt}(\pi)$, then the term cannot be narrowed to a constructor-rooted term, and narrowing fails.
- If $\mathcal{T} = \mathit{branch}(\pi, p, \overline{\mathcal{T}_k})$ with the inductive position p, we consider the subterm $t|_p$.
  - If $t|_p$ is rooted by a constructor c and there exists a child tree $\mathcal{T}_i$ of $\mathcal{T}$ for $i \in \{1, \ldots, k\}$ with the constructor c at the inductive position, we proceed with $\mathcal{T}_i$. If there does not exist such a child tree, then narrowing fails.

- ▷ If $t|_p$ is a variable, we non-deterministically instantiate this variable with the constructor term at the inductive position of some child $\mathcal{T}_i$ of $\mathcal{T}$ for $i \in \{1, \ldots, k\}$ and again proceed with $\mathcal{T}_i$.
- ▷ If $t|_p$ is operation-rooted, we continue with recursively applying the computation of a needed narrowing step with $\sigma(t|_p)$, where σ is the instantiation of variables made in the previous case distinctions.

Formally, the needed narrowing strategy λ is a function that takes an operation-rooted term t and a definitional tree $\mathcal{T}$ of the operation at the root of t, and computes a set of triples $(p, l \to r, \sigma)$ such that p is a position of t, $l \to r$ is a (variant of) a rewrite rule and σ is a substitution. For each of these computed triples, the narrowing step $t \leadsto_{p,l\to r,\sigma} \sigma(t[r]_p)$ is performed. In the following, $pattern(\mathcal{T})$ denotes the pattern argument of $\mathcal{T}$.

Definition 3.6 (Needed Narrowing [AEH00])**.** *The function λ takes an operation-rooted term t and a pdt $\mathcal{T}$ such that $pattern(\mathcal{T})$ and t unify. The function λ is defined as the smallest set satisfying the following condition:*

$$\lambda(t, \mathcal{T}) \supseteq \begin{cases} \{(\varepsilon, l \to r, mgu(t, l))\} & \text{if } \mathcal{T} = rule(\pi, l \to r) \\ \lambda(t, \mathcal{T}_i) & \text{if } \mathcal{T} = branch(\pi, p, \overline{\mathcal{T}_k}), \text{ and } t \text{ and } pattern(\mathcal{T}_i) \\ & \text{unify for some } i \in \{1, \ldots, k\} \\ \{(p \cdot p', l \to r, \sigma \circ \tau)\} & \text{if } \mathcal{T} = branch(\pi, p, \overline{\mathcal{T}_k}), t \text{ and } pattern(\mathcal{T}_i) \text{ do not} \\ & \text{unify for any } i \in \{1, \ldots, k\}, \tau = mgu(t, \pi), \mathcal{T}' \text{ is} \\ & \text{a definitional tree of the function at the root of } t|_p, \\ & \text{and } (p', l \to r, \sigma) \in \lambda(\tau(t|_p), \mathcal{T}') \end{cases}$$

Note that the strategy of needed narrowing is not a narrowing strategy according to Definition 3.2, since it does not always compute mgus but also more special unifiers. However, this behavior still corresponds to a generalized form of narrowing that does not require the computed unifiers to be most general.

The strategy of needed narrowing fails to compute a triple for an *exempt* node, which implies that the strategy λ is not normalizing. Consider, for instance, the following operation

```
nil [] = True
```

and the term `nil [not True]`. In this case, a normalizing strategy could perform a step to compute the normal form `nil [False]`, while needed narrowing immediately fails. This is, however, no real restriction, since we are interested in the computation of *values* instead of normal forms for the implementation of functional logic languages.

Therefore, the theoretical results of needed narrowing are stated with respect to constructor terms as results. For this purpose, we assume the existence of a *strict equality* on terms, denoted by "`=:=`", such that the equation t_1 `=:=` t_2 is satisfied if and only if t_1 and t_2 are reducible to the same ground constructor term. Note that, in

contrast to the mathematical notation of equivalence, this operation is not reflexive, which is also intended for functional logic languages, since for some terms such as infinite data structures a normal form or value might not exist. The strict equality can be defined as a binary operation using the following set of rewrite rules, where the constant `Success` denotes a satisfied constraint and thus a solved equation.

$$c \texttt{ =:= } c \texttt{ = Success} \qquad \forall c \in \mathcal{C}^{(0)}$$

$$c(\overline{x_n}) \texttt{ =:= } c(\overline{y_n}) \texttt{ = } x_1 \texttt{ =:= } y_1 \texttt{ \& ... \& } x_n \texttt{ =:= } y_n \qquad \forall c \in \mathcal{C}^{(n)}, n > 0$$

$$\texttt{Success \& Success = Success}$$

The main properties of needed narrowing can then be formulated based on equations of the form $t_1 \texttt{ =:= } t_2$. A (correct) solution for an equation $t_1 \texttt{ =:= } t_2$ is a constructor substitution σ such that $\sigma(t_1) \texttt{ =:= } \sigma(t_2) \rightarrow^* \texttt{Success}$.

Theorem 3.7 (Properties of Needed Narrowing [AEH00])**.** *Let $\mathcal{R}$ be an inductively sequential TRS and e an equation.*

Soundness: If $e \rightsquigarrow^*_\sigma$ `Success` *is a needed narrowing derivation, then σ is a solution for e.*

Completeness: For each solution σ of e, there exists a needed narrowing derivation $e \rightsquigarrow^*_{\sigma'}$ `Success` *with $\sigma'(x) \leqslant \sigma(x)$ for all $x \in \mathcal{V}ar(e)$.*

Minimality: If $e \rightsquigarrow^*_\sigma$ `Success` *and $e \rightsquigarrow^*_{\sigma'}$* `Success` *are two distinct needed narrowing derivations, then σ and σ' are independent on $\mathcal{V}ar(e)$, i. e., there is some $x \in \mathcal{V}ar(e)$ such that $\sigma(x)$ and $\sigma'(x)$ are not unifiable.*

Furthermore, for successful derivations, the needed narrowing strategy only performs steps that are necessary to obtain the result, so that it computes the *shortest* of all possible narrowing derivations if derivations on common subterms are shared [AEH00].

3.4.2 Narrowing and Functional Logic Programs

The mechanism of narrowing in general and the needed narrowing strategy in particular seem well-suited for the definition of the semantics of integrated functional logic languages, due to the combination of reduction and variable instantiation. In fact, needed narrowing can be considered to form the computation model of functional logic languages such as Curry or $\mathcal{TOY}$. However, its application is generally restricted to left-linear, constructor-based, inductively sequential TRSs, so that different extensions have been proposed (see [Ant05] for a more detailed description).

Overlapping Left-Hand Sides

Since inductively sequential TRSs are also orthogonal [Ant92], this class of TRSs disallows the usage of even trivially overlapping left-hand sides. For instance, consider the following term rewriting system implementing the parallel-or operation.

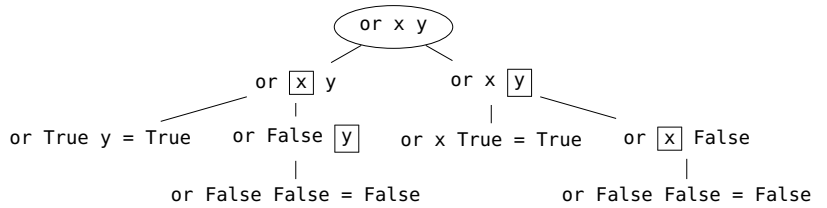


Figure 3.2. Generalized Definitional Tree for the Operation `or`

```
or True  y     = True
or x     True  = True
or False False = False
```

In this TRS, a term like "`or` t t'" can be reduced to `True` whenever one of its arguments can be reduced to `True`. But since the left-hand sides are (trivially) overlapping, there is no inductive position, so that it is unclear which argument should be evaluated first. Thus, the operation `or` has no definitional tree, and consequently the strategy of needed narrowing cannot be applied.

To solve this problem, needed narrowing has been extended to weakly needed narrowing [AEH97], which is applicable to constructor-based, weakly orthogonal TRSs. For this strategy, the notion of *generalized definitional trees* is used. This generalization includes an extension of definitional trees by *or*-nodes, effectively representing a non-deterministic choice between two definitional trees. For instance, the generalized definitional tree for the operation `or` defined above is depicted in Figure 3.2, where the additional *or*-node is surrounded by an ellipse.

Although weakly needed narrowing does not inherit the optimality result of needed narrowing in general, it is correct and sound for the class of constructor-based, weakly orthogonal TRSs, and furthermore still optimal for the subclass of inductively sequential TRSs.

Non-Confluence

Until now, we have only considered *confluent* term rewriting systems, i. e., TRSs such that for every term there exists at most one normal form. Although this restriction is reasonable in the context of functional languages, functional *logic* languages are more powerful in that they contain a built-in search mechanism. Therefore, it is necessary to also consider non-confluent TRSs containing *non-deterministic* operations. Of those operations, the non-deterministic *choice* operation "`?`" can be seen as the prototype representative:

```
x ? y = x
x ? y = y
```

Since generalized definitional trees already cover the notion of an *or*-node for the non-deterministic choice between two subtrees, there trivially exists a generalized definitional tree for the operation "`?`".

It sounds promising to also apply the strategy of weakly needed narrowing for the evaluation of terms such as `0 ? 1`, and it has been shown that this is indeed reasonable [GHL+99]. Furthermore, Antoy [Ant97] shows that it is possible to preserve desirable properties of needed narrowing by considering *overlapping inductively sequential* TRSs, i. e., TRSs consisting of inductively sequential rules with multiple right-hand sides that possibly contain extra variables. However, the combination of non-deterministic operations and demand-driven evaluation again results in the semantical ambiguity of call-time choice versus run-time choice. Since in term rewriting and narrowing the arguments of function calls are passed using substitutions, this effectively leads to a run-time choice semantics, whereas contemporary functional logic languages such as Curry or $\mathcal{TOY}$ usually adopt a call-time choice semantics.

More recent work [AP12; AJ13] presents an effective compiler based on *limited overlapping inductively sequential* graph rewriting systems, i. e., rewriting systems that contain only one single operation with overlapping rules, namely the choice operation "`?`" defined above. To comply to the call-time choice semantics, the authors implement the mechanism of sharing of subterms by graph rewriting [EJ97] instead of term rewriting.

3.5 Summary

This chapter presented a short introduction to the notions of terms, substitutions, and in particular term rewriting and narrowing. We discussed in more detail the strategy of needed narrowing, as well as the applicability of narrowing to cover features of contemporary functional logic languages such as non-determinism. Since the mechanism of narrowing for TRSs leads to a run-time choice semantics, it is not applicable for the definition of the operational semantics of Curry, and we will therefore provide an appropriate operational semantics in Chapter 5. However, needed narrowing has been the operational model of an earlier stage of Curry, and serves as the evaluation strategy of the partial evaluator proposed by Alpuente, Lucas, Hanus, and Vidal [ALH+05], which is described in Chapter 6. Both the operational semantics of Chapter 5 and the ideas presented in Chapter 6 will then be combined to develop a partial evaluation scheme for Curry in Chapter 7.

Part II

Normalization and Operational Semantics

Chapter 4

Normalization of Curry Programs

Simplicity is the ultimate sophistication.

attributed to Leonardo da Vinci

In Chapter 2 we presented the fundamental concepts of the functional logic language Curry, together with its concrete syntax. We introduced several different syntactic constructs, such as pattern matching, the `do`-notation for IO actions, or functional patterns, which all increase the expressiveness of the language. Of these constructs, a large number is defined by means of transformations into simpler ones, so that the set of constructs that are necessary to represent a program can be considerably decreased by an iterative application of transformation rules. Most of these transformations are also described in the Curry Report [Han12], but usually in isolation and not for all language constructs. This chapter is therefore concerned with the formalization of the entire normalization process applied to Curry programs.

4.1 Introduction

The normalization of Curry programs is incorporated into the compilation chain of current Curry systems, which perform the general compilation phases depicted in Figure 4.1. Compilation starts with the phase of program recognition (*lexing* and *parsing*), which is necessary to retrieve the abstract syntax tree of a given program, and continues with a validation phase (*context-sensitive syntax checking* and *type checking*) to ensure the validity of the program, for instance definedness of used identifiers and well-typedness of the program. The subsequent normalization phase then transforms the validated program into its *normalized form*, i. e., a simplified representation suitable for further translation into different target languages, by the following steps.

Qualification All identifiers referring to globally visible entities are fully qualified, i. e., they are extended with the identifier of their defining module. For instance, the function `map` will be qualified to `Prelude.map`.

Desugaring Complex language constructs which can be expressed using simpler ones are subsequently transformed, thus reducing the overall number of distinct constructs. Since special constructs are often referred to as “syntactic sugar”, this step is called *desugaring* accordingly.

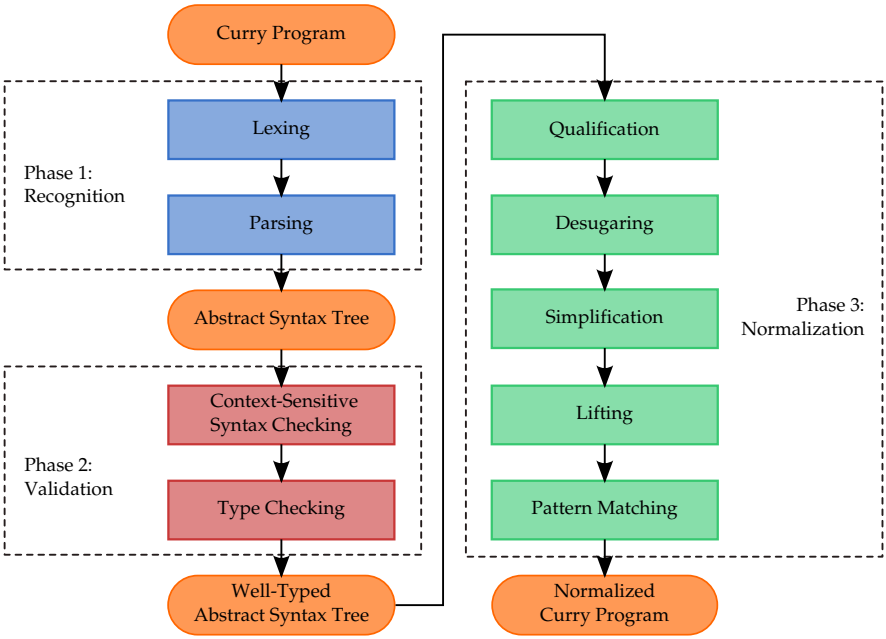


Figure 4.1. Typical Phases in the Compilation of Curry Programs

Simplification The resulting program is then further simplified, for instance by the removal of unused local declarations, the replacement of variables referring to constants by their bindings, and further simplifications of the syntactic structure.

Lifting of Local Declarations The simplified program may still contain local function declarations. In this step, these declarations are lifted to the top-level of the program, such that afterwards all functions are globally defined.

Compilation of Pattern Matching Finally, the pattern matching mechanism is explicitly encoded in the program by means of `case` expressions. Furthermore, different equations defining the same function are combined, such that afterwards every function will be defined by exactly one equation.

The remainder of this chapter is concerned with the formalization of these individual steps, whereas we omit the description of the qualification step due to its simplicity. In the following, we assume that identifiers have already been disambiguated by a renaming in the process of syntax checking, so that the potential shadowing of global identifiers and argument variables by local definitions has been eliminated. In consequence, it will be safe to move declarations without the risk of name capture problems.

4.2 Desugaring

The task of desugaring implements the reduction of the syntactic complexity of Curry programs, and we formalize the transformations applied to the different language constructs in the following. We accompany the description of the individual transformations with the corresponding excerpts of the context-free grammar of Curry, which is presented in Appendix A. We will not consider the entire syntax of Curry, but concentrate on the type and function declarations. Thus, additional constructs for the support of multiple modules (export and import specifications), guidance of the parsing process (fixity declarations), implementation of primitive operations (external data types and functions) and the optional specification of types (in typed expressions and function signatures) will not be covered, since they are irrelevant for the process of normalization.

4.2.1 Type Declarations

$$
\begin{array}{rcll}
\mathit{TypeSynDecl} & ::= & \texttt{type}\ \mathit{SimpleType}\ \texttt{=}\ \mathit{TypeExpr} & \\
\mathit{SimpleType} & ::= & \mathit{TypeConstrID}\ \mathit{TypeVarID}_1\ \ldots\ \mathit{TypeVarID}_n & (n \geqslant 0) \\
\mathit{DataDecl} & ::= & \texttt{data}\ \mathit{SimpleType}\ \texttt{=}\ \mathit{ConstrDecl}_1\ \texttt{|}\ \ldots\ \texttt{|}\ \mathit{ConstrDecl}_n & (n \geqslant 1) \\
\mathit{ConstrDecl} & ::= & \mathit{DataConstr}\ \mathit{SimpleTypeExpr}_1\ \ldots\ \mathit{SimpleTypeExpr}_n & (n \geqslant 0) \\
 & | & \mathit{TypeConsExpr}\ \mathit{ConOp}\ \mathit{TypeConsExpr} & (\textit{infix data constructor}) \\
 & | & \mathit{DataConstr}\ \texttt{\{}\ \mathit{FieldDecl}_1\ \texttt{,}\ \ldots\ \texttt{,}\ \mathit{FieldDecl}_n\ \texttt{\}} & (n \geqslant 0) \\
\mathit{FieldDecl} & ::= & \mathit{Label}_1\ \texttt{,}\ \ldots\ \texttt{,}\ \mathit{Label}_n\ \texttt{::}\ \mathit{TypeExpr} & (n \geqslant 1) \\
\mathit{NewtypeDecl} & ::= & \texttt{newtype}\ \mathit{SimpleType}\ \texttt{=}\ \mathit{NewConstrDecl} & \\
\mathit{NewConstrDecl} & ::= & \mathit{DataConstr}\ \mathit{SimpleTypeExpr} & \\
 & | & \mathit{DataConstr}\ \texttt{\{}\ \mathit{Label}\ \texttt{::}\ \mathit{TypeExpr}\ \texttt{\}} &
\end{array}
$$

The language Curry provides the following three different kinds of type declarations.

Type synonyms define abbreviations for potentially complex type expressions, and may contain place holders in the form of type variables. Since they are *synonyms*, they can always be replaced by their right-hand side (the type expression) using the appropriate mapping for the type variables. To prohibit the construction of infinite type expressions, type synonyms must not be (mutually) recursive.

Data type declarations introduce new algebraic data types, of which values can be constructed and deconstructed using the specified constructors.

Renaming types are declared similarly to data types, but with the keyword `newtype` instead of `data` and with the restriction that they must contain exactly one unary constructor. Conceptually, they provide a combination of data types and type synonyms. On the one hand, they introduce a new type name just like data types so that they can be distinguished from other types to increase type safety. On the other hand, they are converted into type synonyms in the transformation process

by removal of the constructor in order to avoid the overhead originating from (de)construction of values. Due to this transformation, renaming types (potentially combined with type synonyms) also must not be (mutually) recursive.[1]

In addition, data types and renaming types can be extended by record labels, which provide a convenient notation for the selection and update of the components of values. For instance, the record declaration

```
data Person = Lecturer { name, office :: String }
            | Student  { name :: String, studnr :: Int }
```

defines three record labels `name`, `office`, and `studnr`, which can be used to access the respective field of a corresponding value. Record declarations are transformed into ordinary declarations in three steps. Firstly, the sequence of fields in a single field declaration is expanded by repetition of the associated type expression. Secondly, this expanded declaration is converted to an ordinary declaration by a simple omission of the field labels. Thirdly, the labels are converted into *selection functions*, i. e., functions for the selection of a single component of a record value. Thus, the above example will be transformed into

```
data Person = Lecturer String String | Student String Int
name   (Lecturer x _) = x
name   (Student  x _) = x
office (Lecturer _ x) = x
studnr (Student  _ x) = x
```

In consequence, the desugaring of a type declaration may return both a changed type declaration as well as multiple function declarations. While type synonyms remain unchanged, the constructors of data types and renaming types are desugared using the operation dsCons.

$$
\begin{aligned}
\mathsf{dsType} &: \mathit{TypeDecl} \to (\mathit{TypeDecl}, \mathit{FunctionDecl}^*) \\
\mathsf{dsType}(\texttt{type}\ T\ \overline{a_k}\ \texttt{=}\ \tau) &= (\texttt{type}\ T\ \overline{a_k}\ \texttt{=}\ \tau, \varepsilon) \\
\mathsf{dsType}(\texttt{data}\ T\ \overline{a_k}\ \texttt{=}\ \overline{c_n}) &= (\texttt{data}\ T\ \overline{a_k}\ \texttt{=}\ \overline{c'_n},\ \mathsf{join}(\overline{sel_{l_1}}, \ldots, \overline{sel_{l_n}})) \\
&\quad \text{where } (c'_i, \overline{sel_{l_i}}) = \mathsf{dsCons}(c_i) \text{ for } i \in \{1, \ldots, n\} \\
\mathsf{dsType}(\texttt{newtype}\ T\ \overline{a_k}\ \texttt{=}\ c) &= (\texttt{type}\ T\ \overline{a_k}\ \texttt{=}\ \tau, (C\ \texttt{x}\ \texttt{=}\ \texttt{x}) \cdot sel)) \\
&\quad \text{where } (C\ \tau, sel) = \mathsf{dsCons}(c) \\
\mathsf{dsCons}(C\ \overline{\tau_n}) &= (C\ \overline{\tau_n}, \varepsilon) \\
\mathsf{dsCons}(C\ \{\ \overline{fd_m}\ \}) &= (C\ \overline{\tau_n}, \overline{\mathsf{selFun}(C, \overline{l_n}, l_n)}) \\
&\quad \text{where } \overline{l_n\ ::\ \tau_n} = \mathsf{expand}(\overline{fd_m}) \\
\mathsf{selFun}(C, \overline{l_n}, l_i) &= l_i\ (C\ x_1\ \ldots\ x_{i-1}\ y\ x_{i+1}\ \ldots\ x_n)\ \texttt{=}\ y
\end{aligned}
$$

[1] Note that this restriction differs from the functional language Haskell [Mar10], which does allow mutually recursive renaming types. However, the restriction to non-recursive renaming types in Curry ensures that the transformed program is still a valid Curry program.

$$\mathsf{expand}(\varepsilon) = \varepsilon$$
$$\mathsf{expand}((\overline{l_k} :: \tau) \cdot \overline{fd_m}) = \overline{l_k :: \tau} \cdot \mathsf{expand}(\overline{fd_m})$$

In case of record declarations, the operation dsCons yields for each constructor declaration c_i a corresponding declaration c_i' without field labels, and a sequence $\overline{sel_{l_i}}$ of selection function equations for the labels defined in c_i. Since a specific label may occur in more than one constructor declaration (cf. the field `name` from the example above), we assume an operation join that joins the different equations for the selection functions. Renaming types are furthermore converted into ordinary type synonyms, where the constructor is converted to an identity function since it may still occur as a higher-order argument or result.

4.2.2 Patterns

The desugaring of a pattern basically is a simple traversal of the pattern's structure, where special patterns are subsequently transformed into simpler ones. However, the special cases of as-patterns and irrefutable patterns may introduce new local pattern declarations, so that the desugaring of a single pattern in general results in a simplified pattern and a list of new simple pattern declarations. After this desugaring, patterns are only composed of variables, literals, constructors and as-patterns, whereas the latter are applied to literal or constructor patterns.

Note that the transformation of non-linear patterns and functional patterns will be later described in Section 4.2.4, since they may only occur in the left-hand side of function declarations.

Basic Patterns

Pattern	::=	*ConsPattern* [*QConOp Pattern*]	(*infix constructor pattern*)
ConsPattern	::=	*GDataConstr SimplePat*$_1$ ... *SimplePat*$_n$	(*constructor pattern*, $n \geqslant 1$)
	|	- *Int*	(*negative integer pattern*)
	|	-. *Float*	(*negative float pattern*)
	|	*SimplePat*	
SimplePat	::=	*Variable*	(*variable*)
	|	_	(*wildcard*)
	|	*GDataConstr*	(*constructor*)
	|	*Literal*	(*literal*)
	|	(*Pattern*)	(*parenthesized pattern*)
	|	(*Pattern*$_1$, ... , *Pattern*$_n$)	(*tuple pattern*, $n \geqslant 2$)
	|	[*Pattern*$_1$, ... , *Pattern*$_n$]	(*list pattern*, $n \geqslant 1$)
	|	...	

The desugaring of basic patterns such as variable or literal patterns is straightforward. Variable patterns remain unchanged, as well as literal patterns for numbers or

characters, and no additional pattern declarations are extracted. A string literal pattern, however, is replaced by a list pattern with subjacent character patterns, and then further desugared.

$$\begin{aligned}
&\mathsf{dsPat} : \mathit{Pattern} \to (\mathit{Pattern}, \mathit{PatternDecl}^*)\\
&\mathsf{dsPat}(v) = (v, \varepsilon) \quad \text{where } v \text{ is a variable or wildcard pattern}\\
&\mathsf{dsPat}(l) = \begin{cases} (l, \varepsilon) & \text{if } l \text{ is an integer, float, or character literal}\\ \mathsf{dsPat}(\texttt{[}c_1\texttt{, ..., }c_n\texttt{]}) & \text{if } l \text{ is a string literal representing the character sequence } \overline{c_n} \end{cases}
\end{aligned}$$

Negative patterns are only allowed around integer or float literals to denote patterns for negative numbers like "`-1`" or "`-.3.14`". These patterns are transformed by a simple negation of the literal's value:

$$\begin{aligned}
\mathsf{dsPat}(\texttt{-}\ i) &= \mathsf{dsPat}(-i) \quad \text{where } i \text{ is an integer literal}\\
\mathsf{dsPat}(\texttt{-.}\ f) &= \mathsf{dsPat}(-f) \quad \text{where } f \text{ is a float literal}
\end{aligned}$$

The constructors of renaming types are removed in patterns, and infix constructor applications are converted to prefix applications:

$$\begin{aligned}
\mathsf{dsPat}(C\ \overline{p_n}) &= \begin{cases} \mathsf{dsPat}(p_1) & \text{if } C \text{ is a newtype constructor}\\ (C\ \overline{p'_n}, \bigcup \overline{ds_n}) & \text{otherwise, where } \overline{(p'_n, ds_n)} = \overline{\mathsf{dsPat}(p_n)} \end{cases}\\
\mathsf{dsPat}(p_1 \oplus p_2) &= \mathsf{dsPat}(\texttt{(}\oplus\texttt{)}\ p_1\ p_2) \quad \text{if } \oplus \text{ is an infix constructor}\\
\mathsf{dsPat}(p_1\ \texttt{`}C\texttt{`}\ p_2) &= \mathsf{dsPat}(C\ p_1\ p_2)
\end{aligned}$$

Parentheses in patterns are only necessary for syntactic disambiguation and thus removed in the transformation. Tuple patterns are recursively desugared and converted into constructor patterns, while list patterns are first converted into a series of applications of the list constructors "`:`" and "`[]`". For instance, the pattern `[1,2,3]` is transformed to `1 : 2 : 3 : []`. Since the desugaring of lists can later be reused for the desugaring of list expressions, we generalize it to desugar a list of arbitrary objects.

$$\begin{aligned}
\mathsf{dsPat}(\texttt{(}p\texttt{)}) &= \mathsf{dsPat}(p)\\
\mathsf{dsPat}(\texttt{(}p_1\texttt{, ..., }p_n\texttt{)}) &= \mathsf{dsPat}(\texttt{(,...,)}\ p_1\ \ldots\ p_n)\\
\mathsf{dsPat}(\texttt{[}p_1\texttt{, ..., }p_n\texttt{]}) &= \mathsf{dsPat}(\mathsf{dsList}(\texttt{[}p_1\texttt{, ..., }p_n\texttt{]}))\\
\mathsf{dsList}(\texttt{[}o_1\texttt{, ..., }o_n\texttt{]}) &= \begin{cases} \texttt{[]} & \text{if } n = 0\\ o_1\ \texttt{:}\ \mathsf{dsList}(\texttt{[}o_2\texttt{, ..., }o_n\texttt{]}) & \text{if } n > 0 \end{cases}
\end{aligned}$$

As-Patterns

SimplePat ::= ... | *Variable* `@` *SimplePat* (*as-pattern*)

An as-pattern of the form `v@p`, where v is a variable and p a pattern, introduces a

binding of the variable v to the value that is matched by the pattern p. For example, the function definition

```
mirror l@(Leaf   _) = l
mirror (Node l n r) = Node (mirror r) n (mirror l)
```

is equivalent to the definition

```
mirror t = mirror' t
  where
  mirror' (Leaf     _) = t
  mirror' (Node l n r) = Node (mirror r) n (mirror l)
```

Since the pattern matching will later be transformed into `case` expressions, we do not generate new local functions for as-patterns, but retain them until the compilation of pattern matching. However, the application of an as-pattern to a variable pattern or a nested as-pattern is redundant, and we remove redundant patterns by the following transformation:

$$\mathsf{dsPat}(x@p) = \mathsf{dsAs}(x, \mathsf{dsPat}(p))$$

$$\mathsf{dsAs}(x, (p, ds)) = \begin{cases} (p, (x = y) \cdot ds) & \text{if } p = y \text{ is a variable pattern} \\ (p, (x = y) \cdot ds) & \text{if } p = y@p' \text{ is an as-pattern} \\ (x@p, ds) & \text{otherwise} \end{cases}$$

In this transformation, nested as-patterns are transformed into new variable declarations. For instance, the following (circumstantial) definition

```
f x@(y@z) = (x, y, z)
```

will be transformed to the definition

```
f z = let x = y; y = z in (x, y, z)
```

Irrefutable Patterns

SimplePat ::= ... | ~ *SimplePat* (*irrefutable pattern*)

Irrefutable patterns allow the definition of patterns that inspect the structure of an argument while the pattern matching always succeeds. If an irrefutable pattern $\sim p$ is matched against a value v, then the variables in p are bound to the respective values in v if p matches against v, or to `failed` otherwise. This may become useful if an argument should be matched only in specific cases. For instance, in the following function definition

```
f b ~(Just y) = if b then "True" else show y
```

the second argument is matched against an irrefutable pattern, so that the expressions `f True Nothing` and `f False (Just True)` will both evaluate to the string `"True"`.

However, if the underlying pattern `Just y` is matched, the variable `y` is bound to `failed` if matching fails, and thus `f False Nothing` results in a failed evaluation. This is comparable to the lazy pattern matching semantics in `let` expressions (see also Section 4.3.1), so that this function is equivalent to the following definition:

```
f b x = let Just y = x
        in  if b then "True" else show y
```

Irrefutable patterns are converted into local pattern declarations as shown above, and the transformation is defined as:

$$\mathsf{dsPat}(\sim p) = \mathsf{dsLazy}(\mathsf{dsPat}(p))$$

$$\mathsf{dsLazy}((p, ds)) = \begin{cases} (v, (p = v) \cdot ds) & \text{if } p \text{ is a literal or constructor pattern,} \\ & \text{where } v \text{ is a fresh variable} \\ \mathsf{dsAs}(x, \mathsf{dsLazy}((p', ds))) & \text{if } p = x@p' \\ (p, ds) & \text{if } p \text{ is a variable pattern} \end{cases}$$

The first case establishes a local pattern binding for literal or constructor patterns, just like in the example presented above. In the second case, the variable of an irrefutable as-pattern is moved upwards, such that a pattern $\sim(x@p)$ is first transformed to $x@(\sim p)$ and then further desugared. Since pattern matching always succeeds for variables, an irrefutable variable pattern is redundant, so that a pattern like $\sim x$ where x is a variable is simplified to x in the third case.

Record Patterns

$$\begin{aligned} \mathit{SimplePat} &::= \ldots \mid \mathit{QDataConstr}\ \{\ \mathit{FieldPat}_1\ ,\ \ldots\ ,\ \mathit{FieldPat}_n\ \} \qquad (\textit{labeled pattern},\ n \geqslant 0) \\ \mathit{FieldPat} &::= \mathit{QLabel} = \mathit{Pattern} \end{aligned}$$

To facilitate the pattern matching of record components, Curry provides a special syntax for pattern matching using field labels instead of matching by position. For instance, if we consider the following record declaration

```
data Person = Lecturer { name :: String, office :: String }
            | Student  { name :: String, studnr :: Int    }
```

we can match a student with the student number `1234` by the following pattern:

```
is1234 (Student { studnr = 1234 }) = success
```

For multiple labels occurring in a record pattern, the components are matched in the order of their definition in the record declaration, not in the order of the occurrence of the field labels in the pattern, to achieve the same order as for constructor patterns.

Naturally, the field pattern sequence may only contain fields that are defined for the respective constructor, and each field may occur at most once. If some fields are omitted, then the corresponding components are implicitly matched against a wildcard pattern. Furthermore, the sequence can be empty, which abbreviates a

sequence of wildcard patterns of the length of the constructor's arity. Therefore, even for constructors without field declarations, this syntax can be used to abbreviate multiple wildcard patterns:

```
data Complex = Simple | Complex Int String Bool Float Char
isComplex (Complex {}) = success
```

Thus, the above pattern matchings will be transformed to the following patterns:

```
is1234    (Student    _ 1234) = success
isComplex (Complex _ _ _ _ _) = success
```

The transformation of record patterns converts a record pattern into the corresponding constructor pattern by extracting the respective field patterns from the given sequence, where a missing pattern defaults to a wildcard pattern:

$$\mathsf{dsPat}(C\ \{\ bs\ \}) = \mathsf{dsPat}(C\ \mathsf{pick}_1^C(bs, \_)\ \ldots\ \mathsf{pick}_k^C(bs, \_)) \quad \text{where } k \text{ is the arity of } C$$

In this definition, the auxiliary operation $\mathsf{pick}_i^C(bs, d)$ selects the binding for the i-th component of the constructor C (with has the field label f_i) from a sequence of bindings bs. If $f_i = v_i$ appears in bs, then $\mathsf{pick}_i^C(bs, d)$ is v_i, otherwise it defaults to d.

4.2.3 Expressions

Like the desugaring of a pattern, the desugaring of an expression is mainly a simple traversal of the expression's structure, where special constructs are subsequently transformed into simpler ones. Most of the desugaring transformations for expressions are specified by the Curry Report [Han12] to define the semantics of the respective constructs. However, in the report the transformations are presented independently of each other, so that the precise order of the transformations is ambiguous. We therefore provide the entire desugaring process to avoid any ambiguities. After desugaring, an expression is only composed of literals, variables, function and constructor names, applications of an expression to another expression, `case` expressions, `let` expressions, and expressions with an explicit type signature.

Basic Expressions

BasicExpr ::=	*Variable*	(*variable*)
|	_	(*anonymous free variable*)
|	*QFunction*	(*qualified function*)
|	*GDataConstr*	(*general constructor*)
|	*Literal*	(*literal*)
|	(*Expr*)	(*parenthesized expression*)
|	($Expr_1$, ... , $Expr_n$)	(*tuple*, $n \geqslant 2$)
|	[$Expr_1$, ... , $Expr_n$]	(*finite list*, $n \geqslant 1$)
|	...	

The transformation of basic expressions is rather trivial, but we provide them for completeness. Named variables are left unchanged, while anonymous free variables denote by an underscore "_" are translated into function calls to the function `unknown`, which is predefined in the module `Prelude` as

```
unknown :: a
unknown = let x free in x
```

Function and constructor symbols are left unchanged, whereas literals are desugared in the same way as literal patterns. While parentheses around expressions are removed, tuple expressions are converted into their prefix notation. Furthermore, finite list expressions are decomposed into applications of the list constructors "`(:)`" and "`[]`" by means of the auxiliary operation dsList previously introduced.

$$\begin{aligned}
\mathsf{dsExpr} &: \mathit{Expression} \to \mathit{Expression} \\
\mathsf{dsExpr}(v) &= v \quad \text{where } v \text{ is a variable} \\
\mathsf{dsExpr}(_) &= \texttt{Prelude.unknown} \\
\mathsf{dsExpr}(f) &= f \quad \text{where } f \text{ is a function symbol} \\
\mathsf{dsExpr}(C) &= C \quad \text{where } C \text{ is a constructor symbol} \\
\mathsf{dsExpr}(l) &= \begin{cases} l & \text{if } l \text{ is an integer, float, or character literal} \\ \mathsf{dsExpr}(\texttt{[}c_1\texttt{, ..., }c_n\texttt{]}) & \text{if } l \text{ is a string literal representing the character sequence } \overline{c_n} \end{cases} \\
\mathsf{dsExpr}(\texttt{(}e\texttt{)}) &= \mathsf{dsExpr}(e) \\
\mathsf{dsExpr}(\texttt{(}e_1\texttt{, ..., }e_n\texttt{)}) &= \mathsf{dsExpr}(\texttt{(,...,)}\ e_1\ \ldots\ e_n) \\
\mathsf{dsExpr}(\texttt{[}e_1\texttt{, ..., }e_n\texttt{]}) &= \mathsf{dsExpr}(\mathsf{dsList}(\texttt{[}e_1\texttt{, ..., }e_n\texttt{]}))
\end{aligned}$$

Typed Expressions, Applications and Sections

Expr	::=	*InfixExpr* `::` *TypeExpr*	(*expression with type signature*)
	|	*InfixExpr*	
InfixExpr	::=	*NoOpExpr QOp InfixExpr*	(*infix operator application*)
	|	`-` *InfixExpr*	(*unary int minus*)
	|	`-.` *InfixExpr*	(*unary float minus*)
	|	*NoOpExpr*	
NoOpExpr	::=	... | *FuncExpr*	
FuncExpr	::=	[*FuncExpr*] *BasicExpr*	(*application*)
BasicExpr	::=	...	
	|	`(` *InfixExpr QOp* `)`	(*left section*)
	|	`(` $QOp_{\langle \texttt{-}, \texttt{-.} \rangle}$ *InfixExpr* `)`	(*right section*)

To disambiguate the type of otherwise polymorphic expressions, Curry allows the explicit notation of the type of an expression by the special syntax $e :: \tau$, specifying

that the expression e is of type τ. Such type annotations remain unchanged during the desugaring of the underlying expression:

$$\mathsf{dsExpr}(e :: \tau) = \mathsf{dsExpr}(e) :: \tau$$

For (function) application, Curry provides both the prefix application notation $e_1\ e_2$ as well as an infix application, where the function to be applied can be an operator ($e_1 \oplus e_2$) or a function name surrounded by backquotes (e_1 `` `f` `` e_2). In addition, infix operators can be partially applied to either the first or second argument, which is called a *section* and provides a convenient notation for their partial application to a single argument.

These special notations are transformed into prefix applications by the following transformation. In addition, the application of a renaming type constructor is discarded, and applications of the unary negation operations remain unchanged.

$$\begin{aligned}
\mathsf{dsExpr}(e_1\ e_2) &= \begin{cases} \mathsf{dsExpr}(e_2) & \text{if } e_1 \text{ is a newtype constructor} \\ \mathsf{dsExpr}(e_1)\ \mathsf{dsExpr}(e_2) & \text{otherwise} \end{cases} \\
\mathsf{dsExpr}(e_1 \oplus e_2) &= \mathsf{dsExpr}(\texttt{(}\oplus\texttt{)}\ e_1\ e_2) \\
\mathsf{dsExpr}(e_1\ \texttt{`}f\texttt{`}\ e_2) &= \mathsf{dsExpr}(f\ e_1\ e_2) \\
\mathsf{dsExpr}(e \oplus) &= \mathsf{dsExpr}(\texttt{(}\oplus\texttt{)}\ e) \\
\mathsf{dsExpr}(e\ \texttt{`}f\texttt{`}) &= \mathsf{dsExpr}(f\ e) \\
\mathsf{dsExpr}(\oplus e) &= \begin{cases} \oplus\ \mathsf{dsExpr}(e) & \text{if } \oplus \in \{\texttt{-}, \texttt{-.}\} \\ \mathsf{dsExpr}(\texttt{flip (}\oplus\texttt{)}\ e) & \text{otherwise} \end{cases} \\
\mathsf{dsExpr}(\texttt{`}f\texttt{`}\ e) &= \mathsf{dsExpr}(\texttt{flip}\ f\ e)
\end{aligned}$$

Note that left sections are translated into prefix applications and right sections into applications of the auxiliary function `flip`, which is defined as

```
flip :: (a -> b -> c) -> b -> a -> c
flip f x y = f y x
```

This is in contrast to Haskell, where sections are transformed into local anonymous functions using lambda expressions [Mar10]. However, this deviation is necessary to obtain the semantics of sections as a notation for partial application. For instance, consider the definition

```
add x y = x + y
```

and the expression `map (add (1 ? 2)) [3, 4]` containing a partial application. During evaluation of this expression, the non-determinism will be shared, so that the expression yields the results `[4, 5]` or `[5, 6]`. By the presented transformation, the section `map ((1 ? 2) +) [3, 4]` is transformed to `map ((+) (1 ? 2)) [3, 4]`, which evaluates to the same results as if a partial application was used. By a translation into lambda expression as done in Haskell, it would be transformed to `map (\x -> (1 ? 2) + x) [3, 4]`. Since lambda expressions are later transformed into anonymous

functions (see below), the non-determinism is duplicated and the expression yields the additional results `[4, 6]` and `[5, 5]`.

Case Expressions

$$
\begin{array}{rcll}
\mathit{NoOpExpr} & ::= & \ldots & \\
& | & \texttt{case}\ \mathit{Expr}\ \texttt{of}\ \{\ \mathit{Alt}_1\ ;\ \ldots\ ;\ \mathit{Alt}_n\ \} & (\textit{case expression},\ n \geqslant 1) \\
& | & \texttt{fcase}\ \mathit{Expr}\ \texttt{of}\ \{\ \mathit{Alt}_1\ ;\ \ldots\ ;\ \mathit{Alt}_n\ \} & (\textit{fcase expression},\ n \geqslant 1) \\
\mathit{Alt} & ::= & \mathit{Pattern}\ \texttt{->}\ \mathit{Expr}\ [\texttt{where}\ \mathit{LocalDecls}] & \\
& | & \mathit{Pattern}\ \mathit{GdAlts}\ [\texttt{where}\ \mathit{LocalDecls}] & \\
\mathit{GdAlts} & ::= & |\ \mathit{InfixExpr}\ \texttt{->}\ \mathit{Expr}\ [\mathit{GdAlts}] &
\end{array}
$$

The syntax of `case` expressions provides a convenient notation for pattern matching inside an expression, comparable to the pattern matching in the left-hand side of a function declaration, but limited to a single expression to be matched. There exist two different kinds of `case` expressions, namely *rigid* and *flexible* `case` expressions, which are denoted by the keywords "`case`" and "`fcase`", respectively (the leading "f" stands for "flexible").

The two kinds are distinguished to implement two different strategies of pattern matching. For *rigid* `case` expressions, the alternatives are tried sequentially, so that the first matching pattern determines the value of the expression by its right-hand side. If none of the patterns match, then the evaluation fails. *Flexible* `case` expressions, in contrast, non-deterministically try *all* matching alternatives, which corresponds to the pattern matching mechanism of functions. A second difference arises when the expression to be matched evaluates to a logic variable. While the evaluation of a *rigid* `case` expression suspends, a *flexible* `case` expression is evaluated by a non-deterministic instantiation of the variable to a pattern and the subsequent evaluation of the corresponding right-hand side.

In addition to a simple expression in the right-hand side of an alternative, a `case` expression may also contain guards as well as local declarations. We will therefore use the notation $p \to r$ to express an alternative with pattern p and right-hand side r, where r may be guarded and contain local declarations. The desugaring of `case` expressions starts with the desugaring of the scrutinized expression and the patterns, and possible new declarations originating from a pattern are added to the declarations of the corresponding right-hand side.

$$
\begin{aligned}
\mathsf{dsExpr}(\texttt{fcase}\ e\ \texttt{of}\ \overline{alt_n}) &= \mathsf{dsFCase}(\mathsf{dsExpr}(e), \overline{\mathsf{dsAlt}(alt_n)}) \\
\mathsf{dsExpr}(\texttt{case}\ e\ \texttt{of}\ \overline{alt_n}) &= \mathsf{dsRCase}(\mathsf{dsExpr}(e), \overline{\mathsf{dsAlt}(alt_n)}) \\
\mathsf{dsAlt}(p \to r) &= p' \to \mathsf{addDecls}(ds, r) \quad \text{where } (p', ds) = \mathsf{dsPat}(p)
\end{aligned}
$$

The addition of local declarations to the right-hand side is defined as

$$
\begin{aligned}
\mathsf{addDecls}(ds, \texttt{->}\ e\ \texttt{where}\ ds') &= \texttt{->}\ e\ \texttt{where}\ ds \cdot ds' \\
\mathsf{addDecls}(ds, \overline{\texttt{|}\ c_n\ \texttt{->}\ e_n}\ \texttt{where}\ ds') &= \overline{\texttt{|}\ c_n\ \texttt{->}\ e_n}\ \texttt{where}\ ds \cdot ds'
\end{aligned}
$$

where we assume that a right-hand side without local declarations contains a `where` block with an empty sequence of declarations.

For *flexible* `case` expressions, the process continues with the desugaring of the alternatives' right-hand sides. Since it is generally possible that all conditions in a guarded right-hand side fail, we provide the expression `failed` as the default alternative. Furthermore, in the rare case that the expression has no alternatives at all, it is directly transformed to `failed`.[2]

$$\mathsf{dsFCase}(e, \overline{p_n \to r_n}) = \begin{cases} \texttt{failed} & \text{if } n = 0 \\ \texttt{fcase } e \texttt{ of } \overline{p_n \texttt{ -> } \mathsf{dsRhs}(\texttt{failed}, r_n)} & \text{if } n > 0 \end{cases}$$

In the desugaring of the right-hand side, any local declarations are moved into a new `let` expression such that the `where` block can be removed (a `let` expression without declarations will later be simplified). Afterwards, conditional expressions are converted into ordinary expressions, before the entire expression itself gets desugared.

$$\begin{aligned} \mathsf{dsRhs}(e', r) &= \mathsf{dsExpr}(\mathsf{expandRhs}(e', r)) \\ \mathsf{expandRhs}(e', e \texttt{ where } ds) &= \texttt{let } ds \texttt{ in } e \\ \mathsf{expandRhs}(e', \overline{\texttt{| } c_n \texttt{ = } e_n} \texttt{ where } ds) &= \texttt{let } ds \texttt{ in } \mathsf{dsConds}(e', \overline{\texttt{| } c_n \texttt{ = } e_n}) \end{aligned}$$

The desugaring of conditional expressions depends on the number of alternatives and the type of the conditions. If there only exists a single alternative of type `Success`, the condition and the expression are combined using the primitive conditional operation "`&>`". Multiple alternatives, which then must be of type `Bool`, are combined by a sequence of nested `if-then-else` expressions.

$$\mathsf{dsConds}(e', (\texttt{| } c \texttt{ = } e) \cdot (\overline{\texttt{| } c_m \texttt{ = } e_m})) = \begin{cases} c \texttt{ \&> } e & \text{if } m = 0 \text{ and } c \text{ is of type } \texttt{Success} \\ \texttt{if } c \texttt{ then } e \texttt{ else } e' & \text{if } m = 0 \text{ and } c \text{ is of type } \texttt{Bool} \\ \texttt{if } c \texttt{ then } e \texttt{ else } \mathsf{dsConds}(e', \overline{\texttt{| } c_m \texttt{ = } e_m}) & \text{if } m > 0 \text{ and } c, \overline{c_m} \text{ are of type } \texttt{Bool} \end{cases}$$

In contrast to flexible `case` expressions, the transformation of *rigid* `case` expressions is slightly more complicated, due to their sequential pattern matching semantics. For instance, consider the following expression:

```
f x y = case x of
  True | y -> 0
  False    -> 1
  _        -> 2
```

In this expression, the guard of the first alternative may fail if `y` evaluates to `False`, and in this case the expression should evaluate to `2`. To express this fall-through semantics, the first alternative has to be extended with an additional pattern matching

[2] This case may occur after the desugaring of other expressions.

against the remaining alternatives. Thus, the example should be transformed to an expression equivalent to:

```
f x y = case x of
  True | y         -> 0
       | otherwise -> case x of
                        _ -> 2
  False -> 1
  _     -> 2
```

Since the scrutinized expression may be duplicated, it is bound to a fresh variable to avoid repeated evaluations or duplication of non-determinism in the following transformation.

$$\mathsf{dsRCase}(e, \overline{p_n \to r_n}) = \begin{cases} \texttt{failed} & \text{if } n = 0 \\ \texttt{let } v \texttt{ = } e \texttt{ in case } v \texttt{ of } \overline{p_n \texttt{ -> } \mathsf{dsRhs}(e'_n, r_n))} & \text{if } n > 0 \end{cases}$$

where v is a fresh variable and
$e'_i = \texttt{case } v \texttt{ of } (p_j \to r_j \in \overline{p_n \to r_n} \mid j > i \wedge \mathsf{compat}(p_i, p_j))$

The default alternatives e'_i for $i \in \{1, \ldots, n\}$ proceed with a matching of the remaining case alternatives, whereas the considered alternatives are restricted to *compatible patterns* to avoid the construction of unreachable code. Consider, for instance, the example above where the added `case` expression

```
case x of _ -> 2
```

did not consider the alternative `False -> 1`, since the patterns `True` and `False` are incompatible. The compatibility of patterns is defined as follows:

$$\mathsf{compat}(p_1, p_2) = \begin{cases} \text{true} & \text{if } p_1 \text{ or } p_2 \text{ is a variable pattern} \\ \text{true} & \text{if } p_1 = p_2 \text{ are literal pattern} \\ \mathsf{compat}(p'_1, p_2) & \text{if } p_1 = x\texttt{@}p'_1 \\ \mathsf{compat}(p_1, p'_2) & \text{if } p_2 = x\texttt{@}p'_2 \\ \bigwedge_{i=1}^{n} \mathsf{compat}(q_i, q'_i) & \text{if } p_1 = C\ \overline{q_n} \text{ and } p_2 = C\ \overline{q'_n} \\ \text{false} & \text{otherwise} \end{cases}$$

Lambda Abstractions and Conditionals

NoOpExpr ::= ...
| `\` *SimplePat*$_1$ ... *SimplePat*$_n$ `->` *Expr* (*lambda expression*, $n \geqslant 1$)
| `if` *Expr* `then` *Expr* `else` *Expr* (*conditional*)

Lambda expressions denote local anonymous functions, and are converted into local function declarations by means of fresh function names. Conditionals expressed by

`if-then-else` are converted into a corresponding `case` expression. Note that the `case` expression is rigid, i. e., if the condition evaluates to a logic variable, evaluation of the entire expression will be suspended.

$$\begin{aligned}
\mathsf{dsExpr}(\backslash\overline{p_n}\ \texttt{->}\ e) &= \mathsf{dsExpr}(\texttt{let}\ f\ \overline{p_n}\ \texttt{=}\ e\ \texttt{in}\ f) \quad \text{where } f \text{ is a fresh name}\\
\mathsf{dsExpr}(\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3) &= \texttt{case}\ \mathsf{dsExpr}(e_1)\ \texttt{of}\\
&\qquad \texttt{True}\ \ \texttt{->}\ \mathsf{dsExpr}(e_2)\\
&\qquad \texttt{False}\ \texttt{->}\ \mathsf{dsExpr}(e_3)
\end{aligned}$$

Local Declarations

$$\begin{aligned}
\mathit{NoOpExpr} &::= \ldots \mid \texttt{let}\ \mathit{LocalDecls}\ \texttt{in}\ \mathit{Expr} && (\textit{let expression})\\
\mathit{LocalDecls} &::= \texttt{\{}\ \mathit{LocalDecl}_1\ \texttt{;}\ \ldots\ \texttt{;}\ \mathit{LocalDecl}_n\ \texttt{\}} && (n \geqslant 0)\\
\mathit{LocalDecl} &::= \mathit{FunctionDecl}\\
&\mid \mathit{PatternDecl}\\
&\mid \mathit{Variable}_1\ \texttt{,}\ \ldots\ \texttt{,}\ \mathit{Variable}_n\ \texttt{free} && (n \geqslant 1)\\
\mathit{PatternDecl} &::= \mathit{Pattern}\ \mathit{Rhs}
\end{aligned}$$

The syntax of Curry allows the local declaration of functions, patterns and logic variables inside expressions by means of `let` expressions. The desugaring of local declarations is rather simple, since it reduces to other transformations in most cases. In particular, a declaration of logic variables remains unchanged, and local function declarations will be desugared just like global function declarations (see Section 4.2.4). For the transformation of local pattern declarations, we denote a pattern declaration for a pattern p and a right-hand side r by $p \leftarrow r$, due to the different syntactic forms of the right-hand side. Pattern declarations are desugared by individually considering the pattern and the right-hand side, where new declarations originating from the pattern are added to the right-hand side like for `case` expressions. We can reuse the operation dsRhs, since the right-hand sides of patterns and `case` expressions share the same structure and only differ in the connection symbol.

$$\begin{aligned}
\mathsf{dsExpr}(\texttt{let}\ \overline{d_n}\ \texttt{in}\ e) &= \texttt{let}\ \bigcup\overline{\mathsf{dsDecl}(d_n)}\ \texttt{in}\ \mathsf{dsExpr}(e)\\
\mathsf{dsDecl}(\overline{x_n}\ \texttt{free}) &= \overline{x_n}\ \texttt{free}\\
\mathsf{dsDecl}(\mathit{fun}) &= \mathsf{dsFun}(\mathit{fun}) \quad \text{where } \mathit{fun} \text{ is a function declaration}\\
\mathsf{dsDecl}(p \leftarrow r) &= (p' \texttt{ = } \mathsf{dsRhs}(\texttt{failed}, r)) \cdot \overline{\mathsf{dsDecl}(d_m)} \quad \text{where } (p', \overline{d_m}) = \mathsf{dsPat}(p)
\end{aligned}$$

Arithmetic Sequences

$$\mathit{BasicExpr} ::= \ldots \mid \texttt{[}\ \mathit{Expr}\ [\texttt{,}\ \mathit{Expr}]\ \texttt{..}\ [\mathit{Expr}]\ \texttt{]} \qquad (\textit{arithmetic sequence})$$

The arithmetic sequence `[` e_1`,` e_2 `..` e_3 `]` denotes the list of elements e_1, e_2 up to e_3, where the difference between two elements e_i, e_{i+1} is computed from the difference of e_1 and e_2. Thus, if $e_2 < e_1$, the numbers will be in decreasing order. In general, e_2 or e_3 can be omitted, so that there exist four different variants [Han12]:

- `[` e `..` `]` denotes the infinite list `[` e, $e+1$, ... `]`.
- `[` e_1, e_2 `..` `]` denotes the infinite list `[` e_1, e_1+i, e_1+2*i, ... `]`, where $i = e_2 - e_1$. Note that i can be positive, zero, or negative, which results in increasing, constant, or decreasing elements.
- `[` e_1 `..` e_2 `]` denotes the finite list `[` e_1, e_1+1, e_1+2, ..., e_2 `]`. If $e_2 < e_1$, then the resulting list is empty.
- `[` e_1, e_2 `..` e_3 `]` denotes the finite list `[` e_1, e_1+i, e_1+2*i, ..., e_3 `]`, where $i = e_2 - e_1$. If there is no m such that $e_3 = e_1 + m*i$, then e_3 is not contained in the list.

The arithmetic sequences are transformed into calls of auxiliary functions, which are defined in the module `Prelude`:

$$\begin{aligned}
\mathsf{dsExpr}(\texttt{[}\ e\ \texttt{..}\ \texttt{]}) &= \mathsf{dsExpr}(\texttt{enumFrom}\ e)\\
\mathsf{dsExpr}(\texttt{[}\ e_1\texttt{,}\ e_2\ \texttt{..}\ \texttt{]}) &= \mathsf{dsExpr}(\texttt{enumFromThen}\ e_1\ e_2)\\
\mathsf{dsExpr}(\texttt{[}\ e_1\ \texttt{..}\ e_2\ \texttt{]}) &= \mathsf{dsExpr}(\texttt{enumFromTo}\ e_1\ e_2)\\
\mathsf{dsExpr}(\texttt{[}\ e_1\texttt{,}\ e_2\ \texttt{..}\ e_3\ \texttt{]}) &= \mathsf{dsExpr}(\texttt{enumFromThenTo}\ e_1\ e_2\ e_3)
\end{aligned}$$

List Comprehensions

BasicExpr	::=	... | `[` *Expr* `|` $Qual_1$ `,` ... `,` $Qual_n$ `]`	(*list comprehension*, $n \geqslant 1$)
Qual	::=	*Pattern* `<-` *Expr*	(*generator*)
	|	`let` *LocalDecls*	(*local declarations*)
	|	*Expr*	(*guard*)

A list comprehension of the form `[` e `|` $\overline{q_n}$ `]` consists of an expression e describing the elements of the resulting list, where each of the *qualifiers* $\overline{q_n}$ is either

- a *generator* of the form p `<-` l, where p is a local pattern and l is an expression of type `[`τ`]`, or
- a *guard* of the form b, where b must be an expression of type `Bool`, or
- a local declaration of the form `let` *ds*, where *ds* is a sequence of declarations.

The variables introduced in a qualifier by a pattern or a declaration can be used in subsequent qualifiers, as well as in the expression e. A list comprehension denotes the list of elements that result from evaluating the qualifiers in depth-first order such that all Boolean guards are satisfied. For example, the list comprehension

```
[(x, y, z) | z <- [1..10], y <- [1..z], x <- [1..y], x*x + y*y == z*z ]
```

denotes the list of Pythagorean triples `[(3,4,5),(6,8,10)]`. The transformation of list comprehensions is defined as follows, where `concatMap` is also defined in the `Prelude`:

$$
\begin{aligned}
\mathsf{dsExpr}(\texttt{[}\ e\ \texttt{|}\ \overline{q_n}\ \texttt{]}) &= \mathsf{dsExpr}(\mathsf{dsLC}(\overline{q_n}, e)) \\
\mathsf{dsLC}(\varepsilon, e) &= \texttt{[}e\texttt{]} \\
\mathsf{dsLC}(b \cdot \overline{q_m}, e) &= \texttt{if}\ b\ \texttt{then}\ \mathsf{dsLC}(\overline{q_m}, e)\ \texttt{else}\ \texttt{[]} \\
\mathsf{dsLC}(\texttt{let}\ ds \cdot \overline{q_m}, e) &= \texttt{let}\ ds\ \texttt{in}\ \mathsf{dsLC}(\overline{q_m}, e) \\
\mathsf{dsLC}(p\ \texttt{<-}\ e' \cdot \overline{q_m}, e) &= \texttt{let}\ \ \texttt{ok}\ p''\ \ \texttt{=}\ \texttt{let}\ ds\ \texttt{in}\ \mathsf{checkLits}(\overline{(c_l, z_l)}, \mathsf{dsLC}(\overline{q_m}, e)) \\
&\qquad\quad\ \ \texttt{ok}\ p_1\ \ \texttt{=}\ \texttt{[]} \\
&\qquad\quad\ \ \ldots \\
&\qquad\quad\ \ \texttt{ok}\ p_k\ \ \texttt{=}\ \texttt{[]} \\
&\qquad \texttt{in}\ \ \ \texttt{concatMap ok}\ e' \\
&\qquad \text{where } \texttt{ok} \text{ is a fresh name, } (p', ds) = \mathsf{dsPat}(p), \\
&\qquad (p'', \overline{(c_l, z_l)}) = \mathsf{removeLits}(p'), \text{ and } \overline{p_k} = \mathsf{compl}(p'')
\end{aligned}
$$

Within the last rule, we first desugar the pattern p to a a simplified pattern p' and a sequence of declarations ds. In p', we then replace all literal patterns $\overline{c_l}$ by fresh variables $\overline{z_l}$ to obtain p'', and compute the set of complementary constructors of p'' using the following operation:

$$
\begin{aligned}
\mathsf{compl}(x) &= \{\} \\
\mathsf{compl}(x\texttt{@}p) &= \mathsf{compl}(p) \\
\mathsf{compl}(C\ \overline{p_n}) &= \{C_1\ \overline{x_{1n_1}}\ \ldots\ C_k\ \overline{x_{kn_k}}\} \\
&\quad \cup \{C\ p_1 \ldots p_{i-1}\ p'\ y_{i+1}\ \ldots\ y_n \mid i \in \{1, \ldots, n\}, p' \in \mathsf{compl}(p_i)\} \\
&\quad \text{where } \{C, C_1, \ldots, C_k\} \text{ are all constructors of the result type of } C, \\
&\quad C_i \text{ has arity } n_i, \text{ and all } x_{ij} \text{ and } y_i \text{ are fresh variables}
\end{aligned}
$$

We finally add an additional check for the extracted literals to the right-hand side of `ok` by the operation `checkLits`:

$$
\mathsf{checkLits}(\overline{(c_l, z_l)}, e) = \begin{cases} e & \text{if } l = 0 \\ \texttt{if}\ z_1\ \texttt{==}\ c_1\ \texttt{\&\&}\ \ldots\ \texttt{\&\&}\ z_l\ \texttt{==}\ c_l\ \texttt{then}\ e\ \texttt{else}\ \texttt{[]} & \text{if } l > 0 \end{cases}
$$

This special treatment of literal patterns is unfortunately necessary since there may exist an infinite set of literals, and thus no finite set of complementary literal patterns can be computed. For instance, the list comprehension

```
[ y | (2, y) <- [(0,1), (1, 2), (2, 4), (3, 8)]]
```

will be then transformed (modulo further desugaring) into

```
let ok (x, y) = if x == 2 then [y] else []
in  concatMap ok [(0,1), (1, 2), (2, 4), (3, 8)]
```

For our introductory example of Pythagorean triples, the list comprehension is transformed to the following expression (modulo further desugaring):

```
let ok1 z = let ok2 y = let ok3 x = if x*x + y*y == z*z then [(x, y, z)] else []
                        in  concatMap ok3 [1..y]
            in  concatMap ok2 [1..z]
in  concatMap ok1 [1..10]
```

Do-Notation

$$
\begin{array}{rcll}
\mathit{NoOpExpr} & ::= & \ldots \mid \texttt{do \{}\ \mathit{Stmt}_1\ \texttt{;}\ \ldots\ \texttt{;}\ \mathit{Stmt}_n\ \texttt{;}\ \mathit{Expr}\ \texttt{\}} & (\textit{do expression},\ n \geqslant 0) \\
\mathit{Stmt} & ::= & \mathit{Pattern}\ \texttt{<-}\ \mathit{Expr} & \\
& \mid & \texttt{let}\ \mathit{LocalDecls} & \\
& \mid & \mathit{Expr} &
\end{array}
$$

To facilitate the notation of monadic IO operations, the Curry syntax provides the so-called `do`-notation as a special construct for sequences of monadic actions, such as

```
main = do
  name <- getLine
  let greeting = "Hello " ++ name
  putStrLn name
```

This syntax mimics the traditional notation of statement sequences known from imperative programming languages, and is desugared using predefined operators by the following transformation:

$$
\begin{aligned}
\mathsf{dsExpr}(\texttt{do}\ \overline{s_n}\texttt{;}\ e) &= \mathsf{dsExpr}(\mathsf{dsDo}(\overline{s_n}, e)) \\
\mathsf{dsDo}(\varepsilon, e) &= e \\
\mathsf{dsDo}(e'\texttt{;}\ \overline{s_n}, e) &= e'\ \texttt{>>}\ \mathsf{dsDo}(\overline{s_n}, e) \\
\mathsf{dsDo}(p\ \texttt{<-}\ e'\texttt{;}\ \overline{s_n}, e) &= e'\ \texttt{>>=}\ \texttt{\textbackslash}p\ \texttt{->}\ \mathsf{dsDo}(\overline{s_n}, e) \\
\mathsf{dsDo}(\texttt{let}\ ds\texttt{;}\ \overline{s_n}, e) &= \texttt{let}\ ds\ \texttt{in}\ \mathsf{dsDo}(\overline{s_n}, e)
\end{aligned}
$$

The example given above will thus be transformed into

```
main = getLine >>= \name ->
       let greeting = "Hello " ++ name
       in  putStrLn name
```

Record Construction and Update

$$
\begin{array}{rcll}
\mathit{BasicExpr} & ::= & \ldots & \\
& \mid & \mathit{QDataConstr}\ \texttt{\{}\ \mathit{FBind}_1\ \texttt{,}\ \ldots\ \texttt{,}\ \mathit{FBind}_n\ \texttt{\}} & (\textit{record construction},\ n \geqslant 0) \\
& \mid & \mathit{BasicExpr}_{\langle \mathit{QDataConstr} \rangle}\ \texttt{\{}\ \mathit{FBind}_1\ \texttt{,}\ \ldots\ \texttt{,}\ \mathit{FBind}_n\ \texttt{\}} & (\textit{record update},\ n \geqslant 1) \\
\mathit{FBind} & ::= & \mathit{QLabel}\ \texttt{=}\ \mathit{Expr} &
\end{array}
$$

Curry provides a special syntax that allows the construction and update of record values using field labels. If we consider the (expanded) record declaration of the introductory example

```
data Person = Lecturer { name :: String, office :: String }
            | Student  { name :: String, studnr :: Int    }
```

then a new student can be constructed using the following expression:

```
student = Student { studnr = 1234 }
```

Note that the order of the fields in the construction may be arbitrary, provided that each field occurs at most once and is defined for the respective constructor. If some fields are omitted, then they are initialized with a call to the function `unknown` representing a fresh logic variable.

In analogy to the record construction, a value can be updated like in the following example:

```
setName newName person = person { name = newName }
```

Again, the order of the fields may be arbitrary, where each field may occur at most once and must be defined for the type of the updated expression. For omitted fields, the original value of the respective component is kept. In contrast to the construction of record values, the update operation is applicable to values with *different constructors*, provided that the respective constructor defines all mentioned field labels.

The transformation of a record construction or update is defined as follows, where the auxiliary function pick has already been introduced for the desugaring of record patterns.

$$\begin{aligned}
\mathsf{dsExpr}(C\ \{\ bs\ \}) &= \mathsf{dsExpr}(C\ \mathsf{pick}_1^C(bs, \mathtt{unknown})\ \ldots\ \mathsf{pick}_k^C(bs, \mathtt{unknown})) \\
&\quad \text{where } k \text{ is the arity of } C \\
\mathsf{dsExpr}(e\ \{\ bs\ \}) &= \mathsf{dsExpr}(\mathtt{fcase}\ e\ \mathtt{of}\ \overline{b_k}) \\
&\quad \text{where } \{\overline{C_k}\} \text{ are those constructors of the type of } e \text{ containing all} \\
&\quad \text{labels in } bs,\ b_i = C_i\ \overline{x_{n_i}}\ \texttt{->}\ C_i\ \mathsf{pick}_1^{C_i}(bs, x_1)\ \ldots\ \mathsf{pick}_{n_i}^{C_i}(bs, x_{n_i}) \text{ for} \\
&\quad i \in \{1, \ldots, k\},\ n_i \text{ is the arity of } C_i, \text{ and } \overline{x_{n_i}} \text{ are fresh variables}
\end{aligned}$$

Thus, the two examples above are transformed into

```
student = Student unknown 1234
setName newName person = fcase person of
  Lecturer x1 x2 -> Lecturer newName x2
  Student  x1 x2 -> Student  newName x2
```

4.2.4 Function Declarations

$$\begin{aligned}
\mathit{FunctionDecl} &::= \ldots \mid \mathit{Equation} \\
\mathit{Equation} &::= \mathit{FunLhs}\ \mathit{Rhs} \\
\mathit{FunLhs} &::= \mathit{Function}\ \mathit{SimplePat}_1\ \ldots\ \mathit{SimplePat}_n && (n \geqslant 0) \\
&\mid\ \mathit{ConsPattern}\ \mathit{FunOp}\ \mathit{ConsPattern} \\
&\mid\ (\ \mathit{FunLhs}\ )\ \mathit{SimplePat}_1\ \ldots\ \mathit{SimplePat}_n && (n \geqslant 1)
\end{aligned}$$

$$
\begin{array}{rcl}
\mathit{Rhs} & ::= & \texttt{=}\ \mathit{Expr}\ [\texttt{where}\ \mathit{LocalDecls}] \\
& \mid & \mathit{CondExprs}\ [\texttt{where}\ \mathit{LocalDecls}] \\
\mathit{CondExprs} & ::= & \texttt{|}\ \mathit{InfixExpr}\ \texttt{=}\ \mathit{Expr}\ [\mathit{CondExprs}]
\end{array}
$$

A function declaration in Curry may consist of multiple equations, and all equations defining the same function must share the same number of patterns. A single equation consists of a left-hand side, composed of the function's name and the argument patterns, and a right-hand side which can either be a simple expression or a sequence of conditional expressions. In addition, the right-hand side of a function declaration may also contain local declarations for patterns, free variables, and local functions. Since there again exist different syntactic forms for both sides of a function declaration, we use the notation $l \to r$ to uniformly refer to an equation consisting of the left-hand side l and the right-hand side r.

Generally, the desugaring of a function declaration begins with the desugaring of the left-hand side before the right-hand side is desugared, since the former may influence the latter. To start with, the syntactic variance for the left-hand side is eliminated by a transformation to prefix notation, i. e., the function's name followed by a sequence of patterns:

$$
\begin{array}{rcll}
\mathsf{flatLhs}(f\ \overline{p_n}) & = & f\ \overline{p_n} & \\
\mathsf{flatLhs}(p_1\ \oplus\ p_2) & = & (\oplus)\ p_1\ p_2 & \text{where } \oplus \text{ is an infix operator} \\
\mathsf{flatLhs}(p_1\ \texttt{`}f\texttt{`}\ p_2) & = & f\ p_1\ p_2 & \\
\mathsf{flatLhs}((l)\ \overline{p_n}) & = & \mathsf{flatLhs}(l)\ \overline{p_n} &
\end{array}
$$

For the desugaring of the patterns of the left-hand side, we have to consider two special cases, namely *non-linear* and *functional* patterns. These kinds of patterns are only allowed in the left-hand side of function declarations, but not in pattern declarations nor `case` expressions. Since non-linear patterns stretch over multiple patterns, we will consider their transformation first, before we continue with the transformation of functional and other patterns. Note that we deliberately reuse the notions of positions, substitutions and alike introduced in the context of term rewriting, since these can easily be extended to the considered patterns and expressions.

Non-Linear Patterns

If the left-hand side of a function declaration is non-linear, i. e., if it contains multiple occurrences of the same variable, these occurrences are considered to denote equational constraints between the respective arguments. Thus, the function declaration

```
same (x, y) (x, y) = success
```

is considered to be an abbreviation of

```
same (x, y) (x1, y1) | x =:= x1 & y =:= y1 = success
```

Generally, the equational constraints originating from non-linear patterns are solved after regular pattern matching, but before any constraints of the right-hand side.

Note that it is an error for any repeated variable to occur below an irrefutable pattern, since the strict equality implied by non-linear patterns conflicts with the lazy semantics of irrefutable patterns. Furthermore, multiple occurrences of the same variable inside a *functional pattern* only result in an equational constraint if the variables also occur outside the functional pattern. Thus, the functions

```
f x (pair x x) = x
g   (pair x x) = x
```

are transformed into

```
f x (pair x1 x1) | x =:= x1 = x
g   (pair x  x )            = x
```

This behavior is reasonable because the result of the expression `pair x x` may or may not contain the variable `x`, depending on the implementation of the function `pair`. Therefore, it is in the responsibility of functional patterns to cope with non-linearity in their arguments.

The transformation of non-linear patterns replaces repetitive occurrences of the same variable by fresh variables, and yields a sequence of linearized patterns (up to non-linearity inside single functional patterns) as well as a sequence of equational constraints which will later be added to the right-hand side.

$$\mathsf{dsNonLinear}(\overline{p_n}) = \mathsf{dsNL}(\overline{p_n}, \varepsilon)$$

$$\mathsf{dsNL}(\overline{p_n}, cs) = \begin{cases} \mathsf{dsNL}(\overline{p_n}[\sigma(p')]_q, cs \cdot c) & \text{if } p' = \overline{p_n}|_q \text{ is the leftmost outermost pattern in } \overline{p_n} \text{ such that } p' \text{ is a variable pattern } x \text{ or a functional pattern containing the variable } x, \text{ and } x \text{ occurs left or above of } q \text{ in } \overline{p_n}, \text{ where } y \text{ is a fresh variable, } \sigma = \{x \mapsto y\}, \text{ and } c = x \texttt{ =:= } y \\ (\overline{p_n}, cs) & \text{otherwise} \end{cases}$$

Functional Patterns

SimplePat ::= ...
| (*QFunction SimplePat*$_1$ ... *SimplePat*$_n$) (*functional pattern*, $n \geqslant 1$)
| (*ConsPattern QFunOp Pattern*) (*infix functional pattern*)

Functional patterns [AH05] allow the abbreviation of multiple patterns in the left-hand side of a function declaration by means of defined operations. Conceptually, a functional pattern represents the set of all patterns (constructor terms) the functional pattern evaluates to if interpreted as an expression. For instance, in the function declaration

```
last (xs ++ [x]) = x
```

the functional pattern `xs ++ [x]` represents the (infinite) set of constructor patterns for finite, non-empty lists. Thus, this declaration is an abbreviation for the following set of equations:

```
last [x]       = x
last [_, x]    = x
last [_, _, x] = x
⋮
```

Note that functional patterns are not allowed to occur below an irrefutable pattern, since functional patterns are evaluated during pattern matching which conflicts with the lazy nature of irrefutable patterns. Furthermore, irrefutable patterns are not allowed inside functional patterns, since their evaluation would depend on the semantics of the surrounding function, and thus their laziness cannot be guaranteed.

Functional patterns may potentially abbreviate an *infinite* sets of patterns so that they cannot be transformed into regular patterns at compile time, but are instead evaluated at run time using an auxiliary operation. Since this transformation idea requires the evaluation of the operation occurring in the functional pattern, circular dependencies of operations defined using functional patterns must be avoided. For instance, definitions like

```
(xs ++ ys) ++ zs = xs ++ (ys ++ zs)
```

are not allowed, since the meaning of the operation "++" would depend on its own definition. This requirement can be formalized by assigning a *level mapping* to operations representing their dependencies.

Definition 4.1 (Level Mapping [AH05])**.** *A level mapping l for a functional logic program P without functional patterns is a mapping from functions defined in P to natural numbers such that, for all rules $f\ \overline{p_n} \to r$, if g is a function occurring in r, then $l(g) \leqslant l(f)$.*

Based on this definition, we can define the set of *stratified* programs which contain only occurrences of functional patterns that depend on global operations defined on a lower level.

Definition 4.2 (Stratified Program [AH05])**.** *A functional logic program P with functional patterns is stratified if there exists a level mapping l for P such that for all rules $f\ \overline{p_n} \to r$ it holds that if g is a defined function occurring in $\overline{p_n}$, then g is globally defined and $l(g) < l(f)$.*

The restriction to stratified programs ensures that operations defined using functional patterns do not depend, directly or indirectly, on their own definition by operation calls. The restriction to *globally defined* operations furthermore ensures that an operation occurring in a functional pattern cannot access variables bound in a surrounding scope, which will be motivated after presentation of the transformation idea.

The central idea for the implementation of functional patterns is to evaluate such a pattern as an expression where the variables of the patterns are considered as logic variables, and then interpret its values as ordinary patterns. Conceptually, functional patterns are transformed as follows [AH05]:

$$\mathsf{elim}(f\ \overline{p_n}\ \texttt{|}\ c\ \texttt{=}\ r) = \begin{cases} \mathsf{elim}\left(\begin{array}{l} f\ \overline{p_{i-1}}\ x\ p_{i+1} \ldots p_n \\ \quad \texttt{|}\ p_i\ \texttt{=:<=}\ x\ \texttt{\&>}\ c\ \texttt{=}\ r \end{array}\right) & \text{if } p_i \text{ is the leftmost functional pattern and } x \text{ a fresh variable} \\ f\ \overline{p_n}\ \texttt{|}\ c\ \texttt{=}\ r & \text{otherwise} \end{cases}$$

If a functional pattern p_i contains variables, these are furthermore added as logic variables to the respective rule. Thus, the definition of function `last` will be transformed into the definition

```
last ys | xs ++ [x] =:<= ys = x
  where xs, x free
```

This transformation relies on the presence of a functional pattern unification operator "`=:<=`" as a primitive operation. Informally, e_1 `=:<=` e_2 evaluates its first argument e_1 to a head normal form h_1. If h_1 is a logic variable, then the variable is bound to e_2, such that the variable is internally converted from a logic to a pattern variable. If instead h_1 is rooted by a constructor, then e_2 is evaluated to a head normal form h_2. If h_2 is a logic variable, it is instantiated to the constructor of h_1 applied to fresh logic variables, and the matching proceeds recursively. If h_2 is rooted by the same constructor, both h_1 and h_2 are recursively matched, and if they are rooted by different constructors, the operation fails.

Since the transition of functional pattern variables to logic variables and vice versa does not comply with the standard semantics of logic variables, it must not be observable in the program. Because the logic variables are *locally* introduced and only accessible on the right-hand side of an equation *after* the functional pattern unification, this can be guaranteed as long as no locally defined operations occur in the functional patterns, as the following example demonstrates.

Example 4.3 (Functional Pattern with Locally Defined Operation)**.** *Consider the following (complicated) equality operation*

```
eq :: a -> a -> Bool
eq x y = h y
  where
  g True  = x
  h (g a) = a
```

where the locally defined function `g` *occurs in the functional pattern* `g a` *of* `h`*. If we applied the above transformation, we would gain the following result:*

```
eq :: a -> a -> Bool
eq x y = h y
  where
```

```
g True  = x
h z     | g a =:<= z = a where a free
```

Note that `g a` *will now evaluate to the value of* `x`*, and if* `x` *evaluates to a logic variable, this variable will be bound to the expression of the second parameter* `y`*. Thus, the transition of the logic variable* `x` *becomes observable, which violates its semantics.*

While the sketched transformation scheme presents the central idea, it is not applicable in general, since it neither considers functional patterns below constructors nor special pattern syntax inside a functional pattern that is not allowed in expressions. The general transformation therefore considers a sequence of patterns and yields a triple containing the transformed pattern sequence, a sequence of logic variables to be introduced, and a sequence of functional pattern unification constraints to be added to the right-hand side. Furthermore, the order of functional patterns in the left-hand side is retained in this transformation, in contrast to the original transformation proposed by Antoy and Hanus [AH05].

$$\mathsf{dsFunPats}(\overline{p_n}) = \mathsf{dsFP}(\overline{p_n}, \varepsilon, \varepsilon)$$

$$\mathsf{dsFP}(\overline{p_n}, xs, cs) = \begin{cases} \mathsf{dsFP}(\overline{p_n}[x]_q, xs \cdot x, cs \cdot c) & \text{if } \overline{p_n}|_q \text{ is the leftmost outermost functional pattern, } x \text{ a fresh variable, and } c = \mathsf{genFP}(\overline{p_n}|_q, x) \\ (\overline{p_n}, xs, cs) & \text{otherwise} \end{cases}$$

The operation $\mathsf{genFP}(p, x)$ basically generates the constraint p `=:<=` x, but furthermore extracts nested as-patterns since they are not allowed in expressions.

$$\mathsf{genFP}(p, x) = \begin{cases} \mathsf{genFP}(p[v]_q) \cdot \mathsf{genFP}(p', v) & \text{if } p|_q = v\texttt{@}p' \text{ is the leftmost outermost as-pattern in } p \\ p \texttt{ =:<= } x & \text{otherwise} \end{cases}$$

The conversion of as-patterns inside functional patterns allows certain subpatterns to be referenced by a new variable, just like as-patterns for ordinary patterns. For instance, this allows the definition

```
f (_ ++ x@[(42,_)] ++ _) = x
```

to be transformed to

```
f xs | (_ ++ x ++ _ =:<= xs) & ([(42, _)] =:<= x) = x  where x free
```

Entire Transformation

Based on the presented transformations of non-linear and functional patterns, we can proceed with the formal transformation of a function's equation. This transformation applies the single transformations of flatLhs, dsNonLinear and dsFunPats successively. Furthermore, the variables occurring in functional patterns have to be declared as

free variables, and the constraints of non-linear and functional patterns have to be added to the right-hand side:

$$\begin{aligned}
&\mathsf{dsFun} : \mathit{FunctionDecl} \to \mathit{FunctionDecl}\\
&\mathsf{dsFun}(l \to r) = f\ \overline{p_n^3}\ \texttt{=}\ \texttt{let}\ ds\ \texttt{in}\ \mathsf{addC}(cs_2, \mathsf{addC}(cs_1, e))\\
&\qquad\qquad f\ \overline{p_n} &&= \mathsf{flatLhs}(l)\\
&\qquad\qquad \overline{(p_n^1, cs_1)} &&= \mathsf{dsNonLinear}(\overline{p_n})\\
&\text{where}\qquad \overline{(p_n^2, xs, cs_2)} &&= \mathsf{dsFunPats}(\overline{p_n^1})\\
&\qquad\qquad \overline{(p_n^3, ds_n)} &&= \overline{\mathsf{dsPat}(p_n^2)}\\
&\qquad\qquad \texttt{let}\ ds\ \texttt{in}\ e &&= \mathsf{dsRhs}(\texttt{failed}, \mathsf{addDecls}(xs\ \texttt{free} \cdot \bigcup \overline{ds_n}, r))\\
&\mathsf{addC}(\overline{c_n}, e) = \begin{cases} e & \text{if } n = 0\\ (c_1\ \texttt{\&}\ \ldots\ \texttt{\&}\ c_n)\ \texttt{\&>}\ e & \text{if } n > 0 \end{cases}
\end{aligned}$$

Note that the constraints in the sequences cs_1 and cs_2 are combined with the concurrent constraint conjunction to allow them to be solved in an arbitrary order. Furthermore, the declaration xs `free` is only added if the sequence of variables xs is non-empty. Since the desugaring of patterns may introduce new declarations, these declarations are added to the right-hand side of the equation by the operation addDecls, which is then desugared by reuse of the operation dsRhs. In summary, this transformation achieves the following order of pattern matching for function declarations:

1. constructor patterns,
2. functional patterns,
3. non-linear patterns,
4. local guards.

For instance, if we consider the following declaration combining a constructor pattern, a functional pattern, an irrefutable pattern, a non-linear left-hand side and a guard in the right-hand side

```
f [x] (id x) ~True | null x = x
```

then this definition will be desugared into

```
f [x] y z = let x1 free
                True = z
            in (id x1 =:<= y) &> (x =:= x1) &> case null x of
                 True  -> x
                 False -> failed
```

so that first the constructor pattern is matched, followed by the functional pattern, the non-linearity constraint, and finally the guard of the right-hand side.

4.3 Simplification

The next step in the process of normalization is concerned with the simplification of the program by reducing the number of local declarations. While the declaration of logic variables can only be reduced for unused variables, the declaration of non-variable patterns will be removed entirely. This is achieved by the following steps:

1. Local pattern declarations are eliminated by replacing them with variable bindings and local selection functions.

2. The sequence of declarations inside a single `let` expression is rearranged to the order of their dependencies, and structured into minimal binding groups. Furthermore, in certain cases some variables are replaced by their binding to reduce the syntactic complexity and remove unused bindings.

Note that this transformation does not remove *all* local patterns, in contrast to the elimination specified by the Curry Report [Han12]. This deviation is intended, in order to allow the notion of *mutually recursive* pattern declarations, which are excluded by the transformation scheme proposed in the Curry Report.

In the following, we provide each transformation as a single step which is repeatedly applied to the right-hand side of all equations until a fixpoint of the respective transformation is reached. In the formalization, we denote by $e = C[e']_q$ an expression e where q is the leftmost outermost position in e such that $e|_q = e'$ and e' is surrounded by the context C in e.

4.3.1 Elimination of Pattern Declarations

As the first step of the transformation, the leftmost outermost `let` expression containing at least one non-variable pattern is selected. The right-hand sides of the non-variable pattern declarations are then extracted and bound to fresh variables. Since the declarations of a `let` expression share the same scope, the semantics does not depend on their order, and we assume the pattern declarations to come first:

$$C[\texttt{let } \{\overline{p_k = e_k};\ ds\} \texttt{ in } e]_q \to C[\texttt{let } \{\overline{p_k = x_k};\ \overline{x_k = e_k};\ ds\} \texttt{ in } e]_q$$

where $\overline{p_k}$ are non-variable patterns, and $\overline{x_k}$ fresh variables

In the next step, the non-variable patterns are replaced by bindings for the variables occurring in the removed patterns, which is achieved by means of the following transformation:

$$C[\texttt{let } \{p = x;\ ds\} \texttt{ in } e]_q \to C[\texttt{let } \{ds;\ \overline{x_k = \texttt{let } \{f_k\ p' = y_k\} \texttt{ in } f_k\ x}\} \texttt{ in } e]_q$$

where p is a non-variable pattern, $\{\overline{x_k}\} = \mathcal{V}ar(p)$, $\overline{f_k}$ are fresh function symbols and $\overline{y_k}$ fresh variables, and $p' = \sigma(p)$ for $\sigma = \{\overline{x_k \mapsto y_k}\}$

For instance, if we consider the following pattern declaration

```
let (x, y) = (1, 2) in x + y
```

then the first transformation produces the intermediate result

```
let (x, y) = v
    v      = (1, 2)
in x + y
```

while the second transformation yields

```
let x = let f1 (x', y') = x' in f1 v
    y = let f2 (x', y') = y' in f2 v
    v = (1, 2)
in  x + y
```

so that the non-variable pattern declarations are eliminated.

4.3.2 Inlining of Declarations

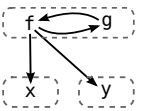
To further reduce the complexity of local declarations, unnecessary bindings are removed in the next step. For this purpose, first nested `let` declarations are lifted to the surrounding binding group by means of the following transformation:

$$C[\texttt{let } \{x = \texttt{let } \{ds'\} \texttt{ in } e'; \; ds\} \texttt{ in } e]_q \to C[\texttt{let } \{ds'; x = e'; \; ds\} \texttt{ in } e]_q$$

Afterwards, the declarations inside a single binding group will be sorted according to their dependency relation, and then separated into the strongly connected components of the corresponding dependency graph. For instance, if we consider the local declaration

```
let x = 1
    y free
    f b = if b then x else g y
    g z = 2 * f True
in  f False
```

then `x` and `y` are referenced by `f`, where `f` and `g` mutually depend on each other. Thus, the corresponding dependency graph is



where an arrow from `f` to `x` indicates that `f` depends on `x`, and the strongly connected components are surrounded by dashed lines. According to the dependency graph, the declarations will be rearranged to

```
let x = 1
in  let y free
```

```
in  let f b = if b then x else g y
        g z = 2 * f True
    in  f False
```

where the declarations of `x` and `y` may occur in any order. For this rearrangement, we assume an operation scc that computes the strongly connected components of a sequence of declarations in a topological sorting, such that in the resulting sequence of strongly connected components, every component depends only on preceding ones. Based on this operation, we apply the following transformation:

$$C[\texttt{let } \{ds\} \texttt{ in } e]_q \to C[\texttt{let } \{ds_1\} \texttt{ in } \ldots \texttt{ let } \{ds_n\} \texttt{ in } e]_q \quad \text{where } \overline{ds_n} = \mathsf{scc}(ds)$$

Since the declarations of free variables do not depend on any other declaration, each of them could be declared in its own declaration group. However, we will also allow multiple logic variables to be placed in the same group to allow more compact representations.

To further simplify the expressions, two more transformations are applied. Firstly, any unused declaration of a free variable, local variable or local function is removed:

$$\begin{aligned} C[\texttt{let } \overline{x_n} \texttt{ free in } e]_q &\to C[\texttt{let } \{\overline{x_n}\} \cap \mathcal{V}ar(e) \texttt{ free in } e]_q \\ C[\texttt{let } \{\} \texttt{ in } e]_q &\to C[e]_q \\ C[\texttt{let } \{\overline{x_k = e_k};\ \overline{f_l\ \overline{p_{n_l}} = e_l}\} \texttt{ in } e]_q &\to C[e]_q \quad \text{if } \{\overline{x_k}\} \cap \mathcal{V}ar(e) = \varnothing \text{ and } \{\overline{f_l}\} \cap \mathsf{funs}(e) = \varnothing \end{aligned}$$

In this definition, $\mathsf{funs}(e)$ yields the set of function symbols that occur in e. Note that the third case also applies for $k = 0$ or $l = 0$, so that unused variables as well as unused functions are removed whenever possible.

Secondly, a variable x is replaced by its definition e (also referred to as *inlining* [PM02]) if the definition is either a value (variable, literal, constructor or non-nullary function symbol), or an expression in which x does not occur, supposed that x occurs only once in the subjacent expression:

$$C[\texttt{let } \{x = e'\} \texttt{ in } e]_q \to C[\sigma(e)]_q$$

if e' is a variable $y \neq x$, or a literal, or a constructor or non-nullary function symbol, or ($x \notin \mathcal{V}ar(e')$ and x occurs once in e), where $\sigma = \{x \mapsto e'\}$

4.4 Lifting of Local Function Declarations

After the process of simplification, the next task is the removal of local function declarations. This process is often called *lambda lifting*, and we follow the algorithm of Johnsson [Joh85] which proceeds in two subsequent steps. Firstly, every local function is extended by adding those variables as parameters that occur in any right-hand side but are not introduced in the respective equation. Simultaneously, calls to this function are extended to supply the additional arguments. For instance, the local declaration of `g` in the example

```
f x = let y free in let g z = x + y + z
                    in  g 0
```

which refers to x and y is extended to

```
f x = let y free in let g x y z = x + y + z
                    in  g x y 0
```

Formally, the following transformation rule is applied, which first computes for every local function the set of variables occurring unbound in any right-hand side, i. e., variables that are not introduced in the surrounding context of the right-hand side or in the respective left-hand side. These variables are then prepended as additional patterns to all left-hand sides of the function, and calls to this function are extended accordingly:

$$C[\texttt{let } \{\overline{f\ \overline{p_{mn}} = e_m};\ ds\} \texttt{ in } e]_q \rightarrow C[\texttt{let } \{\overline{f\ \overline{x_l}\ \overline{p_{mn}} = e'_m};\ ds'\} \texttt{ in } e']_q$$

where f is an n-ary function defined by m equations, $\{\overline{x_l}\}$ where $l > 0$ are those variables that occur unbound in $\overline{e_m}$, and $\overline{e'_m}$, ds', e' are obtained from $\overline{e'_m}$, ds and e by replacing all occurrences of f by $f\ \overline{x_l}$

After this completion process, local function declarations refer in their right-hand side only to variables bound in the respective equation, and these functions can be safely moved to the top-level. To avoid possible name clashes, they are uniquely renamed in the entire program, and now possibly empty `let` declarations are removed. For instance, the above example will result in the following program:

```
f x = let y free
      in  g x y 0
g x y z = x + y + z
```

4.5 Compilation of Pattern Matching

The final normalization step is concerned with the implementation of pattern matching. At this stage of the normalization, function declarations may still contain multiple equations as well as nested patterns, so that the order of pattern matching for a specific function is still defined by its (generalized) definitional tree. This is an unfortunate situation, since definitional trees are a rather complicated representation of pattern matching and cannot be directly expressed in the program. This final step will therefore replace the pattern matching in function declarations by means of flexible `case` expressions, and both flexible and rigid `case` expressions in the right-hand sides of equations will be transformed to contain only simple patterns, i. e., patterns consisting of either a literal or a constructor applied to variables.

This transformation is justified by the results of Hanus and Prehofer [HP99], who successfully proved the equivalence of narrowing based on `case` expressions and

narrowing based on definitional trees, and provided a transformation of a (non-generalized) definitional tree into equivalent `case` expressions. In the following, we will present an amalgamation of both the construction of generalized definitional trees [Han12] and their transformation into `case` expressions by providing a direct translation of pattern matching into `case` expressions.

The transformation is based on the following notions of *demanded* and *inductive* positions, which are used to decide whether or not an argument has to be scrutinized by a `case` expression, and which argument should be matched first in case of multiple candidates.

Definition 4.4 (Inductive and Demanded Position)**.** *Let f be an n-ary function defined by m equations, i. e., f is defined as*

$$\begin{aligned} f\ p_{11} \ldots p_{1n} &= e_1 \\ &\vdots \\ f\ p_{m1} \ldots p_{mn} &= e_m \end{aligned}$$

Then a position $j \in \{1, \ldots, n\}$ is called inductive if and only if the j-th pattern in each equation is a constructor or literal pattern. Likewise, a position $j \in \{1, \ldots, n\}$ is called demanded if and only if there exists at least one $i \in \{1, \ldots, m\}$ such that p_{ij} is a constructor or literal pattern.

Thus, every inductive position is also demanded, but not vice versa. For instance, in the definition

```
leq Z     _     = True
leq (S _) Z     = False
leq (S n) (S m) = leq n m
```

both positions 1 and 2 are demanded, but only position 1 is inductive. Informally, an inductive position identifies an argument that has to be matched for any equation to become applicable, while a demanded position identifies an argument that has to be matched only to apply certain equations.

4.5.1 Flexible Pattern Matching

Flexible pattern matching, which is the pattern matching semantics of both function declarations and `fcase` expressions, specifies that the order of equations is irrelevant, and that in case of overlapping equations all matching equations are tried non-deterministically. Furthermore, the order in which the arguments are matched is not determined by their position (e. g., from left to right), but by their classification as demanded or inductive arguments. That is, first the inductive arguments are matched to reduce the number of non-deterministic alternatives, before the demanded arguments are matched thereafter. Hence, in the function definition

```
f True True  = 0
f _    False = 1
```

pattern matching will start with the second argument. This ensures the completeness of the needed narrowing strategy based on definitional trees [AEH00], and consequently of flexible pattern matching. For instance, pattern matching succeeds for the expression `f failed False` whereas is would fail with a left-to-right order.

The compilation of flexible pattern matching is defined by means of the operation flexMatch. Using this operation, function declarations are transformed by

$$\begin{array}{c} f\ p_{11}\ \ldots\ p_{1n} = e_1 \\ \vdots \\ f\ p_{m1}\ \ldots\ p_{mn} = e_m \end{array} \quad \to \quad f\ x_1 \ldots x_n = \mathsf{flexMatch} \left(\begin{array}{c} \overline{x_n}, \\ \end{array} \begin{array}{c} p_{11}, \ldots, p_{1n} \to e_1 \\ \vdots \\ p_{m1}, \ldots, p_{mn} \to e_m \end{array} \right)$$

and flexible `case` expressions by

$$\begin{array}{l} \texttt{fcase}\ e\ \texttt{of} \\ \quad p_1\ \texttt{->}\ e_1 \\ \quad\ \vdots \\ \quad p_m\ \texttt{->}\ e_m \end{array} \quad \to \quad \texttt{let}\ x\ \texttt{=}\ e\ \texttt{in}\ \mathsf{flexMatch} \left(\begin{array}{c} x, \\ \end{array} \begin{array}{c} p_1 \to e_1 \\ \vdots \\ p_m \to e_m \end{array} \right)$$

In the call of flexMatch, $\overline{x_n}$ denotes a sequence of fresh variables, and it is an invariant of the operation flexMatch that the length of the matched variables $\overline{x_n}$ and the length of the patterns coincide. The result will be an expression that matches the variables $\overline{x_n}$ in the order specified by the pattern sequences. Informally, the operation flexMatch proceeds as follows:

1. If the sequence of variables is empty, the expressions of the alternatives are combined using the non-deterministic choice operator "`?`".

2. Otherwise, as-patterns of the form x`@`p in the pattern sequences are replaced by their subjacent pattern p, and their variable x is replaced with the matched variable in the alternative's expression.

3. Afterwards, the leftmost *inductive* position is determined. If such a position exists, then a flexible `case` expression matching this position is constructed, and the nested and remaining patterns are recursively matched.

4. If there is no inductive position, the leftmost *demanded* position is determined. If such a position exists, then the variable patterns and constructor patterns are separated, both groups are recursively matched, and afterwards combined to a non-deterministic expression.

5. If there exists no demanded position, then all pattern sequences must consist of variable patterns only. In this case, the variables in the pattern sequences are renamed to the matched variables $\overline{x_n}$ and the alternatives are combined by "`?`".

Note that the flexible pattern matching proceeds uniformly for constructor and literal patterns, and we will refer to both kinds as constructor patterns in the following.

Empty Variable Sequence

If the variable sequence is empty, we distinguish whether or not the list of alternatives is also empty. If this is the case, then pattern matching will fail, otherwise the alternatives are combined using the non-determinism choice operator "`?`". Note that we assume "`?`" to be a right-associative primitive operator predefined in the language implementation.

$$\mathsf{flexMatch}(\varepsilon, \overline{(\varepsilon, e_m)}) = \begin{cases} \texttt{failed} & \text{if } m = 0 \\ e_1 \texttt{ ? } e_2 \ldots \texttt{ ? } e_m & \text{if } m > 0 \end{cases}$$

For instance, the non-deterministic operation `coin` defined as

```
coin = 0
coin = 1
```

is transformed into `coin = 0 ? 1`.

As-Patterns

If any of the pattern sequences contains at position j an as-pattern of the form $y@p'$, the variable y is removed from the pattern and replaced by the matched variable x_j in the alternative's expression. Note that this step does not remove as-patterns below constructors, which will instead be handled by recursive invocations of flexMatch.

$$\mathsf{flexMatch}\begin{pmatrix} \overline{p_{1n}} \to e_1 \\ \vdots \\ \overline{x_n},\ \ \overline{p_{in}} \to e_i \\ \vdots \\ \overline{p_{mn}} \to e_m \end{pmatrix} = \mathsf{flexMatch}\begin{pmatrix} \overline{p_{1n}} \to e_1 \\ \vdots \\ \overline{x_n},\ \ \overline{p_{in}}[p']_j \to \sigma(e_i) \\ \vdots \\ \overline{p_{mn}} \to e_m \end{pmatrix}$$

if $\exists i \in \{1, \ldots, m\}$, $\exists j \in \{1, \ldots, n\}$ such that $p_{ij} = y@p'$, where $\sigma = \{y \mapsto x_j\}$

Inductive Patterns

After the removal of as-patterns, the algorithm determines the leftmost *inductive* pattern position, i. e., the leftmost position j such that all patterns $p_{1j}, \ldots, p_{mj}$ are constructor patterns. If such a position exists, then the arguments of flexMatch must be of the following structure, where $C_1, \ldots, C_m$ are (not necessarily distinct) constructor symbols:

$$\mathsf{flexMatch}\begin{pmatrix} p_{11}, \ldots, p_{1,j-1} & , C_1\ \overline{q_{1l_1}} & , p_{1,j+1}, \ldots, p_{1n} \to e_1 \\ \overline{x_n}, & \vdots & \\ p_{m1}, \ldots, p_{m,j-1} & , C_m\ \overline{q_{ml_m}} & , p_{m,j+1}, \ldots, p_{mn} \to e_m \end{pmatrix}$$

Then all alternatives are grouped according to the constructors at the inductive position j. Although the order of alternatives with the same constructor is not relevant, we assume a stable sorting so these alternatives are kept in their relative order. Hence, after grouping the arguments are of the structure

$$\mathsf{flexMatch}\begin{pmatrix} p'_{11}, \ldots p'_{1,j-1} & , C^1\ \overline{q'_{1l'_1}} & , p'_{1,j+1}, \ldots, p'_{1n} \to e'_1 \\ \overline{x_n}, & \vdots & \\ p'_{m1}, \ldots p'_{m,j-1} & , C^k\ \overline{q'_{ml'_m}} & , p'_{m,j+1}, \ldots, p'_{mn} \to e'_m \end{pmatrix}$$

where $\{C^1, \ldots, C^k\}$ is the set of different constructors that appear in $C_1, \ldots, C_m$, but not necessarily the set of all constructors of the type of x_j. These constructors are then matched by a flexible `case` expression, where the patterns below the constructors and the remaining patterns are recursively matched using flexMatch:

$$\begin{array}{l} \texttt{fcase}\ x_j\ \texttt{of} \\ \quad C^1\ \overline{x_{1o_1}}\ \texttt{->}\ \mathsf{flexMatch}\begin{pmatrix} xs_1, & ps_1 \to e'_1 \\ & \vdots \end{pmatrix} \\ \qquad\qquad \vdots \\ \quad C^k\ \overline{x_{ko_k}}\ \texttt{->}\ \mathsf{flexMatch}\begin{pmatrix} xs_k, & \vdots \\ & ps_m \to e'_m \end{pmatrix} \end{array}$$

where each constructor C^k has arity o_k and all x_{ij} are fresh variables. The variable sequences $xs_1, \ldots, xs_k$ are constructed by replacing the variable x_j of the inductive position with the sequence of fresh variables $\overline{x_{ko_k}}$, thus

$$xs_{k'} = (x_1, \ldots, x_{j-1}) \cdot \overline{x_{k'o_{k'}}} \cdot (x_{j+1}, \ldots, x_n) \quad \text{for } k' \in \{1, \ldots, k\}\ .$$

Likewise, the new pattern sequences $ps_1, \ldots, ps_m$ are obtained by replacing each pattern p'_{ij} with the subpatterns $\overline{q'_{il'_i}}$:

$$ps_i = (p'_{i1}, \ldots, p'_{i,j-1}) \cdot \overline{q'_{il'_i}} \cdot (p'_{i,j+1}, \ldots, p'_{in}) \quad \text{for } i \in \{1, \ldots, m\}\ .$$

Finally, the alternatives $ps_i \to e'_i$ are distributed to the calls of flexMatch in such a way that every $ps_i \to e'_i$ occurs in the call right of $C^{k'}$ if and only if p'_{ij} is rooted by $C^{k'}$.

Demanded Patterns

If there does not exist an inductive pattern position, the leftmost *demanded* position is determined, i. e., a position j such that at least one of the patterns at position j is a constructor pattern. If such a position exists, then the alternatives are separated into two groups, where the first group contains those alternatives with a constructor pattern at position j, while the second group contains the remaining alternatives.

Inside each group, the relative order of the alternatives is kept. Thus, after the grouping the argument structure is as follows:

$$\mathsf{flexMatch}\left(\overline{x_n}, \begin{array}{llll} p_{11} & ,\ldots p_{1,j-1} & ,C_1\ \overline{q_{1l_1}}\ , p_{1,j+1} & ,\ldots, p_{1n} \to e_1 \\ & & \vdots & \\ p_{i1} & ,\ldots p_{i,j-1} & ,C_i\ \overline{q_{il_i}}\ , p_{i,j+1} & ,\ldots, p_{in} \to e_i \\ p_{i+1,1}, \ldots p_{i+1,j-1} & & ,v_{i+1} & , p_{i+1,j+1}, \ldots, p_{i+1,n} \to e_{i+1} \\ & & \vdots & \\ p_{m1} & ,\ldots p_{m,j-1} & ,v_m & , p_{m,j+1} ,\ldots, p_{mn} \to e_m \end{array}\right)$$

Then both groups are recursively transformed by flexMatch to yield

$$e'_1 = \mathsf{flexMatch}\left(\overline{x_n}, \begin{array}{llll} p_{11}, \ldots p_{1,j-1} & ,C_1\ \overline{q_{1l_1}}\ , p_{1,j+1}, \ldots, p_{1n} \to e_1 \\ & \vdots & \\ p_{i1}, \ldots p_{i,j-1} & ,C_i\ \overline{q_{il_i}}\ , p_{i,j+1}, \ldots, p_{in} \to e_i \end{array}\right)$$

for those alternatives where the pattern at the demanded position is rooted by a constructor, and

$$e'_2 = \mathsf{flexMatch}\left(\overline{x_n}, \begin{array}{llll} p_{i+1,1}, \ldots, p_{i+1,j-1} & ,v_{i+1} & , p_{i+1,j+1}, \ldots, p_{i+1,n} \to e_{i+1} \\ & \vdots & \\ p_{m1} & ,\ldots, p_{m,j-1} & ,v_m\ , p_{m,j+1} & ,\ldots, p_{mn} \to e_m \end{array}\right)$$

for the remaining ones. Both expressions are then combined using the choice operator "?" to obtain the final result expression e'_1 ? e'_2.

Variable Patterns

Finally, if there does not exist a demanded position, all patterns in the pattern sequences must be variable patterns. In this case, the expressions of all alternatives can be combined by means of the non-deterministic choice operator. However, the variables in the patterns have to be related to the sequence of matched variables $\overline{x_n}$ first. Therefore, the pattern variables are subsequently replaced by the matched variables:

$$\mathsf{flexMatch}(x \cdot \overline{x_n}, \overline{(v_m \cdot \overline{v_{mn}}, e_m)}) = \mathsf{flexMatch}(\overline{x_n}, \overline{(\overline{v_{mn}}, \sigma_m(e_m))}) \quad \text{where } \sigma_i = \{v_i \mapsto x\}$$

After the subsequent replacement of all variables, the alternatives' expressions will then be combined by the rule for an empty variable sequence.

4.5.2 Rigid Pattern Matching

In contrast to the flexible pattern matching, the rigid pattern matching considers the different equations in sequential order, so that the first matching alternative determines the result. Inside a single alternative, the pattern matching proceeds from left to right. The transformation of rigid `case` expressions is performed by the operation rigidMatch, which is a variant of flexMatch applied for rigid `case` expressions as

$$\texttt{case } e \texttt{ of} \begin{array}{l} \quad p_1 \texttt{ -> } e_1 \\ \quad \vdots \\ \quad p_m \texttt{ -> } e_m \end{array} \quad \rightarrow \quad \texttt{let } x = e \texttt{ in } \mathsf{rigidMatch} \begin{pmatrix} & p_1 \rightarrow e_1 \\ x, & \vdots \\ & p_m \rightarrow e_m \end{pmatrix}$$

where x is a fresh variable. Since the length of the sequences may be extended if nested patterns are matched, rigidMatch generally expects a sequence $\overline{x_n}$ of variables to be matched and a sequence of alternatives $\overline{\overline{p_{mn}} \rightarrow e_m}$. Informally, the operation rigidMatch proceeds as follows:

1. If the sequence of matched variables is empty, the expression of the *first* alternative is chosen as the result expression. If there are no alternatives, then pattern matching fails.

2. Otherwise, the as-patterns are removed just like for flexible pattern matching, and we refer the reader to Section 4.5.1 for a detailed definition.

3. Afterwards, the leftmost *demanded* position in the *first* pattern sequence is determined. If such a position exists, then a rigid `case` expression matching this position is constructed, and the nested and remaining patterns are recursively matched.

4. If there does not exist a demanded position in the first pattern sequence, the sequence must consist of variable patterns only. In this case, the variables of the first alternative are substituted by the matched variables in the first alternative's expression, which then establishes the result expression.

Note that the rigid pattern matching works differently for constructor and literal patterns, and we will discuss both cases individually.

Empty Alternative or Variable Sequence

If there are no more alternatives, the default value `failed` is returned, and if the variable sequence is empty, then the expression of the first alternative is chosen as the result expression.

$$\mathsf{rigidMatch}(\overline{x_n}, \varepsilon) = \texttt{failed}$$
$$\mathsf{rigidMatch}(\varepsilon, \overline{(\varepsilon, e_m)}) = e_1 \quad \text{where } m > 0$$

Constructor Patterns

Since in rigid pattern matching the alternatives are tried subsequently in the order of their definition, the algorithm continues to determine the leftmost constructor pattern in the pattern sequence of the *first* alternative only. If there exists such a position j, a rigid `case` expression is constructed that contains an alternative for all constructors of the type of x_j. Thus, for the situation

$$\mathsf{rigidMatch}\left(\begin{array}{cccc} v_{11},\ldots,v_{1,j-1} & ,C_1\ \overline{q_{1l_1}} & ,p_{1,j+1},\ldots,p_{1n}\to e_1\\ p_{21},\ldots,p_{2,j-1} & ,p_{2j} & ,p_{2,j+1},\ldots,p_{2n}\to e_2\\ & \vdots & \\ p_{m1},\ldots,p_{m,j-1} & ,p_{mj} & ,p_{m,j+1},\ldots,p_{mn}\to e_m \end{array}\right)$$

where p_{1j} is the leftmost constructor pattern in $\overline{p_{1n}}$ and $\{C^1,\ldots,C^k\}$ is the set of constructors of the type of x_j, the following rigid `case` expression is constructed.

$$\begin{array}{l}\mathsf{case}\ x_j\ \mathsf{of}\\ \quad C^1\ \overline{x_{1o_1}}\ \texttt{->}\ \mathsf{rigidMatch}(xs_1,alts_1)\\ \qquad\qquad\vdots\\ \quad C^k\ \overline{x_{ko_k}}\ \texttt{->}\ \mathsf{rigidMatch}(xs_k,alts_k)\end{array}$$

In this expression every constructor C^k has arity o_k, and all x_{ij} are fresh variables. The variable sequences $xs_1,\ldots,xs_k$ are constructed by replacing the matched variable x_j with the sequence $\overline{x_{ko_k}}$ of fresh variables:

$$xs_{k'} = (x_1,\ldots,x_{j-1})\cdot\overline{x_{k'o_{k'}}}\cdot(x_{j+1},\ldots,x_n)\quad\text{for } k'\in\{1,\ldots,k\}.$$

The computation of the alternatives $alts_1,\ldots,alts_k$ is more complicated compared to flexible pattern matching, since for every constructor C^k, all alternatives with a *compatible* pattern at position j have to be taken into account. Hence, the alternatives are defined as

$$alts_{k'} = \left((p_{i1},\ldots,p_{i,j-1})\cdot\mathsf{pats}_{k'}(p_{ij})\cdot(p_{i,j+1},\ldots,p_{in})\to e_i \;\middle|\; \begin{array}{l} i\in\{1,\ldots,m\}\wedge\\ \mathsf{compat}(p_{ij},C^{k'}\ \overline{x_{k'o_{k'}}})\end{array}\right)$$

for $k'\in\{1,\ldots,k\}$, where the extraction of the subpatterns is defined as

$$\mathsf{pats}_{k'}(p) = \begin{cases}\overline{q_{o_{k'}}} & \text{if } p = C^{k'}\ \overline{q_{o_{k'}}}\\ \overline{y_{o_{k'}}} & \text{otherwise, where } \overline{y_{o_{k'}}} \text{ are fresh variables}\end{cases}$$

Literal Patterns

The above transformation is not applicable if the demanded position of the first alternative contains a literal pattern, since there generally may exist an infinite number of literals of the respective type (e. g., numbers). Therefore, literals are compared by means of the Boolean equality operation "`==`" to avoid the construction of an infinite number of alternatives in the resulting `case` expression. Thus, for the initial situation

$$\mathsf{rigidMatch}\left(\begin{array}{cccc} v_{11},\ldots,v_{1,j-1} & ,l_1 & ,p_{1,j+1},\ldots,p_{1n}\to e_1\\ p_{21},\ldots,p_{2,j-1} & ,p_{2j} & ,p_{2,j+1},\ldots,p_{2n}\to e_2\\ & \vdots & \\ p_{m1},\ldots,p_{m,j-1} & ,p_{mj} & ,p_{m,j+1},\ldots,p_{mn}\to e_m \end{array}\right)$$

where j is the leftmost demanded position and l_1 is a literal pattern, first the sequence

$l^1, \ldots, l^k$ of literals occurring at position j is computed in the order of their occurrence, whereas repeated occurrences are removed. These literals are then matched by a nested `case` expression

```
case x_j == l^1 of
  True  -> rigidMatch(xs, alts_1)
  False -> case x_j == l^2 of
    ⋮
    False -> case x_j == l^k of
      True  -> rigidMatch(xs, alts_k)
      False -> rigidMatch(xs, alts')
```

where $xs = x_1, \ldots, x_{j-1}, x_{j+1}, \ldots, x_n$, and the alternatives $alts_1, \ldots, alts_k$ are constructed like in the previous case of constructor patterns. The last sequence of alternatives $alts'$ is constructed by collecting all alternatives with a variable pattern at the demanded position j, thus

$$alts' = \left((p_{i1}, \ldots, p_{i,j-1}) \cdot (p_{i,j+1}, \ldots, p_{in}) \to e_i \;\middle|\; \begin{array}{l} i \in \{1, \ldots, m\} \text{ and} \\ p_{ij} \text{ is a variable pattern} \end{array} \right) .$$

Variable Patterns

Finally, we consider the case that the first pattern sequence consists of variable patterns only. In this case, the pattern matching succeeds for the first alternative. However, the variables of the first alternative first need to be replaced by the variables that are matched, hence

$$\mathsf{rigidMatch}\left(\overline{x_n}, \begin{array}{c} \overline{v_{1n}} \to e_1 \\ \vdots \end{array} \right) = \sigma(e_1) \quad \text{where } \sigma = \{\overline{v_{1n} \mapsto x_n}\}$$

Limitations and Alternative Approaches

The presented transformation algorithm for rigid pattern matching has the advantage that each argument is matched only once, i. e., there are no nested `case` expressions matching the same variable. However, the algorithm may lead to code duplication in case of non-trivial overlapping, i. e., whenever constructor and variable patterns occur at the demanded position. For instance, consider the data type declaration

```
data T = C1 | C2 | C3
```

and the following function containing a rigid `case` expression

```
f x = case x of
  C1 -> C1
  y  -> large
```

where `large` denotes a syntactically large expression. Using the presented algorithm, the transformed result of `f` would be

```
f x = case x of
  C1 -> C1
  C2 -> large
  C3 -> large
```

As the number of occurrences of `large` corresponds to the number of constructors for which the default alternative applies, this may in general lead to a serious growth of the code size. Even worse, this effect may cause an exponential growth in case of nested `case` expressions. A possible optimization is to share the right-hand side of the default alternative using a fresh variable, so that the result would become

```
f x = let y = large in case x of
                        C1 -> C1
                        C2 -> y
                        C3 -> y
```

where only the variable `y` is duplicated. Since the alternatives containing `y` in their right-hand side are mutually exclusive, the sharing of `y` does not affect the semantics.

Furthermore, there also exist alternative algorithms for the compilation of rigid pattern matching. For instance, Wadler [Wad87] presents an algorithm that handles the mixture of constructor and variable patterns by grouping them into either constructor or variables patterns, and using an additional default parameter in the algorithm as the fall-back alternative. While this approach can avoid some code duplication, it may also lead to a repeated matching of the same variable. For instance, if we for now assume that rigid pattern matching would also be applied for function declarations, then this algorithm would transform the definition

```
or True  y     = True
or x     True  = True
or False False = False
```

to the result

```
or x y = case x of
  True  -> True
  False -> case y of
    True  -> True
    False -> case x of
      True  -> failed
      False -> case y of
        True  -> failed
        False -> False
```

where both `x` and `y` are scrutinized twice. Furthermore, the algorithm may still lead to some code duplication, and produces the same result for the previous example.

Generally, the problem of code duplication is caused by the combination of the sequential pattern matching semantics and non-uniform patterns, i. e., patterns where variables and constructors occur at the same position, which makes it is impossible

to completely avoid code duplication in all cases. Thus, we think that the presented algorithm in combination with the sketched optimization is a reasonable approach.

4.6 Summary

In this chapter we introduced the normalization of Curry programs into programs relying only on a small subset of constructs, which thus builds the *kernel* language of Curry. This involved the removal of shorthand expressions (syntactic sugar), the removal of local declarations up to simple variable bindings, and finally the implementation of pattern matching by a compilation into `case` expressions. Most parts of these transformations are also included in the official language report of Curry [Han12], with the exception of more recent language extensions such as mutually recursive `let` expressions, functional patterns, or `newtype` constructors. Furthermore, the transformations presented therein are usually considered in isolation, and thus the intricacies of their combination, such as the combination of non-linear and functional patterns, are not covered.

Chapter 5

Operational Semantics of FlatCurry

Operational reasoning is a tremendous waste of mental effort.

Edsger Dijkstra

In the previous chapter, we presented the normalization process applied to Curry programs to reduce their syntactic complexity, and we finally arrived at a small subset of syntactic constructs sufficient to express the full language of Curry. In the following, we will refer to this subset as FlatCurry. This chapter is concerned with the formal definition of FlatCurry, and the presentation of its operational semantics. Furthermore, we will extend this semantics to cover additional language constructs such as strict unification and functional patterns.

5.1 Programs and Expressions

The result of the normalization process is a *well-typed* program including type declarations, type signatures and typed expressions. However, for the remainder of the thesis this type information is no longer necessary, and we will therefore omit it in the following and consider *untyped* programs instead. The structure of untyped FlatCurry programs is depicted in Figure 5.1.

An untyped FlatCurry *program* P consists of a sequence of function definitions $\overline{D_m}$ such that each function f is defined by a single rule $f(\overline{x_n}) = e$ with a linear left-hand side, i.e., the variables $\overline{x_n}$ must be pairwise different. The right-hand side e of a function definition is an expression composed of variables (x, y, z, ...), constructor calls (c, d, c_1, c_2, ...), and function calls (f, g, h, ...). We consider $\mathcal{V}$ to be the (countably infinite) set of variables, and $\mathcal{C}$ and $\mathcal{F}$ to be the disjoint sets of defined constructor and function symbols, respectively. When we make no distinction between function or constructor symbols, we will use $\phi, \psi, \ldots \in \mathcal{C} \cup \mathcal{F}$ to denote either of them. Furthermore, by $\mathcal{P} \subseteq \mathcal{F}$ we refer to the set of primitive function symbols.

By $\mathcal{F}^{(n)}$ and $\mathcal{C}^{(n)}$ we denote the set of n-ary function and constructor symbols, and $n = \mathsf{arity}(\phi)$ denotes the *arity* of ϕ if and only if $\phi \in \mathcal{F}^{(n)} \cup \mathcal{C}^{(n)}$. A function (constructor) application to a sequence of argument expressions is denoted by $\phi(\overline{e_k})$. When the number k of arguments supplied to a constructor (function) call $\phi(\overline{e_k})$ is smaller than the arity n of ϕ, we say that $\phi(\overline{e_k})$ is a *partial* constructor (function) call,

P	::=	$\overline{D_m}$	(program)
D	::=	$f(\overline{x_n}) = e$	(defined function, $f \in \mathcal{F}$)
e	::=	x	(variable, $x \in \mathcal{V}$)
	|	$c(\overline{e_k})$	(constructor call, $c \in \mathcal{C}$)
	|	$f(\overline{e_k})$	(function call, $f \in \mathcal{F}$)
	|	`let {`$\overline{x_k = e_k}$`} in` e	(recursive let bindings)
	|	`let` $\overline{x_k}$ `free in` e	(free variables)
	|	e_1 `?` e_2	(disjunction)
	|	`case` e `of {` $\overline{p_k \to e_k}$ `}`	(case expression)
p	::=	$c(\overline{x_n})$	(constructor pattern)

Figure 5.1. The Untyped FlatCurry Representation of Programs

and we assume that literals occurring in the source program, such as numbers (`0`, `3.14`, ...) or characters (`'a'`, `'b'`, ...), are represented by nullary constructors.

Additionally, we allow local (mutually recursive) bindings of variables to expressions, the introduction of free (logic) variables, disjunctions to represent overlapping left-hand sides in the source language, and pattern matching using `case` expressions. The patterns $\overline{p_k}$ in `case` expressions may only consist of constructors applied to variables, and they are required to contain pairwise different constructors. In analogy to expressions, we assume that literal patterns are represented as nullary constructor patterns. By *Exp*, we denote the set of expressions that are constructed according to the rules in Figure 5.1 and the restrictions mentioned above.

Note that in FlatCurry, we do no longer distinguish between *flexible* and *rigid* `case` expressions, but only consider flexible ones. Because the patterns of a `case` expression do not overlap by definition, flexible and rigid `case` expressions only differ in their semantics if the expression to be scrutinized evaluates to a logic variable. Furthermore, the residuation mechanism of rigid `case` expressions can be simulated by a flexible `case` expression in conjunction with the primitive operator `ensureNotFree`, and the distinction is of no further relevance for partial evaluation [AHV02], so that we omit rigid `case` expressions for the sake of simplicity.

We call an expression that is either a variable, a (partial) constructor call or a partial function call a *head normal form*, and an expression is said to be *completely evaluated* if it is a logic variable, a partial (constructor or function) call applied to arbitrary expressions, or a constructor call applied to completely evaluated expressions.

For constants, i. e., symbols with an arity of zero, we will often omit the argument parentheses and just write, for instance, `[]` instead of `[]()`. Furthermore, we may write well-known binary functions and constructors in infix notation if appropriate, such as `1 + 2` or `1 : []`. To support the visual distinction of Curry and FlatCurry programs, we decorate entire listings (but not fragments) of FlatCurry programs with a gray background in the following.

Example 5.1 (Untyped FlatCurry). *The Boolean negation function* `not` *is represented as*

```
not(x) = case x of { True → False ; False → True }
```

The list concatenation `++` *may be denoted using infix application:*

```
xs ++ ys = case xs of { [] → ys ; z:zs → z : (zs ++ ys) }
```

Note that untyped FlatCurry is a *first-order* language, i. e., it does only provide the application of defined functions or constructors to expressions, but not the application of one expression to another. This restriction is justified by the observation that there only exists a finite number of functions and constructors for each program and no new functions can be generated at run time, so that every expression which is applied to an argument must evaluate to a partial call of one of these functions or constructors. Therefore, an application can be replaced by a call to a primitive operation `apply` that dispatches the application of a partial call to an additional argument. This process of *defunctionalization* goes back to Reynolds [Rey72], and we assume the representation of higher-order application by means of the primitive operation `apply` in the following.

Unlike in the context of term rewriting, expressions may not only refer to variables defined in the left-hand side of a function definition, but also to variables that are *locally* introduced in `let` expressions or in the patterns of `case` expressions. More precisely, `let` expressions of the form `let` $\{\overline{x_n = e_n}\}$ `in` e or `let` $\overline{x_n}$ `free in` e and `case` alternatives of the form $c(\overline{x_n}) \to e$ are said to *introduce* or *bind* the variables $\overline{x_n}$ in the respective subordinate expressions $\overline{e_n}$ and e. An expression e is said to have *unique variables* if and only if every variable in e is introduced at most once in e. If we refer to the variables *occurring* in an expression in the following, we will only consider the usage but not the introduction of variables, and therefore define the (multi)set of variables occurring in an expression as follows.

Definition 5.2 (Variables in Expressions/Patterns). *The set of variables occurring in an expression e, denoted by $\mathcal{V}ar(e)$, is inductively defined as*

$$\begin{aligned}
\mathcal{V}ar(x) &= \{x\} \\
\mathcal{V}ar(\phi(\overline{e_k})) &= \bigcup_{i=1}^{k} \mathcal{V}ar(e_i) \\
\mathcal{V}ar(\texttt{let } \{\overline{x_k = e_k}\} \texttt{ in } e) &= \bigcup_{i=1}^{k} \mathcal{V}ar(e_i) \cup \mathcal{V}ar(e) \\
\mathcal{V}ar(\texttt{let } \overline{x_k} \texttt{ free in } e) &= \mathcal{V}ar(e) \\
\mathcal{V}ar(e_1 \texttt{ ? } e_2) &= \mathcal{V}ar(e_1) \cup \mathcal{V}ar(e_2) \\
\mathcal{V}ar(\texttt{case } e \texttt{ of } \{ \overline{p_k \to e_k} \}) &= \mathcal{V}ar(e) \cup \bigcup_{i=1}^{k} \mathcal{V}ar(e_i)
\end{aligned}$$

The multiset of variables in e, denoted by $\mathcal{V}ar_M(e)$, is defined accordingly.

We call the occurrence of a variable x in an expression *bound* if it was introduced by a surrounding binding, otherwise the variable is called *unbound* in this expression.[1]

[1] In the context of functional programs, unbound variables are often referred to as *free* variables. However, this term is also used for logic variables, so that we choose a distinct term to avoid confusion.

The set of variables occurring bound in an expression e will be denoted by $\mathcal{BV}(e)$, and the set of unbound variables by $\mathcal{UV}(e)$. In addition, for a set E of expressions, $\mathcal{BV}(E)$ and $\mathcal{UV}(E)$ are defined as the union of the results of $\mathcal{BV}(e)$ (resp. $\mathcal{UV}(e)$) for $e \in E$. Furthermore, an expression e is said to be *linear* if it does not contain multiple occurrences of an unbound variable, and a variable x is *fresh* with respect to an expression e if $x \notin \mathcal{V}ar(e)$ and x is not introduced in e.

Example 5.3 (Unique, Bound and Unbound Variables)**.** *Consider the two expressions*

```
e1 = let { x = 1 in (let x free in x, x, y) }
e2 = let { x = 1; y = 2 } in x : y : []
```

Then e_1 does not have unique variables since x is introduced twice, $\mathcal{BV}(e_1) = \{\mathtt{x}\}$ and $\mathcal{UV}(e_1) = \{\mathtt{y}\}$, while e_2 has unique variables, $\mathcal{BV}(e_2) = \{\mathtt{x},\mathtt{y}\}$ and $\mathcal{UV}(e_2) = \varnothing$.

To avoid notational burdens in the remainder of the thesis, we establish the following variable convention, which can always be accomplished by a renaming of bound variables to fresh ones (see [Bar84] for details). Furthermore, we will also consider two expressions as equivalent if they can be transformed into each other by a renaming of their bound variables, which is referred to as α-equivalence in the context of the λ-calculus [Bar84].

Convention 5.4 (Variable Convention for Expressions [Bar84])**.** *For all expressions $e_1, \ldots, e_n$ that occur in a mathematical context (definition, theorem, ...), all variables bound in these expressions are chosen to be unique and different from the unbound variables.*

Based on the variable convention, we can provide the following definition of substitutions on FlatCurry expressions.

Definition 5.5 (Substitution on FlatCurry Expression)**.** *A substitution on FlatCurry expressions is a mapping $\sigma : \mathcal{V} \to Exp$ from a set of variables to expressions such that only for a finite set of variables, we have $\sigma(x) \neq x$. The extension σ^* of a substitution σ to a mapping from expressions to expressions is inductively defined as follows:*

$$\begin{aligned}
\sigma^*(x) &= \sigma(x) \\
\sigma^*(\phi(\overline{e_k})) &= \phi(\overline{\sigma^*(e_k)}) \\
\sigma^*(\texttt{let } \{\overline{x_k = e_k}\} \texttt{ in } e) &= \texttt{let } \{\overline{x_k = \sigma^*(e_k)}\} \texttt{ in } \sigma^*(e) \\
\sigma^*(\texttt{let } \overline{x_k} \texttt{ free in } e) &= \texttt{let } \overline{x_k} \texttt{ free in } \sigma^*(e) \\
\sigma^*(e_1 \texttt{ ? } e_2) &= \sigma^*(e_1) \texttt{ ? } \sigma^*(e_2) \\
\sigma^*(\texttt{case } e \texttt{ of } \{ \overline{p_k \to e_k} \}) &= \texttt{case } \sigma^*(e) \texttt{ of } \{ \overline{p_k \to \sigma^*(e_k)} \}
\end{aligned}$$

Note that we can assume the bound variables to not occur in $\mathcal{D}om(\sigma) \cup \mathcal{R}an(\sigma)$ due to the variable convention. Furthermore, we will usually write σ instead of σ^*.

In analogy to substitutions on terms, an injective variable substitution on expressions is called a *variable renaming*. While two expressions that differ only in their *bound*

variables are considered equivalent, expressions that differ only in their *unbound* variables are considered as *variants*.

Definition 5.6 (Variant of an Expression)**.** *An expression e_1 is a variant of another expression e_2, denoted by $e_1 \equiv e_2$, if there exists a variable renaming σ such that $\sigma(e_1) = e_2$.*

For function calls, we also consider non-injective variable substitutions in order to be able to match a (not necessarily linear) call to a function with its definition.

Definition 5.7 (Variable Instance of a Rule)**.** *A rule $f(\overline{y_n}) = e'$ is said to be a* variable instance *of a rule $f(\overline{x_n}) = e$ if there exists a (not necessarily injective) variable substitution σ such that $\sigma(f(\overline{x_n})) = f(\overline{y_n})$ and $\sigma(e) = e'$.*

In the following, we will only consider finite and well-formed FlatCurry programs, i. e., programs that obey the variable convention and for which the right-hand sides of the function declarations only contain unbound variables defined in the corresponding left-hand side.

Definition 5.8 (Well-Formed Program)**.** *A program P is said to be well-formed if and only if every $D \in P$ defines a different function f and for every declaration D of the form $f\ (\overline{x_n})\ =\ e$ it holds that $f(\overline{x_n})$ is linear, e obeys the variable convention, and $\mathcal{UV}(e) \subseteq \{\overline{x_n}\}$.*

5.2 Operational Semantics

There exist different approaches in the literature to formalize the semantics of functional logic languages with a call-time choice semantics. For instance, the CRWL calculus (Constructor-based conditional ReWriting Logic) [GHL+99] provides a strategy-independent foundation for declarative programs with non-strict and non-deterministic operations obeying call-time choice. However, the high-level abstraction of the CRWL calculus does not respect single evaluation steps, which makes it difficult to use in foresight of the later development of a partial evaluator. This problem of high-level abstraction has been partially addressed by the proposals of *let-rewriting* [LRS07a] and *let-narrowing* [LRS07b], which do consider single evaluation steps. However, they lack a certain (fixed) evaluation strategy, so that their application is highly non-deterministic. Therefore, we will base our work on the operational semantics for FlatCurry of Albert et al. [AHH+05]. Their work proposes a big-step semantics in the style of Launchbury [Lau93] and a small-step semantics in the style of Sestoft [Ses97], which are shown to be equivalent. While the former essentially provides an evaluation rule for each language construct, the latter defines the evaluation process by means of a state transition system. The big-step semantics is often referred to as the *natural semantics* [Kah87] since it assigns a meaning to each language construct, and we will base our work on a variant of this semantics to obtain a more compact notation compared to the small-step semantics.

The semantics as proposed by Albert et al. [AHH+05] is defined for a subset of FlatCurry expressions, the set of *flat expressions*, and uses the auxiliary structure of *heaps* to record the results of previous evaluation steps. An expression is then evaluated in the context of a heap containing variable bindings, and the evaluation steps to be applied are determined by the rules of the operational semantics.

5.2.1 Flat Expressions

Flat expressions form a subset of FlatCurry expressions where the application of function or constructor symbols is constrained to variables only, and we denote the set of flat expressions by *FlatExp*. For any FlatCurry expression e, its corresponding flat expression can be obtained by the following transformation.

Definition 5.9 (Flattening)**.** *The computation of the flat form (or flattening) of an expression e is inductively defined as*

$$\begin{aligned}
\mathsf{flat}(x) &= x \\
\mathsf{flat}(\phi(\overline{e_k})) &= \texttt{let } \{\overline{y_l = \mathsf{flat}(e'_l)}\} \texttt{ in } \phi(\overline{x_k}) \\
&\quad \textit{where } (\overline{y_l}, \overline{e'_l}, \overline{x_k}) = \mathsf{splitArgs}(\overline{e_k}) \\
\mathsf{flat}(\texttt{let } \{\overline{x_k = e_k}\} \texttt{ in } e) &= \texttt{let } \{\overline{x_k = \mathsf{flat}(e_k)}\} \texttt{ in } \mathsf{flat}(e) \\
\mathsf{flat}(\texttt{let } \overline{x_k} \texttt{ free in } e) &= \texttt{let } \overline{x_k} \texttt{ free in } \mathsf{flat}(e) \\
\mathsf{flat}(e_1 \texttt{ ? } e_2) &= \mathsf{flat}(e_1) \texttt{ ? } \mathsf{flat}(e_2) \\
\mathsf{flat}(\texttt{case } e \texttt{ of } \{ \overline{p_k \to e_k} \}) &= \texttt{case } \mathsf{flat}(e) \texttt{ of } \{ \overline{p_k \to \mathsf{flat}(e_k)} \}
\end{aligned}$$

where the splitting of arguments into variables and non-variable expressions is defined as

$$\begin{aligned}
\mathsf{splitArgs}(\varepsilon) &= (\varepsilon, \varepsilon, \varepsilon) \\
\mathsf{splitArgs}(e \cdot \overline{e_n}) &= \begin{cases} (\overline{y_m},\ \overline{e'_m},\ e \cdot \overline{x_n}) & \textit{if } e \in \mathcal{V} \\ (y \cdot \overline{y_m},\ e \cdot \overline{e'_m},\ y \cdot \overline{x_n}) & \textit{if } e \notin \mathcal{V} \end{cases} \\
&\quad \textit{where } y \textit{ fresh and } (\overline{y_m}, \overline{e'_m}, \overline{x_n}) = \mathsf{splitArgs}(\overline{e_n})
\end{aligned}$$

We furthermore establish the convention that the construct `let {} in` e is a complex notation for the expression e, so that no empty `let`-bindings are constructed.

Example 5.10 (Flattening)**.** *For the expression* `and(True, False)`*, flattening yields*

$$\mathsf{flat}(\texttt{and(True, False)}) = \texttt{let } \{x_1 = \texttt{True};\ x_2 = \texttt{False}\} \texttt{ in } \mathsf{and}(x_1, x_2)$$

For functions or constructors applied to variables, flattening does not change the expression:

$$\mathsf{flat}(\texttt{and(x,y)}) = \texttt{and(x,y)}$$

Note that flattening may also introduce nested `let`*-expressions:*

$$\begin{aligned}
&\mathsf{flat}(\texttt{(x : xs) ++ (1 : [])}) \\
= \ &\texttt{let } \{x_1 = \texttt{x : xs};\ x_2 = \texttt{let } \{x_3 = \texttt{1};\ x_4 = \texttt{[]}\} \texttt{ in } x_3 : x_4\} \texttt{ in } x_1 \texttt{ ++ } x_2
\end{aligned}$$

The virtue of flattening becomes apparent when a call to a function should be replaced by the right-hand side of its definition. For instance, consider the definition

```
double(x) = x + x
```

and the function call `double(3 * 7)`. If we applied the substitution $\{\texttt{x} \mapsto \texttt{3 * 7}\}$ to the right-hand side of `double`, we would obtain the expression `(3 * 7) + (3 * 7)` with a duplicated computation of `(3 * 7)`. In contrast, in the flattened call `let x = (3 * 7) in double(x)` the sharing of the argument is made explicit, and we would obtain the new expression `let x = (3 * 7) in x + x` if we replaced the function call. Thus, flattening can be used to support the implementation of a call-by-need evaluation and a call-time-choice semantics.

5.2.2 Heaps and Configurations

A *heap* is a finite subset of the set $\mathcal{V} \times (\{\text{free}, \blacksquare\} \uplus \mathit{FlatExp})$ such that each variable $x \in \mathcal{V}$ appears at most once as the first component of a pair (x, b) within the set. In this definition, the special symbol "free" is used to identify a free (logic) variable, and the symbol "$\blacksquare$" identifies a *blackhole*, i. e., a non-terminating self-dependent loop [Lau93]. Both symbols are required to be distinguishable from any expression, such that no ambiguity can arise. In consequence, a heap can also be seen as a partial mapping of variables to either expressions or the two special symbols, and we adopt the usual notation of $\Gamma(x) = b$ for $(x, b) \in \Gamma$. Furthermore, we will use the notation of $\mathcal{D}om(\Gamma) := \{x \mid (x, b) \in \Gamma\}$ for the *domain* and $\mathcal{R}an(\Gamma) := \{b \mid (x, b) \in \Gamma\}$ for the *range* of a heap Γ. We will say that the variables in $\mathcal{D}om(\Gamma)$ are *bound* in the heap Γ and the *variables occurring in a heap* Γ are defined as $\mathcal{V}ar(\Gamma) := \mathcal{D}om(\Gamma) \cup \bigcup\{\mathcal{V}ar(e) \mid e \in \mathcal{R}an(\Gamma)\}$, and accordingly for the multiset of variables.

We denote heaps with Greek uppercase letters such as Γ, Δ, Θ, or Ω. The empty heap is denoted by $[]$, and $\Gamma[x \mapsto e]$ denotes a heap Γ' with $\Gamma'(x) = e$ and $\Gamma'(y) = \Gamma(y)$ for all $y \neq x$. We will use this notation as a *heap update* and a deconstruction of a heap, where $\Gamma' = \Gamma[x \mapsto e]$ implies $x \notin \mathcal{D}om(\Gamma)$ in case of a deconstruction. Furthermore, for all heaps it holds $\Gamma[x \mapsto e][x \mapsto e] = \Gamma[x \mapsto e]$. By $\Gamma' \uplus \Gamma''$, we denote a heap Γ such that $\mathcal{D}om(\Gamma') \cap \mathcal{D}om(\Gamma'') = \varnothing$ and $\Gamma = \Gamma' \cup \Gamma''$. A *well-formed heap* satisfies the property that all variables bound in $\mathcal{R}an(\Gamma)$ are different from those in $\mathcal{D}om(\Gamma)$.

A *configuration* $\Gamma : e$ is a pair of a heap Γ and a flat expression e, and a *well-formed configuration* additionally satisfies the property that Γ is well-formed and the variables bound in e are different from those bound in Γ. The *variables occurring in a configuration*, denoted by $\mathcal{V}ar(\Gamma : e)$, are those occurring in Γ or e, and we say that a variable x is fresh with respect to a configuration if it does not occur nor is bound in this configuration.

To avoid the possibility of name capture problems for configurations, we extend the variable convention on expressions to also be applicable for heaps and configurations.

Convention 5.11 (Variable Convention for Heaps and Configurations)**.** *For all heaps and configurations that occur in a mathematical context, all bound variables in these heaps and configurations are chosen to be unique and different from the unbound variables.*

This convention in particular implies that heaps and configurations which obey this convention are well-formed, and we will assume well-formedness of heaps and configurations in the remainder of this thesis.

5.2.3 Statements and Derivations

Two configurations are related in *statements* of the form $\Gamma : e \Downarrow \Delta : v$, where $\Gamma : e$ and $\Delta : v$ are called the *in-* and *out*-configurations of the statement. Such statements are interpreted in that the expression e in the context of the heap Γ evaluates to the value v and the (possibly modified) heap Δ. If the corresponding program P is relevant, we may also write $\Gamma : e \Downarrow^P \Delta : v$. We formally define the set of *values* as

$$\begin{array}{rcll} \mathit{Value} & ::= & x & (\text{logic variable}) \\ & | & c(\overline{x_n}) & (\text{constructor application}, n = \mathsf{arity}(c)) \\ & | & \phi(\overline{x_k}) & (\text{partial application}, k < \mathsf{arity}(\phi)) \end{array}$$

where c is an n-ary constructor and ϕ is a constructor or function symbol. Note that a variable is only considered as a flat value if it is bound to the symbol "free" in the corresponding heap. Consequently, we demand every expression to evaluate to either a logic variable or a constructor or partial call applied to variables, so that every value is a flat expression in head normal form. Note that this definition is in slight contrast to the definition of values in the context of term rewriting, where a value is considered to be a constructor term in normal form, which corresponds to completely evaluated expression in FlatCurry. This is caused by the fact that the operational semantics considers the evaluation of an expression only to its head normal form, while term rewriting systems usually consider normal forms.

The operational semantics of FlatCurry is provided as an inference system using *statements* as its formulas, and we refer to this semantics as $\Downarrow_0$ since we will later present modified variants. The rules of $\Downarrow_0$ are depicted in Figure 5.2, and we briefly describe them in the following. Note that $\Downarrow_0$ slightly differs from the original version proposed by Albert et al. [AHH+05], and we will explain the differences in Section 5.2.4.

(Value) A value, i. e., a constructor call, a partial application, or a free variable, is directly returned as the result without modifying the heap.

(VarExp) This rule implements the sharing of subexpressions to comply with the call-by-need and call-time-choice semantics. If a variable to be evaluated is bound to an expression, the expression is evaluated and its value is returned. In addition, the heap is updated with the value. During evaluation of the expression, the binding

(Value) $\Gamma : v \Downarrow_0 \Gamma : v$ where $v = \phi(\overline{x_k})$ with $\phi \in \mathcal{C}$ or $k < \mathsf{arity}(\phi)$, or $v \in \mathcal{V}$ with $\Gamma(v) = \mathsf{free}$

(VarExp) $$\frac{\Gamma[x \mapsto \blacksquare] : e \Downarrow_0 \Delta : v}{\Gamma[x \mapsto e] : x \Downarrow_0 \Delta[x \mapsto v] : v} \quad \text{where } e \notin \{\mathsf{free}, \blacksquare\}$$

(Fun) $$\frac{\Gamma : \mathsf{flat}(e) \Downarrow_0 \Delta : v}{\Gamma : f(\overline{x_n}) \Downarrow_0 \Delta : v} \quad \text{where } f(\overline{x_n}) = e \text{ is a variable instance of a rule in } P$$

(Let) $$\frac{\Gamma[\overline{x_k \mapsto e_k}] : e \Downarrow_0 \Delta : v}{\Gamma : \mathtt{let}\ \{\overline{x_k = e_k}\}\ \mathtt{in}\ e \Downarrow_0 \Delta : v}$$

(Or) $$\frac{\Gamma : e_i \Downarrow_0 \Delta : v}{\Gamma : e_1\ \mathtt{?}\ e_2 \Downarrow_0 \Delta : v} \quad \text{where } i \in \{1,2\}$$

(Free) $$\frac{\Gamma[\overline{x_k \mapsto \mathsf{free}}] : e \Downarrow_0 \Delta : v}{\Gamma : \mathtt{let}\ \overline{x_k}\ \mathtt{free}\ \mathtt{in}\ e \Downarrow_0 \Delta : v}$$

(Select) $$\frac{\Gamma : e \Downarrow_0 \Delta : c(\overline{x_n}) \quad \Delta : \sigma(e_i) \Downarrow_0 \Theta : v}{\Gamma : \mathtt{case}\ e\ \mathtt{of}\ \{\ \overline{p_k \to e_k}\ \} \Downarrow_0 \Theta : v} \quad \begin{array}{l}\text{where } c(\overline{x_n}) = \sigma(p_i) \\ \text{and } i \in \{1,\ldots,k\}\end{array}$$

(Guess) $$\frac{\Gamma : e \Downarrow_0 \Delta : x \quad \Delta[x \mapsto c(\overline{x_n}), \overline{x_n \mapsto \mathsf{free}}] : e_i \Downarrow_0 \Theta : v}{\Gamma : \mathtt{case}\ e\ \mathtt{of}\ \{\ \overline{p_k \to e_k}\ \} \Downarrow_0 \Theta : v}$$
where $\Delta(x) = \mathsf{free}$, $c(\overline{x_n}) = p_i$, and $i \in \{1,\ldots,k\}$

Figure 5.2. Natural Semantics for Flat Expressions

is replaced by "■", which allows the detection of blackholes and is necessary for the correctness of the semantics (see below for a more detailed explanation).

(Fun) This rule unfolds a function call, where the result is obtained by evaluation of the function's right-hand side. We assume that the program P is a global parameter of the calculus, and by $f(\overline{x_n}) = e$ we denote a variable instance of the corresponding program rule in P such that all variables bound in e are fresh with respect to the configuration $\Gamma : f(\overline{x_n})$.

(Let) The bindings of a `let` construct are added to the heap and evaluation proceeds with the subordinate expression.

(Or) This rule non-deterministically chooses one of the arguments to be further evaluated, and thus introduces non-determinism into the calculus itself.

(Free) Like variables bound to expressions, logic variables are bound in the heap and evaluation proceeds with the subjacent expression.

(Select) For `case` expressions whose scrutinized expression evaluates to a constructor-rooted value, the right-hand side of the corresponding alternative is selected and evaluated. To relate the pattern variables in the respective alternative to the variables in the value, a variable renaming σ is applied for the respective right-hand side.

(Guess) For `case` expressions whose scrutinized expression evaluates to a logic variable, one of the alternatives is non-deterministically chosen to be evaluated. The logic variable is furthermore instantiated to the corresponding pattern, where the variables occurring in the pattern are bound as logic variables.

Note that the calculus is non-deterministic in the rules (Or) and (Guess), so that a non-deterministic expression can be evaluated to multiple values. Furthermore, there may exist expressions for which no rule can be applied, and evaluation is said to fail in this case.

A proof of a statement corresponds to a *derivation* using the rules of the calculus presented in Figure 5.2. We will frequently state that with respect to a heap Γ, an expression e evaluates to the value v, and give a proof for the statement $\Gamma : e \Downarrow \Delta : v$.

Example 5.12 (Derivation)**.** *We consider the following program*

```
ones     = 1 : ones
head(xs) = case xs of { y : ys → y }
```

and the flat expression `let xs = ones in head(xs)`*. We can determine the value of the expression using the presented rules, where we obtain the derivation*

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Gamma : \texttt{a:b} \Downarrow_0 \Gamma : \texttt{a:b}}{[\texttt{xs} \mapsto \blacksquare] : \texttt{let a = 1; b = ones in a:b} \Downarrow_0 \Gamma : \texttt{a:b}}}{[\texttt{xs} \mapsto \blacksquare] : \texttt{ones} \Downarrow_0 \Gamma : \texttt{a:b}}}{[\texttt{xs} \mapsto \texttt{ones}] : \texttt{xs} \Downarrow_0 \Delta : \texttt{a:b}} \qquad \dfrac{\Delta[\texttt{a} \mapsto \blacksquare] : 1 \Downarrow_0 \Delta[\texttt{a} \mapsto \blacksquare] : 1}{\Delta : \texttt{a} \Downarrow_0 \Delta : 1}}{[\texttt{xs} \mapsto \texttt{ones}] : \texttt{case xs of \{ y:ys} \to \texttt{y \}} \Downarrow_0 \Delta : 1}}{[\texttt{xs} \mapsto \texttt{ones}] : \texttt{head(xs)} \Downarrow_0 \Delta : 1}}{[] : \texttt{let xs = ones in head(xs)} \Downarrow_0 \Delta : 1}$$

for $\Gamma = [\texttt{xs} \mapsto \blacksquare, \texttt{a} \mapsto 1, \texttt{b} \mapsto \texttt{ones}]$ *and* $\Delta = [\texttt{xs} \mapsto \texttt{a:b}, \texttt{a} \mapsto 1, \texttt{b} \mapsto \texttt{ones}]$.

In addition to the tree-like arrangement of statements in a derivation, we will sometimes refer to the *dependency sequence* of configurations, which is defined as

$$\mathsf{dep}(\Gamma : e \Downarrow \Delta : v) = \Gamma : e, \Delta : v$$
$$\mathsf{dep}\left(\dfrac{\overline{D_n}}{\Gamma : e \Downarrow \Delta : v}\right) = \Gamma : e, \overline{\mathsf{dep}(D_n)}, \Delta : v$$

5.2.4 Differences to the Original Semantics

In comparison to the operational semantics proposed by Albert et al. [AHH+05], we made some adjustments that we like to discuss in the following. For instance, the right-hand sides of functions are flattened in rule (Fun) instead of requiring a flattened program, but it is easy to see that this does not affect the set of statements that could be derived. However, there are also more differences, and not all of them are only notational. We will therefore provide a detailed explanation for changes that impede the equivalence of the semantics.

Omission of Renaming

A minor change in the calculus compared to the original version is the omission of variable renamings for the rules (Fun), (Let), (Free) and (Guess), which were incorporated in the original version to handle variable shadowing. We avoid such complications by means of the variable convention and the restriction to well-formed programs. Assumed that the rules of the semantics are applied to an initially well-formed configuration, it holds that all configurations in a proof are well-formed, so that no name clashes can occur and the renaming becomes dispensable.

Proposition 5.13 (Conservation of Well-Formedness [Bra11])**.** *Let* $\Gamma : e$ *be a well-formed configuration. Then any derivation for a statement* $\Gamma : e \Downarrow_0 \Delta : v$ *contains only well-formed configurations.*

Elimination of Rule (VarCons)

The original semantics contains an additional rule (VarCons) used for the evaluation of a variable x that is bound to a constructor-rooted value v in the heap:

$$\text{(VarCons)} \qquad \Gamma[x \mapsto v] : x \Downarrow' \Gamma[x \mapsto v] : v \quad \text{where } v \text{ is constructor-rooted}$$

In addition, the rule (VarExp) is not applied when a variable is bound to a constructor-rooted value. However, the rule (VarCons) can be seen as a shortcut for applying the two rules (VarExp) and (Value) in sequence:

$$\frac{\Gamma[x \mapsto \blacksquare] : v \Downarrow_0 \Gamma[x \mapsto \blacksquare] : v}{\Gamma[x \mapsto v] : x \Downarrow_0 \Gamma[x \mapsto v] : v}$$

Braßel [Bra11] has shown that the omission of rule (VarCons) does not affect the set of derivable statements, and we also omit this additional rule.

Blackhole Detection

In the semantics presented in Figure 5.2, the rule (VarExp) replaces the variable binding $x \mapsto e$ by the binding $x \mapsto \blacksquare$ for evaluation of the expression e. This allows the detection of blackholes (self-dependent infinite loops) [Lau93] as done in some

implementations of functional (logic) languages. For instance, an attempt to evaluate the expression "`let {x = x} in x`" results in a finitely failing derivation tree with the help of blackhole detection, whereas it would trigger the construction of an infinite derivation tree if the binding $x \mapsto x$ was kept. Note that in the work of Launchbury [Lau93], the binding $x \mapsto e$ was removed from the heap entirely, while it is replaced by the binding $x \mapsto \blacksquare$ in the semantics proposed above. We do this in foresight of later partial evaluation, but this difference is only notational.

Nevertheless, our rule is in contrast with the original rule (OrigVarExp) of Albert et al. [AHH+05], where the binding is kept in the heap (note that free variables are represented by circular bindings $x \mapsto x$ in their work):

$$\text{(OrigVarExp)} \quad \frac{\Gamma[x \mapsto e] : e \Downarrow' \Delta : v}{\Gamma[x \mapsto e] : x \Downarrow' \Delta[x \mapsto v] : v} \quad \begin{array}{l}\text{where } e \text{ is not constructor-rooted} \\ \text{and } e \neq x\end{array}$$

On the first sight, the detection of blackholes seems to be an optimization that could be omitted for deterministic programs [Lau93]. However, it is crucial in combination with non-deterministic operations in order to prevent the binding of a variable to *different* values in the *same* derivation, as has been shown by Braßel [Bra11].

Example 5.14 (Need for Blackhole Detection)**.** *Consider the expression*

```
let { x = T ? case x of { T → F } } in x
```

If we did not replace the variable binding in rule (VarExp), the following derivation would be possible for $\Gamma = [\mathtt{x} \mapsto \mathtt{T} \ \mathtt{?}\ \mathtt{case}\ \mathtt{x}\ \mathtt{of}\ \{\ \mathtt{T} \to \mathtt{F}\ \}]$*:*

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Gamma : \mathtt{T} \Downarrow' \Gamma : \mathtt{T}}{\Gamma : \mathtt{T}\ \mathtt{?}\ \mathtt{case}\ \mathtt{x}\ \mathtt{of}\ \{\ \mathtt{T} \to \mathtt{F}\ \} \Downarrow' \Gamma : \mathtt{T}}}{\Gamma : \mathtt{x} \Downarrow' [\mathtt{x} \mapsto \mathtt{T}] : \mathtt{T}} \qquad [\mathtt{x} \mapsto \mathtt{T}] : \mathtt{F} \Downarrow' [\mathtt{x} \mapsto \mathtt{T}] : \mathtt{F}}{\Gamma : \mathtt{case}\ \mathtt{x}\ \mathtt{of}\ \{\ \mathtt{T} \to \mathtt{F}\ \} \Downarrow' [\mathtt{x} \mapsto \mathtt{T}] : \mathtt{F}}}{\Gamma : \mathtt{T}\ \mathtt{?}\ \mathtt{case}\ \mathtt{x}\ \mathtt{of}\ \{\ \mathtt{T} \to \mathtt{F}\ \} \Downarrow' [\mathtt{x} \mapsto \mathtt{T}] : \mathtt{F}}}{\Gamma : \mathtt{x} \Downarrow' [\mathtt{x} \mapsto \mathtt{F}] : \mathtt{F}}}{[] : \mathtt{let}\ \{\mathtt{x} = \mathtt{T}\ \mathtt{?}\ \mathtt{case}\ \mathtt{x}\ \mathtt{of}\ \{\ \mathtt{T} \to \mathtt{F}\ \}\}\ \mathtt{in}\ \mathtt{x} \Downarrow' [\mathtt{x} \mapsto \mathtt{F}] : \mathtt{F}}$$

In this derivation, the variable `x` *is looked up twice in the heap, where at first the right non-deterministic branch is chosen and afterwards the left branch. Hence,* `x` *is bound to* `T` *as well as* `F`*, which violates the single assignment property of call-time choice. With blackhole detection, there is only one successful derivation where* `x` *is bound to* `T`*:*

$$\dfrac{\dfrac{\dfrac{[\mathtt{x} \mapsto \blacksquare] : \mathtt{T} \Downarrow_0 [\mathtt{x} \mapsto \blacksquare] : \mathtt{T}}{[\mathtt{x} \mapsto \blacksquare] : \mathtt{T}\ \mathtt{?}\ \mathtt{case}\ \mathtt{x}\ \mathtt{of}\ \{\ \mathtt{T} \to \mathtt{F}\ \} \Downarrow_0 [\mathtt{x} \mapsto \blacksquare] : \mathtt{T}}}{[\mathtt{x} \mapsto \mathtt{T}\ \mathtt{?}\ \mathtt{case}\ \mathtt{x}\ \mathtt{of}\ \{\ \mathtt{T} \to \mathtt{F}\ \}] : \mathtt{x} \Downarrow_0 [\mathtt{x} \mapsto \mathtt{T}] : \mathtt{T}}}{[] : \mathtt{let}\ \{\mathtt{x} = \mathtt{T}\ \mathtt{?}\ \mathtt{case}\ \mathtt{x}\ \mathtt{of}\ \{\ \mathtt{T} \to \mathtt{F}\ \}\}\ \mathtt{in}\ \mathtt{x} \Downarrow_0 [\mathtt{x} \mapsto \mathtt{T}] : \mathtt{T}}$$

If the other non-deterministic alternative is chosen, the derivation fails due to blackhole detection, such that the call-time-choice semantics is obeyed:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{[\mathtt{x} \mapsto \blacksquare] : \mathtt{x} \Downarrow_0 \mathit{failure}}}{[\mathtt{x} \mapsto \blacksquare] : \mathtt{case\ x\ of\ \{\ T \to F\ \}} \Downarrow_0}}{[\mathtt{x} \mapsto \blacksquare] : \mathtt{T\ ?\ case\ x\ of\ \{\ T \to F\ \}} \Downarrow_0}}{[\mathtt{x} \mapsto \mathtt{T\ ?\ case\ x\ of\ \{\ T \to F\ \}}] : \mathtt{x} \Downarrow_0}}{[] : \mathtt{let\ \{x = T\ ?\ case\ x\ of\ \{\ T \to F\ \}\}\ in\ x} \Downarrow_0}$$

We also like to state the observation that a blackhole can only be removed by rule (VarExp), so that no blackholes appear or vanish during evaluation.

Lemma 5.15. $\Gamma : e \Downarrow \Delta : v$ *implies that, for all variables* x*, we have* $\Gamma(x) = \blacksquare$ *if and only if* $\Delta(x) = \blacksquare$.

Proof. We prove both directions of the implication by contradiction. Suppose that there does exist a derivation for $\Gamma[x \mapsto \blacksquare] : e \Downarrow \Delta : v$ with $\Delta(x) \neq \blacksquare$. Then there must exist a rule that either removes or replaces a blackhole in the in-configuration. However, the only rules that change existing bindings are rules (VarExp) and (Guess), and neither of them is applicable if the variable in question is bound to $\blacksquare$. If we assume a derivation for $\Gamma : e \Downarrow \Delta[x \mapsto \blacksquare] : v$ with $\Gamma(x) \neq \blacksquare$, then there must exist a rule that binds x to $\blacksquare$ in the out-configuration. However, no such rule exists, and thus the statement must be invalid. □

Representation of Logic Variables

We identify logic variables by a binding to the special symbol "free" in the heap, while the original semantics represents logic variables by a circular binding of the form $\Gamma[x \mapsto x]$. Although this difference is mainly notational, it allows the distinction of circular `let`-bindings and the introduction of logic variables. For instance, we can consider the expression

```
let {x = x} in x
```

to denote a non-terminating computation, while the expression

```
let x free in x
```

denotes a logic variable. Due to our notation, this difference is observable in the presented operational semantics, where it is not in the original version. Interestingly, the behavior of the PAKCS system [HAB+15] corresponds to the original version, whereas the KiCS2 system [HBP+15] corresponds to the semantics proposed above. While the Curry language report [Han12] lacks a precise specification of the semantics of circular bindings such as `let {x = x} in x`, it states that

> ... Curry requires that *each free variable x must be explicitly declared* using a local declaration of the form x `free` [Han12, p. 10].

We think that this clearly advocates the explicit representation of free variables in the heap. Furthermore, the consideration of a self-referential binding as a non-terminating computation is in accordance with the functional language Haskell [Mar10].

5.2.5 Abstract Semantics

The advantage of the operational semantics is the ability to define the evaluation of FlatCurry expressions in a formal framework. Furthermore, it is considerably close to possible implementations, since certain implementation aspects such as the sharing of subexpressions are respected. In the remainder of this thesis, we will develop different variants of this semantics, and provide equivalence claims between those variants. For this purpose, we need to express that a given modification does not change the semantics of an expression, i. e., the value it evaluates to w.r.t. a given inference system. Therefore, we introduce a more abstract notion of semantics which associates an expression with the set of its values.

Definition 5.16 (Abstract Semantics [Bra11]). *The abstract semantics $[\![e]\!]_i^P$ of an expression e with respect to a program P and an operational semantics $\Downarrow_i$ is defined as*

$$[\![e]\!]_i^P := \{[\![\Gamma : v]\!] \mid [] : e \Downarrow_i^P \Gamma : v\}$$

$$[\![\Gamma : v]\!] := \begin{cases} \phi(\overline{\Gamma^+(x_k)}) & \text{if } v = \phi(\overline{x_k}) \text{ with } \phi \in \mathcal{C} \text{ or } k < \mathsf{arity}(\phi) \\ v & \text{otherwise} \end{cases}$$

where the recursive heap lookup operation $\Gamma^+(x)$ is defined as

$$\Gamma^+(x) = x \quad \text{if } x \notin \mathcal{D}om(\Gamma) \vee \Gamma(x) \in \{\mathsf{free}, \blacksquare\}$$

$$\Gamma[x \mapsto y]^+(x) = \begin{cases} x & \text{if } x = y \\ \Gamma^+(y) & \text{otherwise} \end{cases}$$

$$\Gamma[x \mapsto \phi(\overline{e_k})]^+(x) = \begin{cases} \phi(\overline{\Gamma[\overline{y_l \mapsto e'_l}]^+(x_k)}) & \text{if } \phi \in \mathcal{C} \text{ or } k < \mathsf{arity}(\phi) \\ x & \text{otherwise} \end{cases}$$

$$\text{where } (\overline{y_l}, \overline{e'_l}, \overline{x_k}) = \mathsf{splitArgs}(\overline{e_k})$$

$$\Gamma[x \mapsto \texttt{let } \{\overline{x_k = e_k}\} \texttt{ in } e]^+(x) = \Gamma[\overline{x_k \mapsto e_k}, x \mapsto e]^+(x)$$

$$\Gamma[x \mapsto \texttt{let } \overline{x_k} \texttt{ free in } e]^+(x) = \Gamma[\overline{x_k \mapsto \mathsf{free}}, x \mapsto e]^+(x)$$

$$\Gamma[x \mapsto e_1 \texttt{ ? } e_2]^+(x) = x$$

$$\Gamma[x \mapsto \texttt{case } e \texttt{ of } \{ \overline{p_k \to e_k} \}]^+(x) = x$$

Note that for the recursive cases of $\Gamma^+(x)$, the binding of x is either removed or replaced by bindings for sub-expressions, such that the operation is well-defined. Because the sets $[\![\cdot]\!]_i^P$ are potentially infinite due to variants of expressions, we consider equivalence classes of variants, and we may omit the program P if it is clear from the context.

This semantics abstracts from certain aspects of the out-configuration, since bindings to unevaluated expressions (except `let` expressions) or circular bindings are abstracted to the bound variable itself. In consequence, the result only contains evaluated subexpressions, and infinite structures are abstracted to finite ones.

Example 5.17 (Abstract Semantics)**.** *In correspondence to Example 5.12, the abstract semantics of the expression* `head(ones)` *is* $[\![\texttt{head(ones)}]\!]_0 = \{\texttt{1}\}$*. To give a more illustrating example, we consider the same program and the statement*

$$[] : \texttt{let ones = 1:ones in ones} \Downarrow_0 [\texttt{ones} \mapsto \texttt{one:ones}, \texttt{one} \mapsto \texttt{1}] : \texttt{one:ones}$$

so that we obtain $[\![\texttt{let ones = 1:ones in ones}]\!]_0 = \{\texttt{1:1:ones}\}$*. Note that the binding of* `ones` *has been abstracted to a finite list.*

5.3 Generalization to Non-Flat Programs

The presented semantics is defined for flat expressions only, and the flattening is performed to share the arguments of function and constructor calls to implement a call-by-need evaluation. Whenever a variable is evaluated, the binding in the heap is updated accordingly, so that repeated evaluations of the same variable share the same value and multiple evaluations of the binding are avoided.

However, it is neither necessary to apply the flattening operation to the entire program beforehand, nor to the entire right-hand side of a function declaration. Instead, it is also possible to introduce the sharing of subexpressions "on-the-fly", i. e., whenever a function or constructor application should be evaluated. This in turn requires an additional rule to extract non-variable subexpressions of function or constructor calls. While this approach makes the calculus more complex due to the additional rule, it allows the evaluation of non-flat expressions without any pre-processing. In consequence, the flattening of unevaluated subexpressions can be avoided, which generally leads to smaller configurations and preserves a closer correspondence to the initial program.

In the following, we will therefore generalize the semantics $\Downarrow_0$ to the semantics $\Downarrow_1$ which performs the flattening on demand and is thus also applicable for non-flat expressions. For this purpose, we first redefine the notions of heaps and configurations to also consider non-flat expressions, thus

$$\mathit{Heap} \subseteq \mathcal{V} \times (\{\text{free}, \blacksquare\} \uplus \mathit{Exp}) \ ,$$

and allow a configuration $\Gamma : e$ to contain a non-flat expression e. To be able to perform the flattening on demand, we replace the rule (Fun) by two different rules (Flatten) and (FunEval), responsible for the sharing of subexpressions and the evaluation of a flat function call, respectively.

$$\text{(Flatten)} \quad \frac{\Gamma[\overline{y_l \mapsto e'_l}] : \phi(\overline{x_k}) \Downarrow_1 \Delta : v}{\Gamma : \phi(\overline{e_k}) \Downarrow_1 \Delta : v} \quad \begin{array}{l}\text{where } \exists i \in \{1, \ldots, k\} \text{ such that } e_i \notin \mathcal{V}, \\ \text{and } (\overline{y_l}, \overline{e'_l}, \overline{x_k}) = \mathsf{splitArgs}(\overline{e_k})\end{array}$$

$$\text{(FunEval)} \quad \frac{\Gamma : e \Downarrow_1 \Delta : v}{\Gamma : f(\overline{x_n}) \Downarrow_1 \Delta : v} \quad \text{where } f(\overline{x_n}) = e \text{ is a variable instance of a rule in } P$$

In the rule (Flatten), a constructor or function application that contains at least one non-variable argument is transformed using the auxiliary function splitArgs. Note that the bindings added to the heap may still contain non-flat expressions, since the extracted expressions are not flattened themselves. For a flat function call, the rule (FunEval) can be applied, which replaces the function application by the (non-flattened) function body. For ease of further reference, we provide all rules of the semantics $\Downarrow_1$ in Figure 5.3.

(Value) $\Gamma : v \Downarrow_1 \Gamma : v$ where $v = \phi(\overline{x_k})$ with $\phi \in \mathcal{C}$ or $k < \mathsf{arity}(\phi)$, or $v \in \mathcal{V}$ with $\Gamma(v) = \mathsf{free}$

(VarExp) $$\frac{\Gamma[x \mapsto \blacksquare] : e \Downarrow_1 \Delta : v}{\Gamma[x \mapsto e] : x \Downarrow_1 \Delta[x \mapsto v] : v} \quad \text{where } e \notin \{\mathsf{free}, \blacksquare\}$$

(Flatten) $$\frac{\Gamma[\overline{y_l \mapsto e'_l}] : \phi(\overline{x_k}) \Downarrow_1 \Delta : v}{\Gamma : \phi(\overline{e_k}) \Downarrow_1 \Delta : v} \quad \begin{array}{l}\text{where } \exists i \in \{1,\ldots,k\} \text{ such that } e_i \notin \mathcal{V},\\ \text{and } (\overline{y_l}, \overline{e'_l}, \overline{x_k}) = \mathsf{splitArgs}(\overline{e_k})\end{array}$$

(FunEval) $$\frac{\Gamma : e \Downarrow_1 \Delta : v}{\Gamma : f(\overline{x_n}) \Downarrow_1 \Delta : v} \quad \text{where } f(\overline{x_n}) = e \text{ is a variable instance of a rule in } P$$

(Let) $$\frac{\Gamma[\overline{x_k \mapsto e_k}] : e \Downarrow_1 \Delta : v}{\Gamma : \mathtt{let}\ \{\overline{x_k = e_k}\}\ \mathtt{in}\ e \Downarrow_1 \Delta : v}$$

(Or) $$\frac{\Gamma : e_i \Downarrow_1 \Delta : v}{\Gamma : e_1\ \mathtt{?}\ e_2 \Downarrow_1 \Delta : v} \quad \text{where } i \in \{1,2\}$$

(Free) $$\frac{\Gamma[\overline{x_k \mapsto \mathsf{free}}] : e \Downarrow_1 \Delta : v}{\Gamma : \mathtt{let}\ \overline{x_k}\ \mathtt{free}\ \mathtt{in}\ e \Downarrow_1 \Delta : v}$$

(Select) $$\frac{\Gamma : e \Downarrow_1 \Delta : c(\overline{x_n}) \quad \Delta : \sigma(e_i) \Downarrow_1 \Theta : v}{\Gamma : \mathtt{case}\ e\ \mathtt{of}\ \{\ \overline{p_k \to e_k}\ \} \Downarrow_1 \Theta : v} \quad \begin{array}{l}\text{where } c(\overline{x_n}) = \sigma(p_i)\\ \text{and } i \in \{1,\ldots,k\}\end{array}$$

(Guess) $$\frac{\Gamma : e \Downarrow_1 \Delta : x \quad \Delta[x \mapsto c(\overline{x_n}), \overline{x_n \mapsto \mathsf{free}}] : e_i \Downarrow_1 \Theta : v}{\Gamma : \mathtt{case}\ e\ \mathtt{of}\ \{\ \overline{p_k \to e_k}\ \} \Downarrow_1 \Theta : v}$$
where $\Delta(x) = \mathsf{free}$, $c(\overline{x_n}) = p_i$, and $i \in \{1,\ldots,k\}$

Figure 5.3. Natural Semantics for General Expressions

5.3.1 Soundness

Both the semantics $\Downarrow_0$ are $\Downarrow_1$ equivalent in that they compute the same abstract semantics for a given expression e, and we provide a proof of this equivalence in the following. For this purpose, we first extend the operation flat to also be applicable to configurations as follows:

$$\begin{aligned}\mathsf{flat}(\Gamma : e) &= \mathsf{flat}(\Gamma) : \mathsf{flat}(e)\\ \mathsf{flat}(\Gamma) &= \{(x, \mathsf{flat}(b)) \mid (x, b) \in \Gamma\}\\ \mathsf{flat}(\mathsf{free}) &= \mathsf{free}\\ \mathsf{flat}(\blacksquare) &= \blacksquare\end{aligned}$$

We then show the soundness of the generalized semantics, i. e., the statements derivable using the generalized semantics $\Downarrow_1$ are also derivable by the initial semantics $\Downarrow_0$ after flattening of the configurations.

Lemma 5.18 (Soundness of Generalized Semantics)**.** *If* $\Gamma : e \Downarrow_1 \Delta : v$*, then* $\mathsf{flat}(\Gamma : e) \Downarrow_0 \mathsf{flat}(\Delta : v)$.

Proof. By structural induction on the derivation of the premise.

(Value) For the base case of rule (Value), we have to show that $\Gamma : v \Downarrow_1 \Gamma : v$ implies $\mathsf{flat}(\Gamma : v) \Downarrow_0 \mathsf{flat}(\Gamma : v)$, where $v = \phi(\overline{x_k})$ with $\phi \in \mathcal{C}$ or $k < \mathsf{arity}(\phi)$, or $v \in \mathcal{V}$ with $\Gamma(v) = \mathsf{free}$. Since $\mathsf{flat}(v) = v$ and $\mathsf{flat}(\mathsf{free}) = \mathsf{free}$, $\mathsf{flat}(\Gamma : v) \Downarrow_0 \mathsf{flat}(\Gamma : v)$ then holds by rule (Value).

We continue with the inductive cases and assume as the induction hypothesis that the claim holds for the premises of the assumed derivation.

(VarExp) We have to show that $\Gamma[x \mapsto e] : x \Downarrow_1 \Delta[x \mapsto v] : v$ where $e \notin \{\mathsf{free}, \blacksquare\}$ implies $\mathsf{flat}(\Gamma[x \mapsto e] : x) \Downarrow_0 \mathsf{flat}(\Delta[x \mapsto v] : v)$, and the induction hypothesis states that $\Gamma[x \mapsto \blacksquare] : e \Downarrow_1 \Delta : v$ implies $\mathsf{flat}(\Gamma[x \mapsto \blacksquare] : e) \Downarrow_0 \mathsf{flat}(\Delta : v)$. We can then construct the following derivation by rule (VarExp):

$$\dfrac{\begin{array}{c}\mathsf{flat}(\Gamma[x \mapsto \blacksquare] : e) \Downarrow_0 \mathsf{flat}(\Delta : v)\\ = \quad \mathsf{flat}(\Gamma)[x \mapsto \blacksquare] : \mathsf{flat}(e) \Downarrow_0 \mathsf{flat}(\Delta) : v\end{array}}{\begin{array}{rc}& \mathsf{flat}(\Gamma)[x \mapsto \mathsf{flat}(e)] : x \Downarrow_0 \mathsf{flat}(\Delta)[x \mapsto v] : v\\ = & \quad \mathsf{flat}(\Gamma[x \mapsto e] : x) \Downarrow_0 \mathsf{flat}(\Delta[x \mapsto v] : v)\end{array}}$$

(Flatten) We have to show that $\Gamma : \phi(\overline{e_k}) \Downarrow_1 \Delta : v$ where $\exists i \in \{1, \ldots, k\}$ such that $e_i \notin \mathcal{V}$ and $(\overline{y_l}, \overline{e'_l}, \overline{x_k}) = \mathsf{splitArgs}(\overline{e_k})$ implies $\mathsf{flat}(\Gamma : \phi(\overline{e_k})) \Downarrow_0 \mathsf{flat}(\Delta : v)$, and the induction hypothesis states that $\Gamma[\overline{y_l \mapsto e'_l}] : \phi(\overline{x_k}) \Downarrow_1 \Delta : v$ implies $\mathsf{flat}(\Gamma[\overline{y_l \mapsto e'_l}] : \phi(\overline{x_k})) \Downarrow_0 \mathsf{flat}(\Delta : v)$. By the definition of flattening, we have $\mathsf{flat}(\phi(\overline{e_k})) = \mathtt{let}\ \{\overline{y_l = \mathsf{flat}(e'_l)}\}\ \mathtt{in}\ \phi(\overline{x_k})$, and can construct the following derivation by rule (Let):

$$\dfrac{\begin{array}{c}\mathsf{flat}(\Gamma[\overline{y_l \mapsto e'_l}] : \phi(\overline{x_k})) \Downarrow_0 \mathsf{flat}(\Delta : v)\\ = \quad \mathsf{flat}(\Gamma)[\overline{y_l \mapsto \mathsf{flat}(e'_l)}] : \phi(\overline{x_k}) \Downarrow_0 \mathsf{flat}(\Delta) : v\end{array}}{\begin{array}{rc}& \mathsf{flat}(\Gamma) : \mathtt{let}\ \{\overline{y_l = \mathsf{flat}(e'_l)}\}\ \mathtt{in}\ \phi(\overline{x_k}) \Downarrow_0 \mathsf{flat}(\Delta) : v\\ = & \quad \mathsf{flat}(\Gamma : \phi(\overline{e_k})) \Downarrow_0 \mathsf{flat}(\Delta : v)\end{array}}$$

(FunEval) We have to show that $\Gamma : f(\overline{x_n}) \Downarrow_1 \Delta : v$ where $f(\overline{x_n}) = e$ is a variable instance of a rule in P implies $\mathsf{flat}(\Gamma : f(\overline{x_n})) \Downarrow_0 \mathsf{flat}(\Delta : v)$, and the induction hypothesis states that $\Gamma : e \Downarrow_1 \Delta : v$ implies $\mathsf{flat}(\Gamma : e) \Downarrow_0 \mathsf{flat}(\Delta : v)$. We can then construct the following derivation by rule (Fun):

$$\dfrac{\begin{array}{rc} & \mathsf{flat}(\Gamma : e) \Downarrow_0 \mathsf{flat}(\Delta : v) \\ = & \mathsf{flat}(\Gamma) : \mathsf{flat}(e) \Downarrow_0 \mathsf{flat}(\Delta) : v \end{array}}{\begin{array}{rc} & \mathsf{flat}(\Gamma) : f(\overline{x_n}) \Downarrow_0 \mathsf{flat}(\Delta) : v \\ = & \mathsf{flat}(\Gamma : f(\overline{x_n})) \Downarrow_0 \mathsf{flat}(\Delta : v) \end{array}}$$

(Let), (Or), (Free) For these rules, the claim directly follows from the induction hypothesis and the application of the same rule in the semantics $\Downarrow_0$.

(Select) We have to show that $\Gamma : \mathsf{case}\ e\ \mathsf{of}\ \{\ \overline{p_k \to e_k}\ \} \Downarrow_1 \Theta : v$ implies $\mathsf{flat}(\Gamma : \mathsf{case}\ e\ \mathsf{of}\ \{\ \overline{p_k \to e_k}\ \}) \Downarrow_0 \mathsf{flat}(\Theta : v)$, and the induction hypothesis states that $\Gamma : e \Downarrow_1 \Delta : c(\overline{x_n})$ implies $\mathsf{flat}(\Gamma : e) \Downarrow_0 \mathsf{flat}(\Delta : c(\overline{x_n}))$ and $\Delta : \sigma(e_i) \Downarrow_1 \Theta : v$ implies $\mathsf{flat}(\Delta : \sigma(e_i)) \Downarrow_0 \mathsf{flat}(\Theta : v)$, where $c(\overline{x_n}) = \sigma(p_i)$ and $i \in \{1, \ldots, k\}$. Since σ is a variable renaming and flat does not remove variables but only introduces fresh ones, we have $\mathsf{flat}(\sigma(e_i)) = \sigma(\mathsf{flat}(e_i))$ and can construct the following derivation by rule (Select):

$$\dfrac{\begin{array}{rc} & \mathsf{flat}(\Gamma : e) \Downarrow_0 \mathsf{flat}(\Delta : c(\overline{x_n})) \\ = & \mathsf{flat}(\Gamma) : \mathsf{flat}(e) \Downarrow_0 \mathsf{flat}(\Delta) : c(\overline{x_n}) \end{array} \qquad \begin{array}{rc} & \mathsf{flat}(\Delta : \sigma(e_i)) \Downarrow_0 \mathsf{flat}(\Theta : v) \\ = & \mathsf{flat}(\Delta) : \sigma(\mathsf{flat}(e_i)) \Downarrow_0 \mathsf{flat}(\Theta) : v \end{array}}{\begin{array}{rc} & \mathsf{flat}(\Gamma) : \mathsf{case}\ \mathsf{flat}(e)\ \mathsf{of}\ \{\ \overline{p_k \to \mathsf{flat}(e_k)}\ \} \Downarrow_0 \mathsf{flat}(\Theta) : v \\ = & \mathsf{flat}(\Gamma : \mathsf{case}\ e\ \mathsf{of}\ \{\ \overline{p_k \to e_k}\ \}) \Downarrow_0 \mathsf{flat}(\Theta : v) \end{array}}$$

(Guess) This case follows with the same reasoning as for rule (Select) with the simplification that no variable renaming σ has to be considered. □

5.3.2 Completeness

The second result states the completeness of the generalized semantics, i. e., all statements derivable in the initial semantics $\Downarrow_0$ after flattening are also computed by the generalized semantics $\Downarrow_1$ without flattening.

Lemma 5.19 (Completeness of Generalized Semantics)**.** *Let* $\Gamma : e$ *and* $\Gamma' : e'$ *be two configurations such that* $\mathsf{flat}(\Gamma' : e') = \Gamma : e$*. If* $\Gamma : e \Downarrow_0 \Delta : v$*, then there exists a heap* Δ' *such that* $\Gamma' : e' \Downarrow_1 \Delta' : v$ *and* $\mathsf{flat}(\Delta') = \Delta$*.*

Proof. By structural induction on the derivation of the premise.

(Value) For the base case of rule (Value), we have to show that $\Gamma : v \Downarrow_0 \Gamma : v$ and $\mathsf{flat}(\Gamma' : v) = \Gamma : v$ imply $\Gamma' : v \Downarrow_1 \Delta' : v$ and $\mathsf{flat}(\Delta') = \Gamma$, where $v = \phi(\overline{x_k})$ with $\phi \in \mathcal{C}$ or $k < \mathsf{arity}(\phi)$, or $v \in \mathcal{V}$ with $\Gamma(v) = \mathsf{free}$. Since $\Gamma(x) = \mathsf{free}$ implies $\Gamma'(x) = \mathsf{free}$, $\Gamma' : v \Downarrow_1 \Delta' : v$ holds for $\Delta' = \Gamma'$ by rule (Value).

We continue with the inductive cases and assume as the induction hypothesis that the claim holds for the premises of the assumed derivation.

(VarExp) We have to show that $\Gamma[x \mapsto e] : x \Downarrow_0 \Delta[x \mapsto v] : v$ where $e \notin \{\text{free}, \blacksquare\}$ and $\mathsf{flat}(\Gamma'[x \mapsto e'] : x) = \Gamma[x \mapsto e] : x$ imply $\Gamma'[x \mapsto e'] : x \Downarrow_1 \Delta'[x \mapsto v] : v$ and $\mathsf{flat}(\Delta'[x \mapsto v]) = \Delta[x \mapsto v]$, where the induction hypothesis states that $\Gamma[x \mapsto \blacksquare] : e \Downarrow_0 \Delta : v$ and $\mathsf{flat}(\Gamma'[x \mapsto \blacksquare] : e') = \Gamma[x \mapsto \blacksquare] : e$ imply $\Gamma'[x \mapsto \blacksquare] : e' \Downarrow_1 \Delta' : v$ and $\mathsf{flat}(\Delta') = \Delta$. We can then construct the following derivation by rule (VarExp):

$$\frac{\Gamma'[x \mapsto \blacksquare] : e' \Downarrow_1 \Delta' : v}{\Gamma'[x \mapsto e'] : x \Downarrow_1 \Delta'[x \mapsto v] : v}$$

Furthermore, $\mathsf{flat}(\Delta'[x \mapsto v]) = \Delta[x \mapsto v]$ follows from $\mathsf{flat}(\Delta') = \Delta$ and $\mathsf{flat}(v) = v$, where the latter holds because v is a value.

(Fun) We have to show that $\Gamma : f(\overline{x_n}) \Downarrow_0 \Delta : v$ where $f(\overline{x_n}) = e$ is a variable instance of a rule in P and $\mathsf{flat}(\Gamma' : f(\overline{x_n})) = \Gamma : f(\overline{x_n})$ imply $\Gamma' : f(\overline{x_n}) \Downarrow_1 \Delta' : v$ and $\mathsf{flat}(\Delta') = \Delta$, where the induction hypothesis states that $\Gamma : \mathsf{flat}(e) \Downarrow_0 \Delta : v$ and $\mathsf{flat}(\Gamma' : e') = \Gamma : \mathsf{flat}(e)$ imply $\Gamma' : e' \Downarrow_1 \Delta' : v$ and $\mathsf{flat}(\Delta') = \Delta$. For $e' = e$, we can then construct the following derivation by rule (FunEval):

$$\frac{\Gamma' : e \Downarrow_1 \Delta' : v}{\Gamma' : f(\overline{x_n}) \Downarrow_1 \Delta' : v}$$

(Let) We have to show that $\Gamma : \texttt{let } \{\overline{x_k = e_k}\} \texttt{ in } e \Downarrow_0 \Delta : v$ and $\mathsf{flat}(\Gamma' : e'') = \Gamma : \texttt{let } \{\overline{x_k = e_k}\} \texttt{ in } e$ imply $\Gamma' : e'' \Downarrow_1 \Delta' : v$ and $\mathsf{flat}(\Delta') = \Delta$, where the induction hypothesis states that $\Gamma[\overline{x_k \mapsto e_k}] : e \Downarrow_0 \Delta : v$ and $\mathsf{flat}(\Gamma'[\overline{x_k \mapsto e'_k}] : e') = \Gamma[\overline{x_k \mapsto e_k}] : e$ imply $\Gamma'[\overline{x_k \mapsto e'_k}] : e' \Downarrow_1 \Delta' : v$ and $\mathsf{flat}(\Delta') = \Delta$. We distinguish two cases for e'' such that $\mathsf{flat}(e'') = \texttt{let } \{\overline{x_k = e_k}\} \texttt{ in } e$ holds.

1. If $e'' = \texttt{let } \{\overline{x_k = e'_k}\} \texttt{ in } e'$ with $\mathsf{flat}(e'_i) = e_i$ for all $i \in \{1, \ldots, k\}$ and $\mathsf{flat}(e') = e$, then we can construct the following derivation by rule (Let):

$$\frac{\Gamma'[\overline{x_k \mapsto e'_k}] : e' \Downarrow_1 \Delta' : v}{\Gamma' : \texttt{let } \{\overline{x_k = e'_k}\} \texttt{ in } e' \Downarrow_1 \Delta' : v}$$

2. If the `let` expression results from the application of flat, then $e'' = \phi(\overline{e'''_l})$ with $\mathsf{flat}(e'') = \texttt{let } \{\overline{x_k = \mathsf{flat}(e'_k)}\} \texttt{ in } \phi(\overline{y_l}) = \texttt{let } \{\overline{x_k = e_k}\} \texttt{ in } e$. This implies $(\overline{x_k}, \overline{e'_k}, \overline{y_l}) = \mathsf{splitArgs}(\overline{e'''_l})$, so that we can construct the following derivation by rule (Flatten):

$$\frac{\Gamma'[\overline{x_k \mapsto e'_k}] : \phi(\overline{y_l}) \Downarrow_1 \Delta' : v}{\Gamma' : \phi(\overline{e'''_l}) \Downarrow_1 \Delta' : v}$$

(Or), (Free) For these rules, the claim follows from the induction hypothesis and the application of the same rule in the semantics $\Downarrow_1$.

(Select) For rule (Select) we have to show that $\Gamma : \texttt{case}\ e\ \texttt{of}\ \{\ \overline{p_k \to e_k}\ \} \Downarrow_0 \Theta : v$ and $\mathsf{flat}(\Gamma' : \texttt{case}\ e'\ \texttt{of}\ \{\ \overline{p_k \to e'_k}\ \}) = \Gamma : \texttt{case}\ e\ \texttt{of}\ \{\ \overline{p_k \to e_k}\ \}$ imply that $\Gamma' : \texttt{case}\ e'\ \texttt{of}\ \{\ \overline{p_k \to e'_k}\ \} \Downarrow_1 \Theta' : v$ and $\mathsf{flat}(\Theta') = \Theta$. The induction hypothesis states that $\Gamma : e \Downarrow_0 \Delta : c(\overline{x_n})$ and $\mathsf{flat}(\Gamma' : e') = \Gamma : e$ imply $\Gamma' : e' \Downarrow_1 \Delta' : c(\overline{x_n})$ and $\mathsf{flat}(\Delta') = \Delta$, and that $\Delta : \sigma(e_i) \Downarrow_0 \Theta : v$ and $\mathsf{flat}(\Delta' : \sigma(e'_i)) = \Delta : \sigma(e_i)$ imply $\Delta' : \sigma(e'_i) \Downarrow_1 \Theta' : v$ and $\mathsf{flat}(\Theta') = \Theta$, where $c(\overline{x_n}) = \sigma(p_i)$ and $i \in \{1, \ldots, k\}$. By assumption we have $\mathsf{flat}(e'_i) = e_i$, and therefore also $\mathsf{flat}(\sigma(e'_i)) = \sigma(\mathsf{flat}(e'_i)) = \sigma(e_i)$, since flat does not remove variables but only introduces fresh ones. We can then construct the following derivation by rule (Select):

$$\frac{\Gamma' : e' \Downarrow_1 \Delta' : c(\overline{x_n}) \quad \Delta' : \sigma(e'_i) \Downarrow_1 \Theta' : v}{\Gamma' : \texttt{case}\ e'\ \texttt{of}\ \{\ \overline{p_k \to e'_k}\ \} \Downarrow_1 \Theta' : v}$$

(Guess) This case follows with the same reasoning as for rule (Select) with the simplification that no variable renaming σ has to be considered. □

5.3.3 Summary

Based on the two lemmata stating the soundness and completeness of the generalized semantics, we can state its correctness.

Theorem 5.20 (Correctness of Generalized Semantics)**.** *$\Gamma : e \Downarrow_1 \Delta : v$ if and only if $\mathsf{flat}(\Gamma : e) \Downarrow_0 \mathsf{flat}(\Delta : v)$.*

Proof. Direct consequence of Lemma 5.18 and Lemma 5.19. □

While the above result shows that the same statements (modulo flattening) are derivable in both semantics, they furthermore lead to the same abstract semantics of an expression. To be able to formally prove this statement, we first recall the notion of multisets and the ordering of multisets of natural numbers, before we define the *depth* of an expression as well as the *complexity* of a heap and a configuration.

Definition 5.21 (Multiset [BN98])**.** *A multiset M over a set A is a function $M : A \to \mathbb{N}$ where intuitively $M(x)$ is the number of copies of $x \in A$ in M. A multiset M is finite if there are only finitely many x such that $M(x) > 0$. Let $\mathcal{M}(A)$ denote the set of all finite multisets over A.*

In the following, we will consider multisets $M \in \mathcal{M}(\mathbb{N})$ over natural numbers, and denote multisets like ordinary sets using braces with repeated elements, e. g., $\{1, 2, 3\}$ and $\{1, 2, 2, 3\}$ are different multisets of natural numbers.

Definition 5.22 (Multiset Ordering [BN98])**.** *Let $M, M' \in \mathcal{M}(\mathbb{N})$ be two multisets over natural numbers. We say $M <_{\mathsf{mul}} M'$ if and only if there exist $X \subseteq M$ and $X' \subseteq M'$ such that $M = (M' \setminus X') \cup X$ and for all $n \in X$ there exists $n' \in X'$ such that $n < n'$, where $<$ is the usual strict partial order over $\mathbb{N}$.*

For instance, we have $\{1,2,3\} <_{\mathsf{mul}} \{1,2,2,3\}$ with $X' = \{2\}$ and $X = \varnothing$, and $\{1,2,2,2\} <_{\mathsf{mul}} \{1,2,3\}$ for $X' = \{3\}$ and $X = \{2,2\}$. Since there are no infinite decreasing chains for finite multisets of natural numbers, the ordering $<_{\mathsf{mul}}$ is well-founded, and can therefore be used for complete induction over $\mathcal{M}(\mathbb{N})$.

Definition 5.23 (Depth of Expression)**.** *The depth of an expression e, denoted by $\mathsf{depth}(e)$, corresponds to the maximum number of nested symbols in e, and is inductively defined as:*

$$\begin{aligned}
\mathsf{depth}(x) &= 1 \\
\mathsf{depth}(\phi(\overline{e_k})) &= \begin{cases} 1 & \text{if } e_i \in \mathcal{V} \text{ for all } i \in \{1,\ldots,k\} \\ 1 + \max(\overline{\mathsf{depth}(e_k)}) & \text{otherwise} \end{cases} \\
\mathsf{depth}(\texttt{let } \{\overline{x_k = e_k}\} \texttt{ in } e) &= 1 + \max(\mathsf{depth}(e), \overline{\mathsf{depth}(e_k)}) \\
\mathsf{depth}(\texttt{let } \overline{x_k} \texttt{ free in } e) &= 1 + \mathsf{depth}(e) \\
\mathsf{depth}(e_1 \texttt{ ? } e_2) &= 1 + \max(\mathsf{depth}(e_1), \mathsf{depth}(e_2)) \\
\mathsf{depth}(\texttt{case } e \texttt{ of \{ } \overline{p_k \to e_k} \texttt{ \}}) &= 1 + \max(\mathsf{depth}(e), \overline{\mathsf{depth}(e_k)})
\end{aligned}$$

where max *computes the maximum of a sequence of numbers and* $\max(\varepsilon) = 0$*.*

For instance, we have $\mathsf{depth}(\texttt{Just(2 + 3)}) = 1 + \max(\mathsf{depth}(\texttt{2 + 3})) = 1 + 1 + \max(\mathsf{depth}(2), \mathsf{depth}(3)) = 2 + \max(1,1) = 3$. Note that this definition assigns a depth of 1 to constructor and function symbols applied to variables. This definition implies that values always have a depth of one, and furthermore ensures that the depth of the left-hand side of a function definition is independent of the function's arity. For instance, if $f \in \mathcal{F}^{(0)}$ and $g \in \mathcal{F}^{(2)}$, then f and $g(x,y)$ share the same depth.

Using the depth of nested symbols, we can then define the complexity of a multiset of expressions as well as the complexity of a configuration.

Definition 5.24 (Complexity of a Multiset of Expressions)**.** *Let E be a finite multiset of expressions. The complexity $\mathcal{M}_E$ of E is the finite multiset of natural numbers corresponding to the depth of the elements of E, i. e., $\mathcal{M}_E = \{\mathsf{depth}(e) \mid e \in E\}$.*

Definition 5.25 (Complexity of a Configuration)**.** *The multiset complexity of a heap Γ, denoted by $\mathcal{M}_\Gamma$, is defined as*

$$\mathcal{M}_\Gamma = \{\mathsf{depth}(e) \mid (x,e) \in \Gamma \wedge e \in \mathit{Exp}\} \,.$$

The multiset complexity of a configuration $\Gamma : e$, denoted by $\mathcal{M}_{\Gamma:e}$, is defined as

$$\mathcal{M}_{\Gamma:e} = \mathcal{M}_\Gamma \cup \{\mathsf{depth}(e)\} \,.$$

Using these definitions, we can continue to show that the operation of flattening does not influence the result of the heap lookup operation.

Lemma 5.26 (Heap Lookup under Flattening)**.** *For any heap Γ and variable x, it holds $\Gamma^+(x) = \mathsf{flat}(\Gamma)^+(x)$.*

Proof. By well-founded induction on the complexity of the heap Γ. We start with the base case of $\mathcal{M}_\Gamma = \varnothing$, which implies $\Gamma = []$ and we have $[]^+(x) = x = \mathsf{flat}([])^+(x)$. We continue with the inductive cases and assume as the induction hypothesis that the claim holds for all heaps Γ' with $\mathcal{M}_{\Gamma'} <_{\mathsf{mul}} \mathcal{M}_\Gamma$.

- ▷ If $x \notin \mathcal{D}om(\Gamma)$ or $\Gamma(x) \in \{\mathsf{free}, \blacksquare, x\}$, then $\Gamma^+(x) = x = \mathsf{flat}(\Gamma)^+(x)$.
- ▷ If $\Gamma = \Gamma'[x \mapsto y]$ where $x \neq y$, then $\Gamma'[x \mapsto y]^+(x) = \Gamma'^+(y)$ for the left-hand side and $\mathsf{flat}(\Gamma'[x \mapsto y])^+(x) = \mathsf{flat}(\Gamma')^+(y)$ for the right-hand side of the claim, and their equality follows from the induction hypothesis.
- ▷ If $\Gamma = \Gamma'[x \mapsto \phi(\overline{e_k})]$ with $\phi \in \mathcal{C}$ or $k < \mathsf{arity}(\phi)$, then $\Gamma'[x \mapsto \phi(\overline{e_k})]^+(x) = \phi(\Gamma'[\overline{y_l \mapsto e'_l}]^+(x_k))$ for $(\overline{y_l}, \overline{e'_l}, \overline{x_k}) = \mathsf{splitArgs}(\overline{e_k})$, and

$$\begin{aligned}
& \mathsf{flat}(\Gamma'[x \mapsto \phi(\overline{e_k})])^+(x) \\
= \;& \mathsf{flat}(\Gamma')[x \mapsto \texttt{let } \{\overline{y_l = \mathsf{flat}(e'_l)}\} \texttt{ in } \phi(\overline{x_k})]^+(x) \\
= \;& \mathsf{flat}(\Gamma')[\overline{y_l \mapsto \mathsf{flat}(e'_l)}, x \mapsto \phi(\overline{x_k})]^+(x) \\
= \;& \phi(\mathsf{flat}(\Gamma')[\overline{y_l \mapsto \mathsf{flat}(e'_l)}]^+(x_k)) \\
= \;& \phi(\mathsf{flat}(\Gamma'[\overline{y_l \mapsto e'_l}])^+(x_k)) \,,
\end{aligned}$$

  and their equality follows from the induction hypothesis.
- ▷ If $\Gamma = \Gamma'[x \mapsto f(\overline{e_n})]$ with $f \in \mathcal{F}^{(n)}$, then for $(\overline{y_l}, \overline{e'_l}, \overline{x_n}) = \mathsf{splitArgs}(\overline{e_n})$ we have $\Gamma'[x \mapsto f(\overline{e_n})]^+(x) = x$ and

$$\begin{aligned}
& \mathsf{flat}(\Gamma'[x \mapsto f(\overline{e_n})])^+(x) \\
= \;& \mathsf{flat}(\Gamma')[x \mapsto \texttt{let } \{\overline{y_l = \mathsf{flat}(e'_l)}\} \texttt{ in } f(\overline{x_n})]^+(x) \\
= \;& \mathsf{flat}(\Gamma')[\overline{y_l \mapsto \mathsf{flat}(e'_l)}, x \mapsto f(\overline{x_n})]^+(x) \\
= \;& x \,.
\end{aligned}$$

- ▷ If $\Gamma = \Gamma'[x \mapsto \texttt{let } \{\overline{x_k = e_k}\} \texttt{ in } e]$, then

$$\Gamma'[x \mapsto \texttt{let } \{\overline{x_k = e_k}\} \texttt{ in } e]^+(x) = \Gamma'[\overline{x_k \mapsto e_k}, x \mapsto e]^+(x)$$

  and

$$\begin{aligned}
& \mathsf{flat}(\Gamma'[x \mapsto \texttt{let } \{\overline{x_k = e_k}\} \texttt{ in } e])^+(x) \\
= \;& \mathsf{flat}(\Gamma')[x \mapsto \texttt{let } \{\overline{x_k = \mathsf{flat}(e_k)}\} \texttt{ in } \mathsf{flat}(e)])^+(x) \\
= \;& \mathsf{flat}(\Gamma')[\overline{x_k \mapsto \mathsf{flat}(e_k)}, x \mapsto \mathsf{flat}(e)])^+(x) \\
= \;& \mathsf{flat}(\Gamma'[\overline{x_k \mapsto e_k}, x \mapsto e])^+(x) \,,
\end{aligned}$$

  and their equality follows from the induction hypothesis.
- ▷ The case for $\Gamma = \Gamma'[x \mapsto \texttt{let } \overline{x_k} \texttt{ free in } e]$ follows with the same reasoning as for the previous case.
- ▷ If $\Gamma = \Gamma'[x \mapsto e_1 \texttt{ ? } e_2]$, then $\Gamma'[x \mapsto e_1 \texttt{ ? } e_2]^+(x) = x$ and $\mathsf{flat}(\Gamma'[x \mapsto e_1 \texttt{ ? } e_2])^+(x) = \mathsf{flat}(\Gamma')[x \mapsto \mathsf{flat}(e_1) \texttt{ ? } \mathsf{flat}(e_2)]^+(x)) = x$.

▷ The case for $\Gamma = \Gamma'[x \mapsto \texttt{case}\ e\ \texttt{of}\ \{\ \overline{p_k \to e_k}\ \}]$ follows with the same reasoning as for the previous case. □

We can now state the equivalence of the abstract semantics of an expression computed using either the initial semantics $\Downarrow_0$ or the generalized semantics $\Downarrow_1$.

Corollary 5.27. *For any expression e, it holds $[\![\mathsf{flat}(e)]\!]^P_0 = [\![e]\!]^P_1$.*

Proof. Consequence of Definition 5.16, Theorem 5.20, and Lemma 5.26. □

5.4 Extensions of the Semantics

In the following, we will extend the operational semantics to deal with additional language constructs of Curry by providing additional inference rules for their FlatCurry representation. This concerns special built-in operations for primitive functions such as arithmetics, higher-order functions, equational constraints, and functional patterns. The extensions for primitives, higher-order application and strict unification can also be found in a similar style for the original semantics of Albert et al. [AHH+05]. Note that rule (Flatten) is also applied to primitive operations, so that they will always be applied to variable arguments.

5.4.1 Primitive Operations

For applications of primitive operations such as arithmetic functions, we assume these operations to require the evaluation of all their arguments to *head normal form* before the result can be computed. For this purpose, we introduce a predefined function $\texttt{hnf}(x_1, x_2)$ that evaluates its first argument to a head normal form and then evaluates its second argument to obtain the result:

$$\text{(HNF)} \qquad \frac{\Gamma : x_1 \Downarrow_1 \Delta : v_1 \quad \Delta : x_2 \Downarrow_1 \Theta : v_2}{\Gamma : \texttt{hnf}(x_1, x_2) \Downarrow_1 \Theta : v_2}$$

A primitive operation f may then be defined by means of the function `hnf` in conjunction with a specific rule for computation of the result. For instance, the primitive addition operation is considered to be defined as

```
x + y = hnf(x, hnf(y, plus(x, y)))
```

where the auxiliary operation `plus` is defined as:

$$\text{(prim-+)} \qquad \frac{\Gamma : x_1 \Downarrow_1 \Delta : l_1 \quad \Gamma : x_2 \Downarrow_1 \Delta : l_2}{\Gamma : \texttt{plus}(x_1, x_2) \Downarrow_1 \Gamma : l_1 +_{\mathcal{A}} l_2}$$

where l_1, l_2 are integer literals and $+_{\mathcal{A}}$ is the arithmetic sum

Due to the prior evaluation of the arguments by means of the operation `hnf`, their (repeated) evaluation in the definition of `plus` reduces to a dereferencing of variables.

This could also be expressed by a heap lookup operation following variable chains [AHH+05], but the above formalization has the advantage that it also updates the dereferenced variable bindings due to rule (VarExp).

The semantics of operations that require the *complete evaluation* of their arguments can be expressed by means of the auxiliary primitive operation `nf`. This operation completely evaluates its argument by recursive evaluation of the arguments of constructor applications. Since partial applications already are completely evaluated, the recursive evaluation is therefore restricted to fully applied constructors. Since this evaluation corresponds to normal form computation in term rewriting systems, we call it `nf` accordingly.

$$\text{(NF-NCons)}\quad \frac{\Gamma : x \Downarrow_1 \Delta : v}{\Gamma : \mathtt{nf}(x) \Downarrow_1 \Delta : v} \quad \text{where } v \in \mathcal{V} \text{ or } v = \phi(\overline{x_k}) \text{ with } k < \mathsf{arity}(\phi)$$

$$\text{(NF-Cons)}\quad \frac{\Gamma : x \Downarrow_1 \Gamma_0 : c(\overline{x_n}) \quad \Gamma_0 : \mathtt{nf}(x_1) \Downarrow_1 \Gamma_1 : v_1 \quad \cdots \quad \Gamma_{n-1} : \mathtt{nf}(x_n) \Downarrow_1 \Gamma_n : v_n}{\Gamma : \mathtt{nf}(x) \Downarrow_1 \Gamma_n : c(\overline{x_n})} \quad \text{where } c \in \mathcal{C}^{(n)}$$

The definition of a primitive operation f that requires the complete evaluation of its arguments can then be expressed as

```
f (x1, ..., xn) = hnf(nf(x1), hnf(... hnf(nf(xn), prim_f(x1, ..., xn)) ...))
```

For a call $f(\overline{x_n})$, the computation of the result must then also consider the heap to access the values below the head normal form, i. e., the result value is computed by $f_{\mathcal{A}}(\Gamma_n, \overline{v_n})$ where $\overline{v_n}$ are the values of $\overline{x_n}$ and Γ_n is the heap after complete evaluation of all arguments.

5.4.2 Higher-Order Application

Until now, we have mainly considered first-order programs where all function and constructor applications were fully saturated. We already allowed the representation of partial applications in the definition of FlatCurry expressions in Section 5.1, but provided no rule to apply them to additional arguments so far. For this purpose, we use the fact that higher-order application in FlatCurry is represented by a call to the primitive function `apply`, for which we define the semantics by the following rule.

$$\text{(Apply)}\quad \frac{\Gamma : x \Downarrow_1 \Delta : \phi(\overline{x_k}) \quad \Delta : \phi(\overline{x_k}, y) \Downarrow_1 \Theta : v}{\Gamma : \mathtt{apply}(x, y) \Downarrow_1 \Theta : v} \quad \text{where } k < \mathsf{arity}(\phi)$$

Hence, to evaluate $\mathtt{apply}(x, y)$ its first argument x is evaluated to a partial application $\phi(\overline{x_k})$. This application is then extended by the additional argument y and further evaluated afterwards. Note that logic variables as the value of the first argument are not allowed, since this would require arbitrary functions to be guessed.

5.4.3 Strict Unification

An equational constraint of the form e_1 `=:=` e_2 is solvable if both expressions can be evaluated to unifiable constructor expressions. Although it would be possible to implement strict unification by a complete evaluation of both expressions and a subsequent unification, this is rather inefficient since the unification can immediately fail in case of different constructors. Furthermore, the complete evaluation may even not terminate. Therefore, the strict unification e_1 `=:=` e_2 is usually implemented by a recursive evaluation of e_1 and e_2 to head normal forms, followed by a comparison of the constructors, a possible instantiation of logic variables, and a recursive unification of the subexpressions. Informally, the strict unification proceeds as follows:

1. Evaluate both expressions e_1 and e_2 to head normal forms h_1 and h_2.

2. (a) If h_1 and h_2 yield the same logic variable, then yield the result value `Success`.
   (b) If h_1 and h_2 yield two different logic variables, then bind h_1 to h_2 and yield the result value `Success`.
   (c) If h_1 and h_2 are rooted by the *same* constructor, i. e., $h_1 = c(\overline{e'_n})$ and $h_2 = c(\overline{e''_n})$, then recursively evaluate the new constraint e'_1 `=:=` e''_1 `&` $\cdots$ `&` e'_n `=:=` e''_n. If h_1 and h_2 are constants, i. e., if $n = 0$, then the constraint reduces to `Success`.
   (d) If one expression evaluates to a logic variable and the other to a constructor-rooted value $c(\overline{e_n})$, then bind the variable to the constructor applied to fresh logic variables, and recursively unify the fresh variables with the arguments of the constructor-rooted expression (or solve the constraint `Success` if $n = 0$).
   (e) If h_1 and h_2 are rooted by *different* constructors, then evaluation fails.

Note that the case (2d) may lead to non-termination if the variable to be bound occurs in the value it is bound to, such as in the expression `let x free in x =:= 1:x`. This can be improved by means of an *occur check*, which fails if a variable occurs in the set of critical variables of the value it should be bound to. However, this is only an optimization which does not affect the semantics of the operation, so that we skip the occur check in the formal presentation. From the above algorithm, we can directly derive the definition of the strict unification as

```
x =:= y = hnf(x, hnf(y, prim_su(x, y)))
```

together with the following set of rules that specify the operational semantics of `prim_su`.

$$\text{(SU-Same)}\quad \frac{\Gamma : z_1 \Downarrow_1 \Delta : x \quad \Delta : z_2 \Downarrow_1 \Theta : x}{\Gamma : \texttt{prim\_su}(z_1, z_2) \Downarrow_1 \Theta : \texttt{Success}} \quad \text{where } \Delta(x) = \text{free and } \Theta(x) = \text{free}$$

$$\text{(SU-Vars)}\quad \frac{\Gamma : z_1 \Downarrow_1 \Delta : x \quad \Delta : z_2 \Downarrow_1 \Theta : y}{\Gamma : \texttt{prim\_su}(z_1, z_2) \Downarrow_1 \Theta[x \mapsto y] : \texttt{Success}} \quad \begin{array}{l}\text{where } x \neq y, \Delta(x) = \text{free},\\ \text{and } \Theta(y) = \text{free}\end{array}$$

$$\text{(SU-Cons)}\quad \frac{\Gamma : z_1 \Downarrow_1 \Delta : c(\overline{x_n}) \quad \Delta : z_2 \Downarrow_1 \Theta : c(\overline{y_n}) \quad \Theta : x_1 \texttt{ =:= } y_1 \texttt{ \& } \cdots \texttt{ \& } x_n \texttt{ =:= } y_n \Downarrow_1 \Omega : v}{\Gamma : \texttt{prim\_su}(z_1, z_2) \Downarrow_1 \Omega : v}$$

$$\text{(SU-Bind1)}\quad \frac{\Gamma : z_1 \Downarrow_1 \Delta : x \quad \Delta : z_2 \Downarrow_1 \Theta : c(\overline{y_n}) \quad \Theta[x \mapsto c(\overline{x_n}), \overline{x_n \mapsto \text{free}}] : x_1 \texttt{ =:= } y_1 \texttt{ \& } \cdots \texttt{ \& } x_n \texttt{ =:= } y_n \Downarrow_1 \Omega : v}{\Gamma : \texttt{prim\_su}(z_1, z_2) \Downarrow_1 \Omega : v}$$
where $\Delta(x) = \text{free}$ and $\overline{x_n}$ fresh

$$\text{(SU-Bind2)}\quad \frac{\Gamma : z_1 \Downarrow_1 \Delta : c(\overline{x_n}) \quad \Delta : z_2 \Downarrow_1 \Theta : y \quad \Theta[y \mapsto c(\overline{y_n}), \overline{y_n \mapsto \text{free}}] : x_1 \texttt{ =:= } y_1 \texttt{ \& } \cdots \texttt{ \& } x_n \texttt{ =:= } y_n \Downarrow_1 \Omega : v}{\Gamma : \texttt{prim\_su}(z_1, z_2) \Downarrow_1 \Omega : v}$$
where $\Theta(y) = \text{free}$ and $\overline{y_n}$ fresh

What remains to be defined is the semantics of the constraint operators "&>" and "&". While the conditional operator "&>" is predefined by the equation

```
c &> e = case c of Success → e
```

the concurrent constraint conjunction "&" is evaluated by the following rule:

$$\text{(Amp)}\quad \frac{\Gamma : x_i \Downarrow_1 \Delta : \texttt{Success} \quad \Delta : x_{3-i} \Downarrow_1 \Theta : \texttt{Success}}{\Gamma : x_1 \texttt{ \& } x_2 \Downarrow_1 \Theta : \texttt{Success}} \quad \text{where } i \in \{1, 2\}$$

5.4.4 Functional Patterns

In Section 4.2.4, we presented the transformation of functional patterns into a call to the primitive function "=:<=", where the variables of the patterns are introduced as fresh logic variables. For instance, the function

```
last (_ ++ [x]) = x
```

is translated to

```
last xs | (_ ++ [x]) =:<= xs = x where x free
```

What remains is the formal definition of the semantics of the primitive operation "=:<=". In principle, this operation is evaluated similarly to strict unification, but with the difference that the second operand is evaluated *on demand*, and we thus call this operator the *lazy unification operator*. If the left operand evaluates to a logic variable, it is directly bound to the right operand instead of its value, so that the logic variables of functional patterns behave similarly to pattern variables. Informally, the evaluation of e_1 =:<= e_2 proceeds as follows [AH05]:

1. Evaluate e_1 to a head normal form h_1.
2. If h_1 is a logic variable, bind it to e_2.
3. If $h_1 = c(\overline{e'_n})$, evaluate e_2 to a head normal form h_2.

(a) If h_2 is a logic variable, bind h_2 to $c(\overline{y_n})$ where $\overline{y_n}$ are fresh variables and evaluate e'_1 `=:<=` y_1 `&` $\cdots$ `&` e'_n `=:<=` y_n (which reduces to `Success` for $n = 0$).

(b) If $h_2 = c(\overline{e''_n})$, evaluate e'_1 `=:<=` e''_1 `&` $\cdots$ `&` e'_n `=:<=` e''_n (which reduces to `Success` for $n = 0$).

(c) If h_1 and h_2 are rooted by different constructors, then evaluation fails.

Note that the head normal form of the second argument is only computed if the first argument evaluates to a constructor-rooted value, so that the algorithm can be adapted to the operational semantics of FlatCurry by means of the following rules:

(LU-Free)
$$\frac{\Gamma : z_1 \Downarrow_1 \Delta : x}{\Gamma : z_1 \texttt{ =:<= } z_2 \Downarrow_1 \Delta[x \mapsto z_2] : \texttt{Success}} \quad \text{where } \Delta(x) = \text{free}$$

(LU-ConsFree)
$$\frac{\Gamma : z_1 \Downarrow_1 \Delta : c(\overline{x_n}) \quad \Delta : z_2 \Downarrow_1 \Theta : y \qquad \Theta[y \mapsto c(\overline{y_n}), \overline{y_n \mapsto \text{free}}] : x_1 \texttt{ =:<= } y_1 \texttt{ \& } \cdots \texttt{ \& } x_n \texttt{ =:<= } y_n \Downarrow_1 \Omega : v}{\Gamma : z_1 \texttt{ =:<= } z_2 \Downarrow_1 \Omega : v}$$
$$\text{where } \Theta(y) = \text{free and } \overline{y_n} \text{ fresh}$$

(LU-ConsCons)
$$\frac{\Gamma : z_1 \Downarrow_1 \Delta : c(\overline{x_n}) \quad \Delta : z_2 \Downarrow_1 \Theta : c(\overline{y_n}) \qquad \Theta : x_1 \texttt{ =:<= } y_1 \texttt{ \& } \cdots \texttt{ \& } x_n \texttt{ =:<= } y_n \Downarrow_1 \Omega : v}{\Gamma : z_1 \texttt{ =:<= } z_2 \Downarrow_1 \Omega : v}$$

However, the conversion of logic variables to pattern variables introduces a subtle issue whenever a variable occurs more than once in the value of a functional pattern. According to the semantics of non-linear patterns described in Section 4.2.4, multiple occurrences of the *same* pattern variable denote a strict unification of the respective arguments. For instance, if we consider the functions

```
pair      x y       = (x, y)
fromPair (pair x x) = x
```

then the definition of `fromPair` is equivalent to

```
fromPair (x, x) = x
```

by the semantics of functional patterns, which in turn is equivalent to

```
fromPair (x, y) | x =:= y = x
```

by the semantics of non-linear patterns. Consequently, a strict unification has to be generated for every variable occurring more than once in the evaluated functional pattern. However, these strict unifications cannot be generated statically, since the occurrence of the variables depends on the evaluation of the functional pattern and thus is a dynamic property. For instance, if we consider the additional definitions

```
zero        _                 = 0
fromZeroPair (pair (zero x) x) = 0
```

then the function `fromZeroPair` is equivalent to

```
fromZeroPair (0, x) = 0
```

and no strict unification shall be performed. Hence, the generation of strict unifications must be handled dynamically, such that a logic variable is only strictly unified if it is lazily bound at least twice.

The original work on functional patterns [AH05] proposes as a possible solution the linearization of the recursive constraints generated in the third step of the above algorithm, in conjunction with the generation of strict equality constraints. The evaluation of these constraints is then postponed until both variables of the respective constraint have been lazily bound. Since this solution would require a considerable amount of bookkeeping, we will not provide a formalization of this idea and instead concentrate on *linear functional patterns* in the remainder of this thesis, i. e., functional patterns that evaluate to linear values. Note that this is a dynamic property which depends on the semantics of the operations occurring in a functional pattern, and thus has to be ensured by the user. However, this property can be statically approximated as, for instance, is done in the PAKCS system [HAB+15], and we will briefly discuss this point in Section 9.2.1.

5.5 Summary

Based on the formal definition of untyped FlatCurry programs, we presented an operational semantics for FlatCurry based to the version proposed by Albert et al. [AHH+05], but with a slightly different notation and with some adaptations previously described by Braßel [Bra11]. We then generalized this semantics to be applicable also for non-flat expressions, and extended the generalized version to cover additional language constructs such as primitive operations, higher-order application, strict unification, and linear functional patterns. The extensions concerning primitive operations in general, and the extensions for higher-order applications and strict unifications in particular have previously been described by Albert et al. [AHH+05], and the extension for linear functional patterns is an adaptation of the algorithm proposed by Antoy and Hanus [AH05].

Part III

Partial Evaluation

Chapter 6

Partial Evaluation based on Needed Narrowing

If I have seen further, it is by standing on the shoulders of giants.

Isaac Newton

In this chapter we present the generic framework of Alpuente, Falaschi, and Vidal [AFV98] for the partial evaluation of functional logic programs based on narrowing and provide the correctness results of its instantiation with the needed narrowing strategy [ALH+05]. In this context, we consider functional logic programs to be confluent and inductively sequential term rewriting systems. Although the usefulness of this framework for the partial evaluation of contemporary Curry programs is limited due to the missing consideration of non-deterministic operations, it provides the general notions and ideas for partial evaluation we will use in our later developments.

6.1 Narrowing-Driven Partial Evaluation

Partial evaluation of a program (term rewriting system) tries to anticipate the evaluation of certain functions at compile time, based on some known (static) arguments. As its result, it produces a *residual program*, i. e., a specialized version of the original program (term rewriting system).

Example 6.1 (Partial Evaluation)**.** *Consider the TRS*

```
[]       ++ ys = ys
(x : xs) ++ ys = x : (xs ++ ys)
```

defining the list concatenation and the term `(xs ++ ys) ++ zs`*. Note that the calls to* `(++)` *associate to the left, such that the intermediate list* `(xs ++ ys)` *has to be traversed to compute the final list, and in effect the list* `xs` *is traversed twice. If we partially evaluate this term with respect to the TRS, we can obtain the specialized TRS*

```
app  []       ys    = ys
app  (x : xs) ys    = x : (app xs ys)
dapp []       ys zs = app ys zs
dapp (x : xs) ys zs = x : (dapp xs ys zs)
```

where calls to `(xs ++ ys) ++ zs` *should now be replaced by calls to* `dapp xs ys zs`*. In the definition of* `dapp`*, the intermediate list has been removed, so that the first list is only traversed once. Hence, the specialized TRS will be executed more efficiently than the original TRS.*

Generally, a partial evaluator can be built by following the structure of an interpreter [Ses85], and interestingly the partial evaluation of functional logic programs represented as term rewriting system can be obtained by an application of narrowing at partial evaluation time [AFV98]. If a function is called with unknown arguments, narrowing instantiates these arguments, such that the rules defining this function can be applied. Hence, it basically suffices to control the partial evaluator, i. e., to avoid infinite evaluation sequences of functions as well as infinite instantiations of logic variables, in order to obtain residual term rewriting systems.

In general, narrowing-driven partial evaluation follows the ideas of partial deduction [LS91], and has been adapted to the setting of functional logic languages. Given a TRS $\mathcal{R}$ and a set of terms S, the aim of partial evaluation is then to compute a new TRS $\mathcal{R}'$ which computes the same set of answers as $\mathcal{R}$ for any input term that is an instance of some term in S. The TRS $\mathcal{R}'$ is obtained by collecting a set of *resultants* for each term in S, which are constructed as follows.

Definition 6.2 (Resultant [AFV98; ALH+05])**.** *Let $\mathcal{R}$ be a TRS and t be a term. Given a narrowing derivation $t \leadsto_\sigma^+ t'$, its associated resultant is the rewrite rule $\sigma(t) \to t'$.*

Note that if the term t does not form a linear pattern, the left-hand side $\sigma(t)$ of the resultant may not be linear either, so that the computed resultants may not establish valid rewrite rules, as is demonstrated by the following example.

Example 6.3. *[AHL+04] Consider the following inductively sequential TRS:*

```
double  n = n + n
Z     + n = n
(S m) + n = S (m + n)
```

Given the term `(double w) + w` *and the needed narrowing derivation*

$$\underline{\texttt{double w}}\ \texttt{+ w} \leadsto_{id} \underline{\texttt{(w + w)}}\ \texttt{+ w} \leadsto_{\{\texttt{w} \mapsto \texttt{S m}\}} \texttt{S (m + S m) + S m}$$

where the selected redex is underlined for each step, the associated resultant is

$$\texttt{double (S m) + S m} \to \texttt{S (m + S m) + S m}$$

This resultant is not a legal rewrite rule, since it is neither left-linear nor constructor-based (there are multiple occurrences of `m` *and nested function symbols in the left-hand side).*

To solve this problem, a later post-processing phase of *renaming* is applied, which also may eliminate redundant structures. Furthermore, this renaming phase ensures the *independence* of the specializations, such that different specializations for the same function definition are correctly distinguished, which is crucial for a polyvariant specialization.

The *pre-partial evaluation* of a term t is then obtained by the construction of a (possibly incomplete) narrowing tree for t and the extraction of the specialized definition as the non-failing root-to-leaf paths of the narrowing tree.

Definition 6.4 (Pre-Partial Evaluation [AFV98; ALH+05])**.** *Let $\mathcal{R}$ be a TRS and t a term. Let T be a (possibly incomplete) narrowing tree for t in $\mathcal{R}$ such that no constructor-rooted term in the tree has been narrowed. Let $\overline{t_n}$ be the terms in the non-failing leaves of T. Then the set of resultants $\{\sigma_i(t) \to t_i \mid i \in 1, \ldots, n\}$ for the narrowing sequences $\{t \leadsto_{\sigma_i}^{+} t_i \mid i \in 1, \ldots, n\}$ is called a pre-partial evaluation of t in $\mathcal{R}$. The pre-partial evaluation of a set of terms S in $\mathcal{R}$ is defined as the union of the pre-partial evaluations for $s \in S$ in $\mathcal{R}$.*

For a given TRS $\mathcal{R}$ and a term t, there may exist an infinite number of pre-partial evaluations of t in $\mathcal{R}$, due to the variance in the applied narrowing strategy and in the length of the derivation for each resultant. We therefore assume a fixed rule for the creation of resultants, called an *unfolding rule*, that determines the terms to be narrowed by a fixed narrowing strategy and decides how to stop the construction of a narrowing tree to ensure its finiteness (a formal definition of unfolding rules and a concrete example will be given in Section 6.2).

Example 6.5 (Pre-Partial Evaluation)**.** *Consider the following TRS $\mathcal{R}$*

```
[]       ++ ys = ys
(x : xs) ++ ys = x : (xs ++ ys)
```

implementing the list concatenation and the set of calls $S = \{$`(xs ++ ys) ++ zs`, `xs ++ ys`$\}$. For these calls, an associated pre-partial evaluation $\mathcal{R}'$ of S in $\mathcal{R}$ might be

```
   ([] ++ ys) ++ zs → ys ++ zs
((x : xs) ++ ys) ++ zs → x : ((xs ++ ys) ++ zs)
          [] ++ ys → ys
    (x : xs) ++ ys → x : (xs ++ ys)
```

Note that the restriction to not evaluate subterms under constructor symbols is necessary to ensure the correctness of the specialized TRS for a non-strict semantics. Without this restriction, the specialized TRS might lose completeness, as the following example demonstrates.

Example 6.6 (Loss of Completeness [AHV00])**.** *Consider the TRS*

```
isZero  Z        = True
test    x        = (isZero x) : []
notNull (x : xs) = True
```

and the term `test y` *with the (unique) derivation*

$$\texttt{test y} \leadsto_{id} \texttt{(isZero y) : []} \leadsto_{\{\texttt{y} \mapsto \texttt{Z}\}} \texttt{True : []}$$

where the narrowed subterm `(isZero y)` *occurs below a constructor. For this derivation, the residual rule would be*

$$\texttt{test Z} \to \texttt{True : []}$$

Now the term `notNull (test (S Z))` *can be evaluated to* `True` *in the original TRS by*

$$\texttt{notNull (test (S Z))} \leadsto_{id} \texttt{notNull ((isZero (S Z)) : [])} \leadsto_{id} \texttt{True}$$

which is no longer possible using the residual rule.

In consequence, a function application occurring below a constructor symbol in a narrowing derivation must not be evaluated during partial evaluation. Nevertheless, it must be guaranteed that each function call that might occur during the evaluation of the specialized TRS is covered by some rewrite rule. This is ensured by a so-called *closedness* condition, which is recursively checked for all terms in the specialized TRS. Intuitively, a term t rooted by a defined operation symbol is closed with respect to a set of terms S if and only if it is an instance of some term $s \in S$, and the terms in the substitution are recursively closed with respect to S. We furthermore consider a fixed set $\mathcal{P} \subseteq \mathcal{F}$ of *primitive operation* symbols, i. e., operations which are not explicitly defined by rewrite rules, such as integer arithmetics. These primitives are expected to be automatically added to the specialized TRS.

Definition 6.7 (Closedness [AFV98; ALH+05])**.** *Let S be a finite set of terms. We say that a term t is closed with respect to S, or S-closed, if the predicate* $\mathsf{closed}(S, t)$ *holds, which is inductively defined as:*

$$\mathsf{closed}(S,t) :\Leftrightarrow \begin{cases} \text{true} & \text{if } t \in \mathcal{V} \\ \bigwedge_{i=1}^{n} \mathsf{closed}(S, t_i) & \text{if } t = c(\overline{t_n}),\ c \in \mathcal{C} \cup \mathcal{P},\ n \geqslant 0 \\ \bigwedge_{t' \in \mathcal{R}an(\theta)} \mathsf{closed}(S, t') & \text{if } \exists \theta,\ \exists s \in S \text{ such that } t = \theta(s) \end{cases}$$

We say that a set of terms T is S-closed, written $\mathsf{closed}(S, T)$*, if* $\mathsf{closed}(S, t)$ *holds for all $t \in T$, and we say that a TRS $\mathcal{R}$ is S-closed if all right-hand sides of $\mathcal{R}$ are S-closed.*

By this definition, variables are always closed, while terms rooted by a non-primitive operation are closed only if they are an instance of some term in S, and the range of the corresponding substitution is recursively closed. Finally, terms rooted by a constructor or primitive function symbol can either be closed if their arguments are closed, or if they are an instance of some term in S and the terms in the substitution are recursively closed. For instance, the pre-partial evaluation of Example 6.5 is closed with respect to the set of the partially evaluated calls.

After the computation of the pre-partial evaluation, the post-processing step of *renaming* is applied to ensure both the left-linearity and constructor-basedness of the rewrite rules, as well as to supply distinct root symbols for the specialization of different initial terms. This is ensured by introducing a fresh symbol for every specialized term, and by replacing each call in the specialized TRS by a call to the corresponding renamed function.

Definition 6.8 (Independent Renaming [AFI+97; ALH+05]). *Let $\mathcal{R}$ be a TRS. Then an independent renaming ρ for a set of terms S (w.r.t. $\mathcal{R}$) is a mapping from terms to terms defined as follows: for every $s \in S$, we have $\rho(s) = f_s(\overline{x_n})$ such that $\overline{x_n}$ are the distinct variables of s in the left-to-right ordering, and f_s is a new function symbol which does not occur in $\mathcal{R}$ or S and is different from the root symbol of any other $\rho(s')$ with $s' \in S$ and $s' \neq s$. We also denote by $\rho(S)$ the set $S' = \{\rho(s) \mid s \in S\}$.*

Example 6.9 (Independent Renaming). *Consider again the set of terms*

$$S = \{\texttt{(xs ++ ys) ++ zs}, \texttt{xs ++ ys}\}$$

of Example 6.5. For this set, the following mapping is an independent renaming:

$$\texttt{(xs ++ ys) ++ zs} \mapsto \texttt{dapp xs ys zs}$$
$$\texttt{xs ++ ys} \mapsto \texttt{app xs ys}$$

While independent renamings suffice to rename the left-hand sides of resultants, the right-hand sides have to be renamed *recursively* to meet the closedness condition. This is achieved by means of the auxiliary function ren_ρ, which replaces function calls in the given term by a call to the corresponding renamed function, according to a (fixed) independent renaming ρ.

Definition 6.10 (Renaming Function [AFI+97; ALH+05]). *Let S be a finite set of terms and ρ an independent renaming for S. Given a term t, the non-deterministic renaming function ren_ρ is defined as follows:*

$$\mathsf{ren}_\rho(t) = \begin{cases} t & \text{if } t \in \mathcal{V} \\ c(\overline{\mathsf{ren}_\rho(t_n)}) & \text{if } t = c(\overline{t_n}),\ c \in \mathcal{C} \cup \mathcal{P},\ n \geqslant 0 \\ \theta'(\rho(s)) & \text{if } \exists \theta,\ \exists s \in S \text{ such that } t = \theta(s) \text{ and} \\ & \theta' = \{x \mapsto \mathsf{ren}_\rho(\theta(x)) \mid x \in \mathcal{D}om(\theta)\} \\ t & \text{otherwise} \end{cases}$$

In analogy to the definition of closedness, the application of primitive function or constructor symbols can either be renamed to a call of a residual function, or the arguments are recursively renamed instead. Furthermore, the function returns an operation-rooted term t that is not an instance of some term in S without modification, so that ren_ρ is a total function.

Based on the definitions of pre-partial evaluations and independent renamings, the notion of a partial evaluation of a set of terms can be defined as follows.

Definition 6.11 (Partial Evaluation [AFV98; ALH+05]). *Let $\mathcal{R}$ be a TRS, S a finite set of terms, $\mathcal{R}'$ a pre-partial evaluation of S in $\mathcal{R}$, and ρ an independent renaming for S. We define the partial evaluation $\mathcal{R}''$ of S in $\mathcal{R}$ (under ρ) as follows:*

$$\mathcal{R}'' = \bigcup_{s \in S} \{\theta(\rho(s)) \to \mathsf{ren}_\rho(t) \mid \theta(s) \to t \in \mathcal{R}' \text{ is a resultant for } s \text{ in } \mathcal{R}\}$$

To obtain a specialized TRS based on a pre-partial evaluation, the left-hand side is renamed to obtain left-linearity as well as constructor-basedness, and the right-hand side is recursively renamed to obtain closedness.

Example 6.12 (Partial Evaluation)**.** *We consider once more the TRS $\mathcal{R}$, set of terms S and pre-partial evaluation $\mathcal{R}'$ of Example 6.5, together with the independent renaming ρ of Example 6.9. We can then obtain the following partial evaluation $\mathcal{R}''$ of S in $\mathcal{R}$ (under ρ):*

```
dapp []       ys zs = app ys zs
dapp (x : xs) ys zs = x : (dapp xs ys zs)
app  []       ys    = ys
app  (x : xs) ys    = x : (app xs ys)
```

The quality of the renaming process for a term t generally depends on the strategy that selects the term s such that $t = \theta(s)$. If there exists more than one matching term, then different renamings could be applied. For instance, in the example above the term `x : ((xs ++ ys) ++ zs)` can be renamed both to `dapp xs ys zs` or `app (app xs ys) ys`, but some potential for specialization is lost in the latter case. An ad-hoc heuristic is to prefer a term s such that $t = \theta(s)$ and θ is a variable renaming, so that the full potential of specialization is preserved, but more sophisticated heuristics may be useful in general.

The properties of partial evaluation based on narrowing naturally depend on the considered unfolding rule, and furthermore on the chosen narrowing strategy. In the following, we will assume an unfolding rule based on needed narrowing, and call a partial evaluation computed using needed narrowing a *partial needed narrowing evaluation* (*NN-PE*). An important property of partial evaluation based on needed narrowing is the preservation of inductively sequentiality, i. e., if the input TRS is inductively sequential, then so is the specialized TRS.

Theorem 6.13 (Inductively Sequential Specialization [ALH+05])**.** *Let $\mathcal{R}$ be an inductively sequential TRS and S a finite set of operation-rooted terms. Then each NN-PE of S in $\mathcal{R}$ is inductively sequential.*

Another favorable property of NN-PE is that it does not introduce additional non-determinism in the derivation of a term that can be deterministically narrowed to its result. This property is especially desirable from an implementation point of view, since the implementation of non-deterministic narrowing steps may be an expensive operation. Furthermore, additional non-deterministic steps may lead to additional infinite derivations in the residual TRS, so that the result may no longer be computable in case of a non-complete search strategy (e. g., depth-first search).

Theorem 6.14 (Deterministic Evaluation [ALH+05])**.** *Let $\mathcal{R}$ be an inductively sequential TRS, S a finite set of operation-rooted terms, ρ an independent renaming for S, and e an equation. Let $\mathcal{R}'$ be a NN-PE of S in $\mathcal{R}$ (under ρ) such that $\mathcal{R}'$ and e' are S'-closed, where $e' = \mathsf{ren}_\rho(e)$ and $S' = \rho(S)$. If e deterministically normalizes to* `Success` *in $\mathcal{R}$, then e' deterministically normalizes to* `Success` *in $\mathcal{R}'$.*

Finally, NN-PE satisfies the property of strong correctness, which states the computational equivalence of the original and the specialized TRSs, i. e., the sets of computed answers coincide.

Theorem 6.15 (Strong Correctness [ALH+05])**.** *Let $\mathcal{R}$ be an inductively sequential TRS, e an equation, $V \supseteq \mathcal{V}ar(e)$ a finite set of variables, S a finite set of operation-rooted terms, and ρ an independent renaming for S. Let $\mathcal{R}'$ be a NN-PE of S in $\mathcal{R}$ (under ρ) such that $\mathcal{R}'$ and e' are S'-closed, where $e' = \mathsf{ren}_\rho(e)$ and $S' = \rho(S)$. Then, $e \rightsquigarrow^*_\sigma$ `Success` is a needed narrowing derivation for e in $\mathcal{R}$ if and only if there exists a needed narrowing derivation $e' \rightsquigarrow^*_{\sigma'}$ `Success` in $\mathcal{R}'$ such that $\sigma' = \sigma[V]$.*

6.2 Partial Evaluation Procedure

In the previous section, we have presented the basic principles of partial evaluation based on narrowing, and stated its correctness for the needed narrowing strategy under the assumption of a closed and finite set of operation-rooted terms. However, we did not discuss how such a set can be effectively computed based on an initial goal (term or equation) to be partially evaluated.

Informally, for a given goal g and TRS $\mathcal{R}$, at first the set S of function calls appearing in g is determined, and the pre-partial evaluation for S in $\mathcal{R}$ is computed. This process is then repeated for any term occurring in the right-hand sides of the residual TRS which is not closed with respect to the set of terms already evaluated. Assuming that this repetition eventually terminates, we obtain a final set of partially evaluated terms S' and a pre-partial evaluation $\mathcal{R}'$ such that S'-closedness is obtained for $\mathcal{R}'$ and g.

Example 6.16 (Partial Evaluation Procedure)**.** *Consider the following TRS $\mathcal{R}$*

```
Z     + n = n
(S m) + n = S (m + n)
```

and the initial goal $g =$ `S (S Z)) + x`*. If we consider narrowing trees that perform only one unfolding step and start with the initial set of terms $S = \{$*`S (S Z)) + x`$\}$*, we get the following derivation:*

$$\texttt{(S (S Z)) + x} \rightsquigarrow_{\{\texttt{m} \mapsto \texttt{S Z}\}} \texttt{S ((S Z) + x)}$$

We then select the operation-rooted term `(S Z) + x` *for further evaluation, which is not closed with respect to S, and get the derivation*

$$\texttt{(S Z) + x} \rightsquigarrow_{\{\texttt{m}' \mapsto \texttt{Z}\}} \texttt{S (Z + x)}$$

Once again, the term `Z + x` *is not closed with respect to the set $\{$*`S (S Z)) + x, (S Z) + x`$\}$*, and we get the final derivation*

$$\texttt{Z + x} \rightsquigarrow_{id} \texttt{x}$$

In summary, we have computed the following pre-partial evaluation $\mathcal{R}'$

```
(S (S Z)) + x → S ((S Z) + x)
(S Z) + x → S (Z + x)
Z + x → x
```

for the set of terms $S' = \{$`(S (S Z)) + x, (S Z) + x, Z + x`$\}$*, and both* $\mathcal{R}'$ *and the initial goal* g *are* S'*-closed. After the post-processing of renaming, we obtain the following partial evaluation* $\mathcal{R}''$ *of* S' *in* $\mathcal{R}$*, where* $\mathsf{ren}_\rho(g) = \mathsf{ren}_\rho($`(S (S Z)) + x`$) =$ `add2 x`*:*

```
add2 x = S (add1 x)
add1 x = S (add0 x)
add0 x = x
```

To ensure the correctness of this iterative process, both partial correctness of the computed partial evaluation and termination of the process have to be addressed. While the aspect of partial correctness has been considered in the previous section, termination has not, and we can identify two different problems for termination based on the procedure outlined above. Firstly, the termination of unfolding has to be ensured, i. e., the construction of narrowing trees must be finite (*local termination*). Secondly, the global iteration for unevaluated subterms has to terminate as well (*global termination*), which requires the finiteness of the set of terms to be evaluated. In addition, this set of terms also has to satisfy the closedness property, as this is required for partial correctness. Since these requirements can be fulfilled in different ways, the partial evaluation scheme is generic with respect to the following three components:

1. a *narrowing strategy* applied for the construction of derivations,
2. an *unfolding rule* used to construct a finite narrowing tree for a given term, and
3. an *abstraction operator* used to guarantee that the set of terms obtained during partial evaluation is finite and satisfies the closedness property.

In the following, by $\overset{\varphi}{\leadsto}$ we denote a generic narrowing relation applying the narrowing strategy φ. Based on this notion, the construction of a (possibly incomplete) narrowing tree and the extraction of the resultants is formalized by an unfolding rule.

Definition 6.17 (Unfolding Rule [AFV98])**.** *An unfolding rule* $U_\varphi(t, \mathcal{R})$ *(or simply* U_φ*) is a mapping which returns a pre-partial evaluation (a set of resultants) for the term* t *in the TRS* $\mathcal{R}$ *using the narrowing relation* $\overset{\varphi}{\leadsto}$*. Given a set of terms* S*, by* $U_\varphi(S, \mathcal{R})$ *we denote the union of the sets* $U_\varphi(s, \mathcal{R})$ *for all* $s \in S$*.*

To ensure both finiteness and closedness of the set of terms obtained during partial evaluation, an abstraction operator is used. This operator considers a sequence[1] of

[1] The original work [AFV98] considers arbitrary configurations over a set of terms in conjunction with a state transition system, but sequences suffice for our purposes.

terms which have already been evaluated, and adds new terms to this sequence only if the new terms are less than or equal to the terms in the sequence according to some suitable ordering. More precisely, if S_q denotes the set of terms in a sequence of terms q, then the abstraction operator takes a sequence of terms q and a set of terms T to be added and computes a sequence q' as a "safe" approximation of $S_q \cup T$, such that all terms in q' are closed with respect to $S_q \cup T$.

Definition 6.18 (Abstraction Operator [AFV98])**.** *Let q be a sequence of operation-rooted terms and T a finite set of terms, and let S_q denote the set of terms in q. An abstraction operator* abstract *is a mapping which returns a new sequence $q' = \mathsf{abstract}(q, T)$ such that*

1. *if $t' \in S_{q'}$, then t' is operation-rooted, and*
2. *if $t' \in S_{q'}$, then there exists a term $t \in (S_q \cup T)$ such that $t|_p = \sigma(t')$ for some non-variable position p and substitution σ, and*
3. *for all $t \in (S_q \cup T)$, t is closed with respect to $S_{q'}$.*

The first condition, which has not been included in the original work [AFV98], ensures that only *operation-rooted* terms are considered for partial evaluation, which is necessary for the partial correctness results to be applicable. In addition, the second condition ensures that no new operation-symbols are "invented" during abstraction, and the third condition is used to ensure the correct propagation of closedness.

Based on the definitions of an unfolding rule and an abstraction operator, we can define the central transition relation between two subsequent states of the partial evaluation process.

Definition 6.19 (Partial Evaluation Transition Relation [AFV98])**.** *We define the partial evaluation relation $\longmapsto_{\mathcal{PE}}$ as the smallest relation satisfying*

$$\frac{\mathcal{R}' = U_\varphi(S_q, \mathcal{R})}{q \longmapsto_{\mathcal{PE}} \mathsf{abstract}(q, \{r \mid l \to r \in \mathcal{R}'\})}$$

Thus, the transition from an partial evaluation state represented by the sequence of terms q is computed in two steps. Firstly, the unfolding rule U_φ is applied to every term in the sequence q to compute the pre-partial evaluation $\mathcal{R}'$. Secondly, all terms in the right-hand sides of $\mathcal{R}'$ are added to the sequence by means of the operation abstract, which allows the control of polygenetic specializations (i. e., by elimination of variants) and ensures progress towards termination. The initial sequence of terms is computed by abstraction of an empty sequence and the initial goal, and the transition relation is repeatedly applied until a fixpoint is reached.

Definition 6.20 (Partial Evaluation Algorithm [AFV98])**.** *Let $\mathcal{R}$ be a TRS and g a goal. We define the partial evaluation function $\mathcal{PE}$ as follows:*

$$\mathcal{PE}(\mathcal{R}, g) = S_q \quad \textit{if } \mathsf{abstract}(\varepsilon, \{g\}) \longmapsto^*_{\mathcal{PE}} q \textit{ and } q \longmapsto_{\mathcal{PE}} q$$

where ε denotes the empty sequence of terms.

The main property of the partial evaluation algorithm is the fulfillment of the closedness property, which is stated in the following theorem.

Theorem 6.21 (Partial Correctness of the Result of $\mathcal{PE}$ [AFV98])**.** *Let $\mathcal{R}$ be a TRS and g a goal. If $\mathcal{PE}(\mathcal{R}, g)$ terminates computing the sequence q of terms, then $\mathcal{R}'$ and g are S_q-closed, where $\mathcal{R}' = U_\phi(S_q, \mathcal{R})$ is the pre-partial evaluation of S_q in $\mathcal{R}$.*

Note that this algorithm does not compute a residual TRS, but only a sequence of terms such that their pre-partial evaluation is closed. The residual TRS is then constructed from S_q according to Definition 6.11. Since the application of renaming preserves the closedness condition [AFI+97], it follows that the specialized TRS $\mathcal{R}''$ is closed with respect to $\rho(S_q)$ for an independent renaming ρ for S_q. Thus, in conjunction with Theorem 6.15, this theorem establishes the partial correctness of the partial evaluation scheme based on needed narrowing.

6.3 Termination

To ensure termination of the partial evaluation process, we have to provide an unfolding rule as well as an abstraction operator which eventually terminate. For this purpose, the framework of Alpuente, Falaschi, and Vidal [AFV98] adapts well-known techniques from the area of partial deduction and (positive) supercompilation. For *local* termination, there already exist commonly used approaches such as loop-checks, depth-bounds, or the usage of well-founded orderings [BSM92], and the approach of Sørensen and Glück [SG95] to use a *well-quasi ordering* is applied to ensure termination of the construction of narrowing trees. At the *global* level of termination, the same well-quasi ordering as for local termination is used, which can be seen as a simplified version of the approach of Martens and Gallagher [MG95], who used a tree-like structure instead of sequences to increase the degree of specialization.

Orderings are commonly used to restrict the evaluation of terms to those that are "syntactically simpler" than some other term already evaluated. If there are only finitely many simpler terms, then this process will eventually terminate. A commonly used ordering is the *homeomorphic embedding*, which has been successfully employed for termination of term rewriting systems [BN98], as well as in the fields of partial deduction [GJM+96; MG95; LMS98; Leu98] and supercompilation [Tur96].

Definition 6.22 (Pure Homeomorphic Embedding Relation [AFV98])**.** *The pure homeomorphic embedding relation $\trianglelefteq$ on terms in $\mathcal{T}(\Sigma, \mathcal{V})$ is defined as the smallest relation satisfying the following conditions:*

1. $x \trianglelefteq y$ *for all* $x, y \in \mathcal{V}$,
2. $s \trianglelefteq \phi(\overline{t_n})$ *if* $s \trianglelefteq t_i$ *for some* $i \in \{1, \ldots, n\}$,
3. $\phi(\overline{s_n}) \trianglelefteq \phi(\overline{t_n})$ *if* $s_i \trianglelefteq t_i$ *for all* $i \in \{1, \ldots, n\}$.

Intuitively, a term s is embedded in another term t if s can be obtained from t by deletion of some constructors or operators. For example, `rev zs (1:[])` $\trianglelefteq$ `rev xs (x:1:[])` holds by the third condition, since `zs` $\trianglelefteq$ `xs` holds by the first and `1:[]` $\trianglelefteq$ `x:1:[]` by the second condition (and recursively by the third).

Theorem 6.23 ($\trianglelefteq$ is a Well-Quasi Ordering [AFV98])**.** *The relation $\trianglelefteq$ is a well-quasi ordering of the set $\mathcal{T}(\Sigma, \mathcal{V})$ for finite Σ, that is, $\trianglelefteq$ is a quasi-order (a transitive and reflexive relation) and, for any infinite sequence of terms $t_1, t_2, \ldots$ with a finite number of operators, there exist j, k with $j < k$ and $t_j \trianglelefteq t_k$ (the sequence is self-embedding).*

The homeomorphic embedding relation is used to guarantee both local and global termination, and we will give a definition of an unfolding rule and abstraction operator based on this ordering in the following.

6.3.1 Local Termination

The unfolding rule tries to maximize the number of unfolding steps while retaining finiteness of the narrowing tree. The usage of the embedding ordering is less ad-hoc than fixed criteria such as depth-bounds, but still sufficiently simple. In order to avoid an infinite sequence of terms in a narrowing derivation, each narrowing redex is compared with its ancestor redexes in the same derivation, and the derivation is continued only if the new redex does not embed any of its ancestor redexes. If it does embed an ancestor redex, there is a risk of non-termination, and the derivation is stopped. We will formalize this criterion based on the following notions.

Definition 6.24 (Comparable Terms [AFV98])**.** *Let s and t be terms. We say that s and t are comparable, written* $\mathsf{comparable}(s, t)$*, if and only if the outermost function symbols of s and t coincide.*

The notion of comparable terms effectively allows the distinction of different function symbols with respect to termination. Based on this distinction, we can characterize which narrowing derivations are admissible.

Definition 6.25 (Admissible Derivation [AFV98])**.** *Let $\mathcal{R}$ be a TRS and $\mathcal{D}$ be a narrowing derivation $t_0 \overset{\varphi}{\leadsto}_{p_0,\theta_0} \ldots \overset{\varphi}{\leadsto}_{p_{n-1},\theta_{n-1}} t_n$ in $\mathcal{R}$. We say that $\mathcal{D}$ is admissible if and only if it does not contain a pair of comparable redexes included in the embedding relation $\trianglelefteq$. Formally, we define*

$$\mathsf{admissible}(\mathcal{D}) :\Leftrightarrow \forall i \in \{1, \ldots, n\}, \forall (p, R, \sigma) \in \varphi(t_i), \forall j \in \{0, \ldots, i-1\} : \\ \mathsf{comparable}(t_j|_{p_j}, t_i|_p) \implies t_j|_{p_j} \not\trianglelefteq t_i|_p.$$

Based on these two definitions, we can proceed to the definition of a non-embedding narrowing tree which contains only admissible derivations.

Definition 6.26 (Non-Embedding Narrowing Tree [AFV98]). *Given a term t_0 and a TRS $\mathcal{R}$, we define the non-embedding narrowing tree $\mathsf{T}^{\trianglelefteq}_{\varphi}(t_0, \mathcal{R})$ for t_0 in $\mathcal{R}$ as follows:*

$$\mathcal{D} = t_0 \overset{\varphi}{\leadsto}_{p_0,\theta_0} \ldots \overset{\varphi}{\leadsto}_{p_{n-1},\theta_{n-1}} t_n \overset{\varphi}{\leadsto}_{p_n,\theta_n} t_{n+1} \in \mathsf{T}^{\trianglelefteq}_{\varphi}(t_0, \mathcal{R})$$

if the following conditions hold:

1. *the derivation $t_0 \overset{\varphi}{\leadsto}_{p_0,\theta_0} \ldots \overset{\varphi}{\leadsto}_{p_{n-1},\theta_{n-1}} t_n$ is admissible, and*
2. *(a) the leaf t_{n+1} is a head normal form ($\mathcal{D}$ is a successful derivation), or*
   (b) the leaf t_{n+1} is failed ($\mathcal{D}$ is a failing derivation), or
   (c) there exist a triple $(p, R, \sigma) \in \varphi(t_{n+1})$ and a number $i \in \{1, \ldots, n\}$ such that $t_i|_{p_i}$ and $t_{n+1}|_p$ are comparable and $t_i|_{p_i} \trianglelefteq t_{n+1}|_p$ ($\mathcal{D}$ is incomplete and is cut off because there exists a risk of non-termination).

In consequence, the derivation will be stopped if the respective branch either fails, succeeds, or the term under consideration embeds some other term previously evaluated in the respective derivation.

Example 6.27 (Non-Embedding Narrowing Tree). *Consider the following TRS $\mathcal{R}$ reversing a list*

```
reverse      xs = rev xs []
rev []       ys = ys
rev (x : xs) ys = rev xs (x : ys)
```

and the term `reverse (1 : zs)`*, for which exists the following infinite narrowing derivation:*

$$\begin{array}{lll}
\texttt{reverse (1 : zs)} & \leadsto_{\{\texttt{xs}\mapsto\texttt{1 : zs}\}} & \texttt{rev (1 : zs) []} \\
 & \leadsto_{\{\texttt{x}_1\mapsto\texttt{1},\ \texttt{xs}_1\mapsto\texttt{zs},\ \texttt{ys}_1\mapsto\texttt{[]}\}} & \texttt{rev zs (1 : [])} \\
 & \leadsto_{\{\texttt{zs}\mapsto\texttt{x}_2\ :\ \texttt{xs}_2,\ \texttt{ys}_2\mapsto\texttt{1 : []}\}} & \texttt{rev xs}_2\ \texttt{(x}_2\ \texttt{: 1 : [])} \\
 & \leadsto \ldots & \ldots
\end{array}$$

According to the definition of the non-embedding narrowing tree, the development of this derivation is stopped at the fourth term, since the derivation

$$\begin{array}{lll}
\texttt{reverse (1 : zs)} & \leadsto_{\{\texttt{xs}\mapsto\texttt{1 : zs}\}} & \texttt{rev (1 : zs) []} \\
 & \leadsto_{\{\texttt{x}_1\mapsto\texttt{1},\ \texttt{xs}_1\mapsto\texttt{zs},\ \texttt{ys}_1\mapsto\texttt{[]}\}} & \texttt{rev zs (1 : [])}
\end{array}$$

is admissible, but the next step

$$\texttt{rev zs (1 : [])} \leadsto_{\{\texttt{zs}\mapsto\texttt{x}_2\ :\ \texttt{xs}_2,\ \texttt{ys}_2\mapsto\texttt{1 : []}\}} \texttt{rev xs}_2\ \texttt{(x}_2\ \texttt{: 1 : [])}$$

fulfills the embedding ordering since `rev zs (1 : [])` $\trianglelefteq$ `rev xs`$_2$ `(x`$_2$ `: 1 : [])`.

The notion of non-embedding narrowing trees can then be applied to ensure the finiteness of constructed narrowing trees, as is stated in the following theorem.

Theorem 6.28 (Finiteness of Non-Embedding Narrowing Tree [AFV98]). *For a TRS $\mathcal{R}$ and a term t, $\mathsf{T}^{\trianglelefteq}_{\varphi}(t, \mathcal{R})$ is a finite (possibly incomplete) narrowing tree for t in $\mathcal{R}$ using $\overset{\varphi}{\leadsto}$.*

Therefore, it is reasonable to construct an unfolding rule based on the concept of a non-embedding narrowing tree, since termination of the unfolding rule follows from the finiteness of the constructed non-embedding narrowing tree.

Definition 6.29 (Non-Embedding Unfolding Rule [AFV98])**.** *We define* $\mathcal{U}^{\trianglelefteq}_{\varphi}(s,\mathcal{R})$ *as the set of resultants associated with the derivations in* $\mathsf{T}^{\trianglelefteq}_{\varphi}(s,\mathcal{R})$*.*

6.3.2 Global Termination

The usage of the homeomorphic embedding ordering can also be transferred to the problem of global termination, i. e., to ensure the finiteness of the set of terms to be evaluated. In addition to finiteness, the set of terms also has to meet the criterion of closedness, so that terms not contained in the set cannot be simply neglected. Therefore, an abstraction operator based on the concept of *most specific generalizations* is used to ensure both finiteness and completeness of the set of terms.

Definition 6.30 (Most-Specific Generalization)**.** *A generalization of two terms* t_0 *and* t_1 *is a triple* (t,σ,θ) *such that* $t_0 = \sigma(t)$ *and* $t_1 = \theta(t)$*. The triple* (t,σ,θ) *is the most specific generalization (msg) of two terms* t_0 *and* t_1*, written* $\mathsf{msg}(t_0,t_1)$*, if*

1. (t,σ,θ) *is a generalization of* t_0 *and* t_1*, and*

2. *for every other generalization* (t',σ',θ') *of* t_0 *and* t_1*,* t' *is more general than* t*.*

The msg of two terms is unique up to variable renaming [LMM88], and we can use this notion to define the *non-embedding abstraction operator*.

Definition 6.31 (Non-Embedding Abstraction Operator [AFV98])**.** *Let* q *be a finite sequence of terms and* T *be a finite set of terms. We define the non-embedding abstraction operator* $\mathsf{abstract}^*(q,T)$ *inductively as follows:*

$$\mathsf{abstract}^*(q,\{t_1,\ldots,t_n\}) = \begin{cases} q & \text{if } n = 0 \\ \mathsf{abstract}^*(\ldots\mathsf{abstract}^*(q,t_1),\ldots,t_n) & \text{if } n \geqslant 1 \end{cases}$$

where the abstraction of a single term is defined as

$$\mathsf{abstract}^*(q,t) = \begin{cases} q & \text{if } t \in \mathcal{V} \\ \mathsf{abstract}^*(q,\{t_1,\ldots,t_n\}) & \text{if } t = c(\overline{t_n}),\ c \in \mathcal{C},\ n \geqslant 0 \\ \mathsf{absCall}(q,t) & \text{if } t = f(\overline{t_n}),\ f \in \mathcal{F},\ n \geqslant 0 \end{cases}$$

The auxiliary function absCall *responsible for the addition of an operation-rooted term is*

defined as

$$\mathsf{absCall}(q,t) = \begin{cases} q \cdot t & \text{if } \nexists s \in S_q : \mathsf{comparable}(s,t) \wedge s \trianglelefteq t \\ \mathsf{abstract}^*(q,S) & \text{if } i \text{ is the maximum } j \in \{1,\ldots,n\} \text{ such that } \mathsf{comparable}(q_j,t) \wedge q_j \trianglelefteq t, \text{ and } t = \theta(q_i) \text{ for some substitution } \theta, \text{ where } S = \mathcal{R}an(\theta) \\ \mathsf{abstract}^*(q',S) & \text{if } i \text{ is the maximum } j \in \{1,\ldots,n\} \text{ such that } \mathsf{comparable}(q_j,t) \wedge q_j \trianglelefteq t, \text{ and } q_i \not\trianglelefteq t, \text{ where } q' = q_1,\ldots,q_{i-1},q_{i+1},\ldots,q_n,\ (g,\sigma,\theta) = \mathsf{msg}(q_i,t), \text{ and } S = \{g\} \cup \mathcal{R}an(\sigma) \cup \mathcal{R}an(\theta) \end{cases}$$

Intuitively, given a sequence of terms q and a term t to be added to the sequence, the non-embedding abstraction operator proceeds as follows:

- If t is a variable, it is discarded.
- If t is constructor-rooted, then only its subterms are considered.
- If t is operation-rooted, but there is no comparable term in the sequence, it is added to the sequence.
- If t is operation-rooted and embeds at least one comparable term in the sequence, then there are two options:
  - If t is an instance of some comparable term q_i with $t = \theta(q_i)$, then only the terms in the range of the substitution θ are added.
  - Otherwise, the msg of q_i and t is computed, and the generalization as well as the terms in the substitutions are recursively considered to be added to the sequence from which q_i has been removed.

Note that the removal of q_i from the sequence q in the last case is necessary to ensure termination of the process, as the following example illustrates.

Example 6.32 (Risk of Non-Termination of abstract* [AFV98])**.** *Consider the sequence of terms* $q =$ `f x x` *and the term* $t =$ `f x y` *to be added, and suppose that the sequence is left untouched in the third case above. Since both terms are comparable, and* `f x y` *embeds* `f x x` *but is no instance of it, the auxiliary function* absCall *would proceed as follows.*

- *Firstly, the* msg *of both terms is computed, which yields* $(\texttt{f x y}, \{\texttt{y} \mapsto \texttt{x}\}, id)$.
- *Secondly, a call to* $\mathsf{abstract}^*(q, \{\texttt{f x y}, \texttt{x}\})$ *is performed, which in turn reproduces the original call of* $\mathsf{abstract}^*(q, \texttt{f x y})$*, and thus enters into an infinite loop.*

Although the removal of an element from the sequence is a simple and effective solution, it may also lead to a loss of precision, since a term is replaced by a (possibly more general) abstraction. There also exist other approaches to this problem which

provide slightly better specializations [LMS98], but we follow the original definition for simplicity reasons.

In general, the loss of precision due to the use of the most specific generalization is reasonable and compensated by the simplicity and effectiveness of the method. Although more sophisticated methods such as tree-like states [MG95] instead of sequences can be more powerful, the simplicity of sequences also sometimes helps to avoid code duplication, since variants can be detected earlier. The following example demonstrates how the abstraction method both achieves termination and a good level of specialization.

Example 6.33 (Quality of Partial Evaluation)**.** *Consider again the TRS $\mathcal{R}$ of Example 6.27 reversing a list*

```
reverse      xs = rev xs []
rev []       ys = ys
rev (x : xs) ys = rev xs (x : ys)
```

and the initial term `reverse (1 : zs)`*. Based on the partial evaluation algorithm (Definition 6.20), the non-embedding unfolding rule (Definition 6.29) with the needed narrowing strategy and the non-embedding abstraction operator (Definition 6.31), we obtain the following partial evaluation. The initial sequence of terms is*

$$q_1 = \mathsf{abstract}^*(\varepsilon, \{\texttt{reverse (1 : zs)}\}) = \texttt{reverse (1 : zs)}$$

and the pre-partial evaluation of `reverse (1 : zs)` *is*

```
reverse (1 : [])      → 1 : []
reverse (1 : x : xs)  → rev xs (x : 1 : []) .
```

In the next step, we obtain for $S_1 = \{\texttt{1 : []}, \texttt{rev xs (x : 1 : [])}\}$ the sequence

$$q_2 = \mathsf{abstract}^*(q_1, S_1) = \texttt{reverse (1 : zs), rev xs (x : 1 : [])}$$

and the pre-partial evaluation of `rev xs (x : 1 : [])` *is*

```
rev [] (x : 1 : [])      → x : 1 : []
rev (y:ys) (x : 1 : [])  → rev ys (y : x : 1 : []) .
```

We then obtain for $S_2 = S_1 \cup \{\texttt{x : 1 : []}, \texttt{rev ys (y : x : 1 : [])}\}$ the sequence

$$q_3 = \mathsf{abstract}^*(q_2, S_2) = \texttt{reverse (1 : zs), rev xs (x : y : ys)}$$

where `rev xs (x : 1 : [])` *and* `rev ys (y : x : 1 : [])` *have been generalized to* `rev xs ( x : y : ys)`*. The pre-partial evaluation of* `rev xs ( x : y : ys)` *is*

```
rev [] (x : y : ys)      → x : y : ys
rev (z : zs) (x : y : ys) → rev zs (z : x : y : ys) .
```

Now the sequence is stable, since for $S_3 = S_1 \cup \{\texttt{x : y : ys}, \texttt{rev zs (z : x : y : ys)}\}$

we have

$$q_4 = \mathsf{abstract}^*(q_3, S_3) = \texttt{reverse (1 : zs)}, \texttt{rev xs (x : y : ys)} = q_3\ .$$

Hence, the final pre-partial evaluation $\mathcal{R}'$ is

$$\begin{aligned}
\texttt{reverse (1 : [])} &\to \texttt{1 : []} \\
\texttt{reverse (1 : x : xs)} &\to \texttt{rev xs (x : 1 : [])} \\
\texttt{rev [] (x : y : ys)} &\to \texttt{x : y : ys} \\
\texttt{rev (z : zs) (x : y : ys)} &\to \texttt{rev zs (z : x : y : ys)}
\end{aligned}$$

and both $\mathcal{R}'$ and `reverse (1 : zs)` *are closed with respect to S_{q_3}.*

The following lemma states that abstract* is a correct instance of the generic notion of an abstraction operator and thus computes a finite and closed set of operation-rooted terms, which is necessary for the partial correctness of the partial evaluation scheme. Note that the additional requirement for abstraction operators to compute a sequence of *operation-rooted* terms which was not stated by Alpuente, Falaschi, and Vidal [AFV98] directly follows from the definition of abstract*.

Lemma 6.34 (abstract* is an Abstraction Operator [AFV98])**.** *The function* abstract* *is an abstraction operator according to Definition 6.18.*

Furthermore, due to the use of the embedding ordering both for local and global termination, the entire generic partial evaluation algorithm terminates for the non-embedding unfolding rule and the non-embedding abstraction operator. In combination with Theorem 6.15 stating the strong correctness of partial evaluations based on needed narrowing, the following theorem establishes the total correctness of the partial evaluation scheme based on needed narrowing.

Theorem 6.35 (Termination of Narrowing-Driven Partial Evaluation [AFV98])**.** *The narrowing-driven partial evaluation algorithm terminates for the non-embedding unfolding rule (Definition 6.29) and the non-embedding abstraction operator (Definition 6.31).*

6.4 Summary

The generic framework for narrowing-based partial evaluation presented in this chapter establishes a partial evaluation scheme for functional logic programs represented as confluent term rewriting systems. It is able to automatically improve existing programs while preserving its semantics, and at the same time guarantees termination of the entire process. The presented scheme is furthermore able to produce both polyvariant and polygenetic specializations, and has the same potential for specialization as conjunctive partial deduction or positive supercompilation [AFV98]. In addition, the framework pioneered many subsequent articles about partial evaluation of functional logic programs. Examples are the instantiation for specific narrowing

strategies such as needed narrowing [ALH+05], the integration of residuation into the partial evaluation scheme [AAH+99], investigations of the effectiveness of partial evaluation [AAV00], or the usage of abstract program representations during partial evaluation [AHV00; AHV03]. Finally, [AHV02] presented a practical implementation of a partial evaluator based on the generic framework using the needed narrowing strategy.

However, the presented scheme is based on confluent term rewriting systems and narrowing and thus not appropriate for the partial evaluation of functional logic languages, which feature non-deterministic operations in combination with a call-time choice semantics. In Chapter 7, we therefore extend the partial evaluation scheme to deal with the abstract program representation of FlatCurry, and provide an unfolding rule based on an extension of the operational semantics presented in Chapter 5.

Chapter 7

Partial Evaluation of FlatCurry Programs

In theory, there is no difference between theory and practice. But, in practice, there is.

Manfred Eigen

This chapter presents the development of a partial evaluation scheme for the functional logic language Curry, based on its normalized representation of FlatCurry. The partial evaluator follows the ideas of the existing partial evaluator of Albert, Hanus, and Vidal [AHV02], and can also be regarded as an instance of the generic partial evaluation framework of Alpuente, Falaschi, and Vidal [AFV98]. However, to support additional features such as non-deterministic operations or (mutually) recursive `let` expressions, the basic framework is extended to deal with FlatCurry programs instead of confluent term rewriting systems. Furthermore, the evaluation strategy of needed narrowing is replaced by a variant of the operational semantics to correctly consider the effects of sharing. Note that for all claims stated in this chapter that lack a direct proof, the corresponding proof can be found in Appendix B.

7.1 Introduction

The development of partial evaluators for functional logic languages is rather straightforward, since they can be constructed with techniques similar to the implementation of these languages. Online partial evaluators, in particular, can be seen as a kind of a non-standard interpreter combining evaluation with the creation of residual program fragments. In fact, Albert, Hanus, and Vidal [AHV02] proposed a partial evaluator for Curry based on a intermediate language called "Flat", which only slightly differs from FlatCurry. In particular, Flat makes the evaluation strategy of Curry programs explicit by a compilation of pattern matching into nested `case` expressions, and its usage led to a partial evaluator that was able to optimize practical Curry programs.

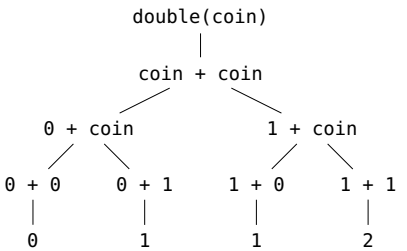
Unfortunately, when this partial evaluator was constructed, the usage of non-deterministic operations, although proposed some years ago [GHL+99], was not well established in Curry. Consequently, the partial evaluation scheme was based on

narrowing and restricted to confluent programs, i. e., all operations were required to be deterministic. Furthermore, at those times the language definition of Curry lacked support for recursive `let` expressions, which thus were also not taken into account. In consequence, this partial evaluator is no longer applicable for contemporary Curry programs, where non-deterministic operations and recursive bindings are regularly used. For these programs, the partial evaluator either produces incorrect specializations, or even fails to produce a specialization at all.

Example 7.1 (Sharing of Non-Determinism)**.** *Consider the following definitions of a non-deterministic operator* `coin` *and a function* `double` *that doubles its (shared) argument:*

```
coin      = 0 ? 1
double(x) = x + x
```

For the expression `double(coin)`*, the original partial evaluator [AHV02] constructs the following narrowing tree (we omit the substitutions for simplicity):*



In consequence, the residual program yields the values `0`*,* `1`*, and* `2` *for the expression* `double(coin)`*. However, according to the call-time choice semantics of Curry, only* `0` *and* `2` *are correct results of* `double(coin)`*, while* `1` *is not.*

This problem arises from the residual semantics [AHV03] of the original partial evaluator. Its unfolding mechanism is based on narrowing, so that during evaluation of a function call, non-deterministic arguments may be substituted more than once in the corresponding right-hand side and thus duplicated, which effectively leads to a run-time choice semantics.

Furthermore, the partial evaluator does not consider (mutually recursive) `let` expressions, i. e., bindings where the variables to be bound may also occur in the right-hand side of the bindings. For example, it is not possible to partially evaluate the program

```
ones = let ones = 1 : ones in ones
main = take 2 ones
```

using the original partial evaluator, since it fails to produce a specialization due to missing support for `let` expressions. One might encounter that this does not impose a real restriction, because recursive `let` bindings could generally be interpreted as

recursive function definitions at the cost of some overhead. While this is indeed possible for the example above, it is not for non-deterministic programs, as the following example illustrates.

Example 7.2 (Recursive `let` Bindings and Non-Determinism)**.** *Consider the following program, computing an infinite sequence of binary digits, where a digit is non-deterministically chosen and then repeated infinitely often.*

```
digits = let digits = (0 ? 1) : digits in digits
```

Because of the `let` *binding, the decision for the first digit (either* `0` *or* `1`*) is shared, and the expression* `take 2 digits` *thus evaluates to either* `[0,0]` *or* `[1,1]`*. If we replaced the definition of* `digits` *by a top-level operation*

```
digits = (0 ? 1) : digits
main   = take 2 digits
```

the expression `main` *would produce the additional results* `[0,1]` *and* `[1,0]`*.*

In consequence, recursive `let` expressions cannot be transformed into mutually recursive operations in general, so that they have to be explicitly considered in the implementation of a partial evaluator.

To conclude, it is crucial for a partial evaluator to cover the full language of Curry in order to deal with realistic programs, including both logic features such as non-determinism as well as functional features such as recursive `let` expressions. Therefore, we follow the preceding work of Albert, Hanus, and Vidal [AHV02] to develop an improved partial evaluator for Curry. Like the original work, we base our implementation on the generic framework for partial evaluation of functional logic languages [AFV98]. However, for the evaluation of expressions we employ a variant of the operational semantics presented in Chapter 5, which supports both non-deterministic operations and the sharing of subexpressions. Furthermore, we adapt the generic framework to consider FlatCurry programs instead of term rewriting systems.

Our approach is comparable to the work of Fischer, Silva, Tamarit, and Vidal [FST+07], who developed a partial evaluator for lazy first-order functional programs which preserves the sharing of expressions. They use a simplified version of the *small-step semantics* of Curry [AHH+05], but do not consider logic features nor higher-order functions due to their focus on first-order functional programs.

7.2 Residualizing Semantics

In the generic partial evaluation scheme, an unfolding rule based on narrowing is responsible for the evaluation of a term to either its value or an operation-rooted term which has to be considered for further evaluation during abstraction. In our context, this evaluation could be achieved by using the operational semantics presented in

Section 5.3, so that an expression e is evaluated by constructing a derivation for the statement $[] : e \Downarrow_1 \Delta : v$. However, in the context of partial evaluation, we have to deal with the following additional requirements.

Termination One important requirement is to ensure termination, i. e., the partial evaluator should produce a correct result after a finite amount of time, irrespectively of the program and expression to be partially evaluated. Therefore, we have to ensure that the evaluation of every single expression terminates, even in presence of non-terminating operations such as

```
loop = loop
```

For this program, evaluation of the expression `loop` based on the operational semantics will fail to terminate, since it would require an infinite derivation to be constructed.

Partial Information In contrast to regular evaluation, we have to deal with partial information about variables, i. e., the value of a variable might be unknown during evaluation if it occurs unbound in the initial expression. For instance, this was the case during the partial evaluation of `(xs ++ ys) ++ zs` in Chapter 6, where all variables are unbound. In terms of partial evaluation, such unbound variables correspond to *dynamic* variables, since their value will only be known during run time, and therefore some residual functions have to be computed.

We will therefore develop a variant of the operational semantics to consider these requirements, which is called the *residualizing semantics* and denoted by $\Downarrow_{PE}$ (where "PE" stands for "partial evaluation"). While the operational semantics evaluates an expression to its value, the residualizing semantics will evaluate an expression to a *residual value*, which may either be a regular value, an unevaluated expression whose evaluation has been deferred due to the risk of non-termination, or a residual expression to be included into the specialized program. We will start with the initial set of inference rules as specified in Figure 5.3, and present its adaptation to partial evaluation in the following. The consideration of extensions of the semantics such as primitive operations will then be discussed in Chapter 8.

Note that it would also be possible to develop the residualizing semantics as an extension of the initial operational semantics $\Downarrow_0$ and consider flat programs and expressions. However, the additional flattening generally increases the size of the respective expressions, which leads to more complex expressions and residual programs. Furthermore, this impedes the comparability with the results obtained by the original partial evaluator [AHV02], so that we employ the more general semantics for partial evaluation.

7.2.1 Deferral of Evaluation

In order to ensure termination of evaluation, we have to be able to postpone some parts of a derivation so that we can construct a finite derivation tree even for an expression whose evaluation fails to terminate in general. For this purpose, we provide the residualizing semantics with the ability to return a *deferred expression* as a residual value, i. e., an expression that is not in head normal form and thus requires further evaluation. We identify a deferred expression by angle parentheses $\langle\!\langle \cdot \rangle\!\rangle$, and define the set of *residual values* as follows:

$$\begin{array}{rcll} \mathit{RValue} & ::= & x & (\text{variable}) \\ & | & c(\overline{x_n}) & (\text{constructor application}, n = \mathsf{arity}(c)) \\ & | & \phi(\overline{x_k}) & (\text{partial application}, k < \mathsf{arity}(\phi)) \\ & | & \langle\!\langle e \rangle\!\rangle & (\text{deferred expression}) \end{array}$$

Since we can identify rule (FunEval) as a potential cause for non-termination, we restrict the applicability of this rule. For this purpose, we extend the inference system with an oracle operation proceed responsible to come to the decision whether to evaluate or defer a function call. We then instrument the rule (FunEval) to only be applicable if the evaluation is safe to proceed, and add a new rule (FunDefer) which implements the deferral of a function call.

$$\text{(FunEval)} \quad \frac{\Gamma : e \Downarrow_{PE} \Delta : v}{\Gamma : f(\overline{x_n}) \Downarrow_{PE} \Delta : v} \quad \text{where } f(\overline{x_n}) = e \text{ is a variable instance of a rule in } P \text{ and } \mathsf{proceed}(\ldots, \Gamma : f(\overline{x_n}))$$

$$\text{(FunDefer)} \quad \Gamma : f(\overline{x_n}) \Downarrow_{PE} \Gamma : \langle\!\langle f(\overline{x_n}) \rangle\!\rangle \quad \text{where } f \in \mathcal{F}^{(n)} \text{ and } \neg\mathsf{proceed}(\ldots, \Gamma : f(\overline{x_n}))$$

The operation proceed is called with the dependency sequence of the current statement up to the current configuration. That is, it may in particular consider the set of function calls that have previously been evaluated to employ a suitable termination criterion. Note that every implementation of proceed is required to allow the evaluation of at least the first function call in a derivation, so that some progress in the evaluation of function calls is guaranteed. We may then defer the evaluation of any following function call to construct a finite derivation. For instance, we can defer repeated evaluations of the function `loop` to ensure termination, like in the following derivation:

$$\frac{\Gamma : \texttt{loop} \Downarrow_{PE} \Gamma : \langle\!\langle \texttt{loop} \rangle\!\rangle}{\Gamma : \texttt{loop} \Downarrow_{PE} \Gamma : \langle\!\langle \texttt{loop} \rangle\!\rangle}$$

Naturally, the rules that are applicable only for certain kinds of values of their premises must also be adjusted to cover deferred expressions. To start with, the rule (VarExp) has to be constrained to be applicable only if the value of its premise is a value in the original sense, i. e., it must not be a deferred expression. This is necessary because rule (VarExp) duplicates the value of its premise by adopting it as the result and as the new binding of the considered variable. Such a duplication is not allowed for deferred expressions, since a deferred expression might be non-deterministic and therefore must not be duplicated.

Example 7.3 (Duplication of Deferred Expression)**.** *Consider the following erroneous application of rule (VarExp):*

$$\frac{\Gamma[\mathtt{x} \mapsto \blacksquare] : \mathtt{coin} \Downarrow_{PE} \Gamma[\mathtt{x} \mapsto \blacksquare] : \langle\!\langle \mathtt{coin} \rangle\!\rangle}{\Gamma[\mathtt{x} \mapsto \mathtt{coin}] : \mathtt{x} \Downarrow_{PE} \Gamma[\mathtt{x} \mapsto \langle\!\langle \mathtt{coin} \rangle\!\rangle] : \langle\!\langle \mathtt{coin} \rangle\!\rangle}$$

This application may destroy the sharing of the value of the non-deterministic expression `coin` *and thus violate the call-time choice semantics.*

Therefore, we constrain the rule (VarExp) to be applicable only for non-deferred values of the premise, and introduce the variant (VarDefer) for deferred expressions.

$$\text{(VarExp)} \quad \frac{\Gamma[x \mapsto \blacksquare] : e \Downarrow_{PE} \Delta : v}{\Gamma[x \mapsto e] : x \Downarrow_{PE} \Delta[x \mapsto v] : v} \quad \begin{array}{l}\text{where } e \notin \{\text{free}, \blacksquare\}, \text{ and } v \in \mathcal{V} \text{ with} \\ \Delta(v) = \text{free}, \text{ or } v = \phi(\overline{x_k}) \text{ with } \phi \in \mathcal{C} \text{ or} \\ k < \text{arity}(\phi)\end{array}$$

$$\text{(VarDefer)} \quad \frac{\Gamma[x \mapsto \blacksquare] : e \Downarrow_{PE} \Delta : \langle\!\langle e' \rangle\!\rangle}{\Gamma[x \mapsto e] : x \Downarrow_{PE} \Delta[x \mapsto \langle\!\langle e' \rangle\!\rangle] : \langle\!\langle x \rangle\!\rangle} \quad \text{where } e \notin \{\text{free}, \blacksquare\}$$

Since the deferred expression cannot be returned as the value, the evaluated variable is returned instead as a deferred value. This allow the identification of variables with unevaluated bindings, so that they can also be handled as unevaluated expressions in subsequent evaluation steps.

Example 7.4 (No Duplication of Deferred Expression)**.** *For the example above, the correct derivation now is*

$$\frac{\Gamma[\mathtt{x} \mapsto \blacksquare] : \mathtt{coin} \Downarrow_{PE} \Gamma[\mathtt{x} \mapsto \blacksquare] : \langle\!\langle \mathtt{coin} \rangle\!\rangle}{\Gamma[\mathtt{x} \mapsto \mathtt{coin}] : \mathtt{x} \Downarrow_{PE} \Gamma[\mathtt{x} \mapsto \langle\!\langle \mathtt{coin} \rangle\!\rangle] : \langle\!\langle \mathtt{x} \rangle\!\rangle}$$

In addition to rule (VarExp), the rule (Value) needs to be adapted, since deferred expressions may also occur as the expression to be evaluated. They are considered as values in the residualizing semantics, so that we extend rule (Value) to cover this case.

$$\text{(Value)} \quad \Gamma : v \Downarrow_{PE} \Gamma : v \quad \begin{array}{l}\text{where } v \in \mathcal{V} \text{ with } \Gamma(v) = \text{free}, \text{ or } v = \phi(\overline{x_k}) \\ \text{with } \phi \in \mathcal{C} \text{ or } k < \text{arity}(\phi), \text{ or } v = \langle\!\langle e \rangle\!\rangle\end{array}$$

Finally, to complete the handling of deferred expressions, we also have to provide a rule for deferred expressions scrutinized by `case` expressions. The general idea is to lift the deferral annotation upwards, such that the entire expression gets annotated as a deferred expression. This is motivated by the fact that annotated expressions are considered as unevaluated subexpressions, and therefore are subject to a later evaluation.

$$\text{(CaseDefer)} \quad \frac{\Gamma : e \Downarrow_{PE} \Delta : \langle\!\langle e' \rangle\!\rangle}{\Gamma : \mathtt{case}\ e\ \mathtt{of}\ \{\ \overline{p_k \to e_k}\ \} \Downarrow_{PE} \Delta : \langle\!\langle \mathtt{case}\ e'\ \mathtt{of}\ \{\ \overline{p_k \to e_k}\ \} \rangle\!\rangle}$$

7.2.2 Partial Information

In contrast to the evaluation in a standard interpreter, partial evaluation has to deal with partial information in the form of unbound variables. For instance, if an

expression like `(xs ++ ys) ++ zs` should be partially evaluated, there is no binding information available for the variables. Hence, they appear unbound in the expression and thus denote dynamic variables in this example, so that some residual code needs to be generated. A possible solution is to handle unbound variables in the same way as logic variables [AFV98], so that they are instantiated during partial evaluation. However, these bindings must then also be respected in the patterns of the generated residual functions, so that the binding information gets propagated from the right-hand side to the left-hand side (*back propagation*).

Example 7.5 (Backpropagation)**.** *In the evaluation of* `(xs ++ ys) ++ zs`*, the variable* `xs` *could be instantiated to the empty list "*`[]`*" or the non-empty list "*`x1 : xs1`*", and two new function rules would have to be generated:*

```
dapp([]      , ys, zs) = ys ++ zs
dapp(x1 : xs1, ys, zs) = x1 : ((xs1 ++ ys) ++ zs)
```

However, these rules do not establish valid FlatCurry functions, so that some post-processing would be necessary.

To avoid this problem, we follow the approach of a *residualizing semantics* [AHV00; AHV03] that does not only compute values, but also *residual program expressions* which are incorporated into the resulting program. This approach is also applied in the original partial evaluator [AHV02] in order to compute residual `case` expressions for both unbound and logic variables. In our setting, however, this is only necessary for unbound variables, since logic variables can be identified during partial evaluation and thus safely instantiated. This separation of unbound and logic variables leads to better specializations, as the following example demonstrates.

Example 7.6 (Unbound versus Logic Variables)**.** *Consider the following expression:*

```
let x free in case x of { True → False; False → True }
```

The variable `x` *can be non-deterministically instantiated to* `True` *or* `False` *using rule (Guess), so that the expression can be non-deterministically evaluated to* `False` *or* `True`*. In contrast, the original partial evaluator residualizes the entire expression.*

To summarize, we want to generate a residual `case` expression if an unbound variable is scrutinized by a `case` expression, where unbound variables can easily be identified by a missing binding in the corresponding heap. We thus extend the set of residual values to also contain residual `case` expressions:

$$
\begin{array}{rcll}
\mathit{RValue} & ::= & x & (\text{variable}) \\
& | & c(\overline{x_n}) & (\text{constructor application}, n = \mathsf{arity}(c)) \\
& | & \phi(\overline{x_k}) & (\text{partial application}, k < \mathsf{arity}(\phi)) \\
& | & \langle\!\langle e \rangle\!\rangle & (\text{deferred expression}) \\
& | & \texttt{case}\ e\ \texttt{of}\ \{\ \overline{p_k \to \langle\!\langle e_k \rangle\!\rangle}\ \} & (\text{residual } \texttt{case} \text{ expression})
\end{array}
$$

Note that this definition allows arbitrary expressions (and not only unbound variables) to be scrutinized in a residual `case` expression, which will later allow the additional consideration of primitive operations. The expressions in the respective branches are furthermore annotated to indicate that they should be considered for a later evaluation. Regarding the inference rules, we extend rule (Value) to be applicable for variables unbound in the heap, such that unbound variables evaluate to themselves. Furthermore, rule (VarDefer) is extended to cover unbound variables as the value of the premise.

$$\text{(Value)} \qquad \Gamma : v \Downarrow_{PE} \Gamma : v \quad \begin{array}{l}\text{where } v \in \mathcal{V} \text{ with } v \notin \mathcal{D}om(\Gamma) \text{ or } \Gamma(v) = \text{free, or}\\ v = \phi(\overline{x_k}) \text{ with } \phi \in \mathcal{C} \text{ or } k < \mathsf{arity}(\phi), \text{ or } v = \langle\!\langle e \rangle\!\rangle\end{array}$$

$$\text{(VarDefer)} \quad \frac{\Gamma[x \mapsto \blacksquare] : e \Downarrow_{PE} \Delta : v}{\Gamma[x \mapsto e] : x \Downarrow_{PE} \Delta[x \mapsto v] : \langle\!\langle x \rangle\!\rangle} \quad \begin{array}{l}\text{where } e \notin \{\text{free}, \blacksquare\}, \text{ and } v \in \mathcal{V} \text{ with}\\ v \notin \mathcal{D}om(\Delta) \text{ or } v = \langle\!\langle e' \rangle\!\rangle\end{array}$$

What remains is the inspection of an unbound variable by a `case` expression, for which a residual expression is generated.

$$\text{(CaseVar)} \quad \frac{\Gamma : e \Downarrow_{PE} \Delta : x}{\Gamma : \mathtt{case}\ e\ \mathtt{of}\ \{\ \overline{p_k \to e_k}\ \} \Downarrow_{PE} \Delta : \mathtt{case}\ x\ \mathtt{of}\ \{\ \overline{p_k \to \langle\!\langle e_k \rangle\!\rangle}\ \}} \quad \begin{array}{l}\text{where } x \notin\\ \mathcal{D}om(\Delta)\end{array}$$

This rule is also applicable if e is an unbound variable, so that the expression to be evaluated may already be a residual `case` expression. We therefore assume an annotation to remove all nested annotations, i. e., we assume $\langle\!\langle e \rangle\!\rangle = \langle\!\langle e' \rangle\!\rangle$ where e' has been obtained from e by removal of all annotations.

It then remains to consider residual `case` expressions as the value of a premise, and we once more extend rule (VarDefer) to cover this additional case. This is motivated by the fact that a residual `case` expression may contain some unevaluated subexpressions inside its branches, which must not be duplicated.

$$\text{(VarDefer)} \quad \frac{\Gamma[x \mapsto \blacksquare] : e \Downarrow_{PE} \Delta : v}{\Gamma[x \mapsto e] : x \Downarrow_{PE} \Delta[x \mapsto v] : \langle\!\langle x \rangle\!\rangle} \quad \begin{array}{l}\text{where } e \notin \{\text{free}, \blacksquare\}, \text{ and } v \in \mathcal{V} \text{ with}\\ v \notin \mathcal{D}om(\Delta) \text{ or } v = \langle\!\langle e' \rangle\!\rangle \text{ or}\\ v = \mathtt{case}\ e'\ \mathtt{of}\ \{\ \overline{p_k \to \langle\!\langle e_k \rangle\!\rangle}\ \}\end{array}$$

Finally, we consider the combination of a residual `case` expression scrutinized by another `case` expression, where we lift the nested `case` expression upwards and defer the expressions in the alternatives:

$$\text{(CaseCase)} \quad \frac{\Gamma : e \Downarrow_{PE} \Delta : \mathtt{case}\ e'\ \mathtt{of}\ \{\ \overline{p'_l \to \langle\!\langle e'_l \rangle\!\rangle}\ \}}{\Gamma : \mathtt{case}\ e\ \mathtt{of}\ \{\ \overline{p_k \to e_k}\ \} \Downarrow_{PE} \Delta : \mathtt{case}\ e'\ \mathtt{of}\ \{\ \overline{p'_l \to \langle\!\langle \mathtt{case}\ e'_l\ \mathtt{of}\ \{\ \overline{p_k \to e_k}\ \} \rangle\!\rangle}\ \}}$$

This approach corresponds to a well-known optimization described in Wadler's article on deforestation [Wad90], which performs the following transformation:

$$\mathtt{case}\ (\mathtt{case}\ e\ \mathtt{of}\ \{\ \overline{p'_l \to e'_l}\ \})\ \mathtt{of}\ \{\ \overline{p_k \to e_k}\ \} \to \mathtt{case}\ e\ \mathtt{of}\ \{\ \overline{p'_l \to \mathtt{case}\ e'_l\ \mathtt{of}\ \{\ \overline{p_k \to e_k}\ \}}\ \}$$

Although this transformation increases the code size due to the repetition of the

expressions $\overline{e_k}$, it does not violate the call-time choice semantics since the different branches are evaluated independently. Moreover, it may enable further progress in partial evaluation that would not be possible otherwise.

Example 7.7 (Lifting of Nested `case` Expression)**.** *Consider the expression*

```
case (case x of {True → False; False → True}) of {True → False; False → True}
```

where `x` *is unbound. By rule (CaseCase), this expression is transformed into*

```
case x of { True  → ⟪case False of {True → False; False → True}⟫
          ; False → ⟪case True  of {True → False; False → True}⟫
          }
```

so that the nested `case` *expressions can later be evaluated to* `True` *and* `False`*, respectively.*

The consideration of unbound variables and the generation of residual `case` expressions as defined above completes the description of the residualizing semantics, and we summarize its rules in Figure 7.1 for ease of reference. Note that the residualizing semantics is a conservative extension of the operational semantics $\Downarrow_1$, provided that the operation proceed yields true for all arguments and that the in-configuration of the initial statement does not contain unbound variables.

Theorem 7.8 ($\Downarrow_{PE}$ is a Conservative Extension of $\Downarrow_1$)**.** *Let* $\Gamma : e$ *be a configuration without unbound variables or annotations, and* $\mathsf{proceed}(\cdot) = \mathsf{true}$*. Then* $\Gamma : e \Downarrow_1 \Delta : v$ *if and only if* $\Gamma : e \Downarrow_{PE} \Delta : v$*.*

Proof. Since all rules in $\Downarrow_1$ are also contained in $\Downarrow_{PE}$ and the additional rules of $\Downarrow_{PE}$ cannot be introduced under the assumption about proceed, every derivation for $\Gamma : e \Downarrow_1 \Delta : v$ is a valid derivation for $\Gamma : e \Downarrow_{PE} \Delta : v$ and vice versa. □

7.2.3 Correctness

To ensure the correctness of the partial evaluation process, the residualizing semantics must evaluate an expression to the same set of values as the operational semantics. However, the residualizing may also defer parts of a derivation and generate residual `case` expressions in case of unbound variables, so that the residual value might need further evaluation. Moreover, unbound variables may only occur at partial evaluation time but not at run time, since a program is generally considered to be well-formed. We therefore distinguish between bindings available at run time, which correspond to *dynamic* bindings, and *static* bindings which are available at partial evaluation time. For this purpose, we separate a heap Γ into a part Γ_S containing the *static* bindings and a part Γ_D containing the *dynamic* bindings, such that $\Gamma = \Gamma_S \uplus \Gamma_D$. At the beginning of partial evaluation, we start with $\Gamma_S = []$, and the heap will then be subsequently extended during partial evaluation. Based on this consideration, we can state the (partial) correctness of the residualizing semantics.

(Value) $\Gamma : v \Downarrow_{PE} \Gamma : v$ where $v \in \mathcal{V}$ with $v \notin \mathcal{D}om(\Gamma)$ or $\Gamma(v) = \text{free}$, or $v = \phi(\overline{x_k})$ with $\phi \in \mathcal{C}$ or $k < \text{arity}(\phi)$, or $v = \langle\!\langle e \rangle\!\rangle$

(VarExp) $$\frac{\Gamma[x \mapsto \blacksquare] : e \Downarrow_{PE} \Delta : v}{\Gamma[x \mapsto e] : x \Downarrow_{PE} \Delta[x \mapsto v] : v}$$ where $e \notin \{\text{free}, \blacksquare\}$, and $v \in \mathcal{V}$ with $\Delta(v) = \text{free}$, or $v = \phi(\overline{x_k})$ with $\phi \in \mathcal{C}$ or $k < \text{arity}(\phi)$

(VarDefer) $$\frac{\Gamma[x \mapsto \blacksquare] : e \Downarrow_{PE} \Delta : v}{\Gamma[x \mapsto e] : x \Downarrow_{PE} \Delta[x \mapsto v] : \langle\!\langle x \rangle\!\rangle}$$ where $e \notin \{\text{free}, \blacksquare\}$, and $v \in \mathcal{V}$ with $v \notin \mathcal{D}om(\Delta)$ or $v = \langle\!\langle e' \rangle\!\rangle$ or $v = \texttt{case}\ e'\ \texttt{of}\ \{\ \overline{p_k \to \langle\!\langle e_k \rangle\!\rangle}\ \}$

(Flatten) $$\frac{\Gamma[\overline{y_l \mapsto e'_l}] : \phi(\overline{x_k}) \Downarrow_{PE} \Delta : v}{\Gamma : \phi(\overline{e_k}) \Downarrow_{PE} \Delta : v}$$ where $\exists i \in \{1, \ldots, k\}$ such that $e_i \notin \mathcal{V}$, and $(\overline{y_l}, \overline{e'_l}, \overline{x_k}) = \text{splitArgs}(\overline{e_k})$

(FunEval) $$\frac{\Gamma : e \Downarrow_{PE} \Delta : v}{\Gamma : f(\overline{x_n}) \Downarrow_{PE} \Delta : v}$$ where $f(\overline{x_n}) = e$ is a variable instance of a rule in P and $\text{proceed}(\ldots, \Gamma : f(\overline{x_n}))$

(FunDefer) $\Gamma : f(\overline{x_n}) \Downarrow_{PE} \Gamma : \langle\!\langle f(\overline{x_n}) \rangle\!\rangle$ where $f \in \mathcal{F}^{(n)}$ and $\neg\text{proceed}(\ldots, \Gamma : f(\overline{x_n}))$

(Let) $$\frac{\Gamma[\overline{x_k \mapsto e_k}] : e \Downarrow_{PE} \Delta : v}{\Gamma : \texttt{let}\ \{\overline{x_k = e_k}\}\ \texttt{in}\ e \Downarrow_{PE} \Delta : v}$$

(Or) $$\frac{\Gamma : e_i \Downarrow_{PE} \Delta : v}{\Gamma : e_1\ \texttt{?}\ e_2 \Downarrow_{PE} \Delta : v}$$ where $i \in \{1, 2\}$

(Free) $$\frac{\Gamma[\overline{x_k \mapsto \text{free}}] : e \Downarrow_{PE} \Delta : v}{\Gamma : \texttt{let}\ \overline{x_k}\ \texttt{free in}\ e \Downarrow_{PE} \Delta : v}$$

(Select) $$\frac{\Gamma : e \Downarrow_{PE} \Delta : c(\overline{x_n}) \quad \Delta : \sigma(e_i) \Downarrow_{PE} \Theta : v}{\Gamma : \texttt{case}\ e\ \texttt{of}\ \{\ \overline{p_k \to e_k}\ \} \Downarrow_{PE} \Theta : v}$$ where $c(\overline{x_n}) = \sigma(p_i)$ and $i \in \{1, \ldots, k\}$

(Guess) $$\frac{\Gamma : e \Downarrow_{PE} \Delta : x \quad \Delta[x \mapsto c(\overline{x_n}), \overline{x_n \mapsto \text{free}}] : e_i \Downarrow_{PE} \Theta : v}{\Gamma : \texttt{case}\ e\ \texttt{of}\ \{\ \overline{p_k \to e_k}\ \} \Downarrow_{PE} \Theta : v}$$ where $\Delta(x) = \text{free}$, $c(\overline{x_n}) = p_i$, and $i \in \{1, \ldots, k\}$

(CaseVar) $$\frac{\Gamma : e \Downarrow_{PE} \Delta : x}{\Gamma : \texttt{case}\ e\ \texttt{of}\ \{\ \overline{p_k \to e_k}\ \} \Downarrow_{PE} \Delta : \texttt{case}\ x\ \texttt{of}\ \{\ \overline{p_k \to \langle\!\langle e_k \rangle\!\rangle}\ \}}$$ where $x \notin \mathcal{D}om(\Delta)$

(CaseDefer) $$\frac{\Gamma : e \Downarrow_{PE} \Delta : \langle\!\langle e' \rangle\!\rangle}{\Gamma : \texttt{case}\ e\ \texttt{of}\ \{\ \overline{p_k \to e_k}\ \} \Downarrow_{PE} \Delta : \langle\!\langle \texttt{case}\ e'\ \texttt{of}\ \{\ \overline{p_k \to e_k}\ \} \rangle\!\rangle}$$

(CaseCase) $$\frac{\Gamma : e \Downarrow_{PE} \Delta : \texttt{case}\ e'\ \texttt{of}\ \{\ \overline{p'_l \to \langle\!\langle e'_l \rangle\!\rangle}\ \}}{\Gamma : \texttt{case}\ e\ \texttt{of}\ \{\ \overline{p_k \to e_k}\ \} \Downarrow_{PE} \Delta : \texttt{case}\ e'\ \texttt{of}\ \{\ \overline{p'_l \to \langle\!\langle \texttt{case}\ e'_l\ \texttt{of}\ \{\ \overline{p_k \to e_k}\ \} \rangle\!\rangle}\ \}}$$

Figure 7.1. Residualizing Semantics

Theorem 7.9 (Correctness of Residualizing Semantics)**.** $(\Gamma_S \uplus \Gamma_D) : e \Downarrow_1 \Delta : v$ *if and only if* $\Gamma_S : e \Downarrow_{PE} \Theta : r$ *and* $(\Theta \uplus \Gamma_D) : r \Downarrow_1 \Delta : v$.

Note that $\Theta \uplus \Gamma_D$ denotes a valid heap since the variables in $\mathcal{D}om(\Theta)$ are either contained in $\mathcal{D}om(\Gamma_S)$, locally bound in e, or introduced as fresh variables, and thus distinct from the variables in $\mathcal{D}om(\Gamma_D)$. Furthermore, the annotations introduced in the residualizing semantics are neglected in the operational semantics, such that the statements $\Gamma : \langle\!\langle e \rangle\!\rangle \Downarrow_1 \Delta : v$ and $\Gamma : e \Downarrow_1 \Delta : v$ are considered as equivalent.

What remains is to show that a deferral of function calls is sufficient to ensure finiteness of the constructed derivations. This is a consequence of the following lemma, which states that for every application of a rule except rule (FunEval), the complexities of the in-configurations strictly decrease.

Lemma 7.10 (Strictly Decreasing Complexity of In-Configurations in $\Downarrow_{PE}$ without (FunEval))**.** *The following conditions hold for every derivation D with respect to* $\Downarrow_{PE}$ *that does not employ rule (FunEval):*

1. *If D is a derivation of the form*

$$\frac{C_1 \Downarrow_{PE} C_1' \quad \ldots \quad C_n \Downarrow_{PE} C_n'}{C \Downarrow_{PE} C'}$$

   where $n > 0$*, it holds that* $\mathcal{M}_{C_i} <_{\mathsf{mul}} \mathcal{M}_C$ *for all* $i \in \{1, \ldots, n\}$*, i. e., the complexities of the in-configurations of the premises are strictly smaller.*

2. *If D is a derivation for* $\Gamma : e \Downarrow_{PE} \Delta : v$*, it holds that* $\mathcal{M}_{\Delta:v} \leqslant_{\mathsf{mul}} \mathcal{M}_{\Gamma:e}$*, i. e., the complexity does not increase, or v is a deferred or residual* `case` *expression.*

Since every right-hand side in the program is of a finite depth, every evaluation of rule (FunEval) can only lead to a finite increase in the complexity of the in-configuration, so that finitely many applications of rule (FunEval) imply a finite derivation.

7.3 Partial Evaluation

In this section, we adapt the general partial evaluation scheme of Alpuente, Falaschi, and Vidal [AFV98] to the context of FlatCurry programs. For this purpose, we have to adjust some concepts and definitions that rely on narrowing as the evaluation mechanism, as well as consider implications from the more complex representation of programs.

7.3.1 Resultants

The general purpose of partial evaluation is to anticipate some computation steps at compile time that otherwise would have been performed at run time. Since there also exist expressions that cannot be further evaluated, such as unbound variables or

constructor applications, it is reasonable to exclude them from the partial evaluation process. For this purpose, we establish the notion of a *partially evaluable* expression.

Definition 7.11 (Partially Evaluable Expression)**.** *An expression e is partially evaluable, denoted by* $\mathsf{peval}(e)$*, if and only if the following conditions hold:*

1. *e is either a fully saturated function call, a* `let` *expression, or a* `case` *expression, and*
2. *all derivations of a statement $[] : e \Downarrow_{PE} \Delta : v$ involve at least one evaluation step, i. e., at least one application of one of the rules (FunEval), (Select), (Guess), or (Or).*

This definition excludes the partial evaluation of unbound variables, constructor applications and partial function applications, since their partial evaluation would not perform any evaluation step. Furthermore, non-deterministic expressions such as e_1 `?` e_2 are also excluded, since their alternatives can be considered individually.

Evaluation of a partially evaluable expression using the residualizing semantics yields an out-configuration as the result, where the residual value may reference variables bound in the heap. Since the general partial evaluation framework as presented in Chapter 6 operates on expressions but not on configurations, we either need to convert a configuration back into an expression, or to extend the framework to consider configurations instead. In principle, both solutions are possible, but the former one is considerably simpler, since it only requires a local adaptation. We therefore convert a configuration into an expression by an addition of the bindings to the expression, which is achieved by means of the following operation.

Definition 7.12 (Dereferencing)**.** *Let $\Gamma : e$ be a configuration such that $\blacksquare \notin \mathcal{R}an(\Gamma)$. Then the dereferencing of Γ with respect to e is defined as follows:*

$$\mathsf{drf}(\Gamma, e) = \begin{cases} \langle\!\langle \mathsf{drf}(\Gamma, e') \rangle\!\rangle & \text{if } e = \langle\!\langle e' \rangle\!\rangle \\ \texttt{case}\ x\ \texttt{of} & \text{if } e = \texttt{case}\ x\ \texttt{of}\ \{\ \overline{p_k \to \langle\!\langle e_k \rangle\!\rangle}\ \},\ x \notin \mathcal{D}om(\Gamma) \\ \quad \{\ \overline{p_k \to \mathsf{drf}(\Gamma, \langle\!\langle e_k \rangle\!\rangle)}\ \} & \\ \mathsf{add}(\Gamma, e) & \text{otherwise} \end{cases}$$

where the addition of a heap Γ to an expression e is defined as

$$\mathsf{add}(\Gamma, e) = \texttt{let}\ \{x \mid (x, \mathrm{free}) \in \Gamma\}\ \texttt{free in let}\ \{\{x = e' \mid (x, e') \in \Gamma \wedge e' \in Exp\}\}\ \texttt{in}\ e$$

In principle, the bindings of the heap are divided into logic variables and bindings to expressions, as both kinds are introduced by different syntactic constructs. Since logic variables may be referenced in `let`-bindings but not vice versa, logic variables are introduced first to obtain correct variable bindings. If either the set of free or bound variables is empty, the corresponding `let` expressions are omitted. In addition, for an annotated expression the annotation is lifted upwards, and for a residual `case` expression the heap is added to the respective alternatives. For the latter case, we assume an appropriate renaming of the bound variables, such that the result still obeys the variable convention.

Note that the exclusion of blackholes in the heap imposes no real restriction, since every expression is partially evaluated with an initially empty heap. Thus, no blackholes can occur in the resulting configuration according to Lemma 5.15, which can easily be transferred to the residualizing semantics.

Example 7.13 (Dereferencing)**.** *For the configuration* $[\mathtt{x} \mapsto \mathsf{free}, \mathtt{y} \mapsto \mathtt{False}] : \langle\!\langle\, \mathtt{f(x,y)} \,\rangle\!\rangle$*, the process of dereferencing produces the expression*

$$\langle\!\langle\, \mathtt{let\ x\ free\ in\ let\ \{y\ =\ False\}\ in\ f(x,y)} \,\rangle\!\rangle$$

and for $[\mathtt{y} \mapsto \mathtt{False}] : \mathtt{case\ x\ of\ \{\ True} \to \langle\!\langle\, \mathtt{x\ ||\ y} \,\rangle\!\rangle\ \mathtt{\}}$ *it produces the result*

$$\mathtt{case\ x\ of\ \{\ True} \to \langle\!\langle\, \mathtt{let\ \{y\ =\ False\}\ in\ x\ ||\ y} \,\rangle\!\rangle\ \mathtt{\}}$$

The presented dereferencing mechanism can still be improved in different ways, for instance by the omission of unused bindings or by the addition of bindings for unbound variables scrutinized in a residual `case` expression [FST+07], such that in the second example above the binding $[\mathtt{x} \mapsto \mathtt{True}]$ would be added to the alternative's expression. However, we postpone such optimizations until Chapter 8 to concentrate on the fundamentals of the partial evaluation scheme.

Based on the dereferencing operation, we can then define the notion of a resultant obtained using the residualizing semantics.

Definition 7.14 (Resultant)**.** *Let P be a program and e a partially evaluable expression. Given a derivation $[] : e \Downarrow_{PE} \Theta : r$, its associated resultant is the pair $e \to \mathsf{drf}(\Theta, r)$. Furthermore, we say that $e \to \mathsf{drf}(\Theta, r)$ is a resultant of e.*

The restriction of resultants to partially evaluable expressions corresponds to the definition of resultants for narrowing-based partial evaluation, where the corresponding narrowing derivation must involve at least one narrowing step. In consequence, this ensures that for every resultant the corresponding partial evaluation achieves some progress.

7.3.2 Pre-Partial Evaluations

In general, the partial evaluation of an expression may lead to multiple derivations, since the residualizing semantics is non-deterministic in the rules (Or) and (Guess). For instance, the expression `let x free in not(x)` may evaluate to the values `True` and `False`. Furthermore, some expressions may have no successful derivation at all, such as the expression `head []`, for which evaluation fails. In the partial evaluation process, we have to respect *all possible derivations* to ensure completeness of the transformation. Since the representation of FlatCurry excludes the definition of a function by multiple rules, we combine the different resultants of an expression to a non-deterministic right-hand side, which is achieved by the following operation.

Definition 7.15 (Disjunctive Combination of Expressions)**.** *Let $\overline{e_n}$ be a finite sequence of expressions. Their disjunctive combination, denoted by* $\mathsf{discomb}(\overline{e_n})$*, is defined as*

$$\mathsf{discomb}(\overline{e_n}) = \begin{cases} \texttt{failed} & \text{if } n = 0 \\ e_1 \texttt{ ? ... ? } e_n & \text{otherwise} \end{cases}$$

where we assume the non-deterministic choice operator to be right-associative.

For instance, the values `True` and `False` will be combined to `True ? False`, and for the expression `case [] of { (x : xs) -> x }` the missing value will be represented by `failed`. We can then use the operation discomb to construct a pre-partial evaluation for an expression e from the sequence of its resultants.

Definition 7.16 (Pre-Partial Evaluation)**.** *Let P be a program, e a partially evaluable expression, and $\overline{e \to e'_n}$ the sequence of all resultants of e constructed by pairwise different derivations. Then the pair $e \to \mathsf{discomb}(\overline{e'_n})$ is called a pre-partial evaluation of e in P. The pre-partial evaluation of a set of partially evaluable expressions E in P is defined as the union of the pre-partial evaluations for the expressions of E in P.*

Just like resultants, pre-partial evaluations are only defined for partially evaluable expressions. This is necessary because expressions which are not partially evaluable have no resultants and would then contain a failing right-hand side in their pre-partial evaluation. This, however, would cause expressions like `True` to be partially evaluated to `failed`, which is clearly unintended.

7.3.3 Partial Evaluations

To guarantee that each function call in the specialized program is covered by some program rule, the pre-partial evaluations have to be checked for their closedness, and subexpressions that are instances of partially evaluated expressions should then be renamed. However, we have to avoid the unintended introduction of sharing, as the following example demonstrates.

Example 7.17 (Partial Evaluation of Non-Linear Expressions)**.** *Consider the narrowing-based partial evaluation of the set $S = \{\texttt{pair x x}\}$ with respect to the TRS*

```
pair x y = (x, y)
```

Then we can obtain the pre-partial evaluation `pair x x → (x, x)`*, the renaming $\rho = \{\texttt{pair x x} \mapsto \texttt{idPair x}\}$, and the partial evaluation*

```
idPair x  = (x, x)
```

If we now consider the term $t =$ `pair (0 ? 1) (0 ? 1)` *where "`?`" is a primitive operation, t is renamed to* $\mathsf{ren}_\rho(t) =$ `idPair (0 ? 1)`*. This is correct in the context of narrowing, since function arguments are substituted in a narrowing step, so that*

$$\texttt{idPair (0 ? 1)} \rightsquigarrow \texttt{(0 ? 1, 0 ? 1)}$$

is a valid narrowing step and we obtain the results `(0, 0)`, `(0, 1)`, `(1, 0)`, *and* `(1, 1)`. *In the operational semantics, however, the value of function arguments is shared, so that* `idPair (0 ? 1)` *would only produce the results* `(0, 0)` *and* `(1, 1)`.

In consequence, the introduction of sharing may violate the call-time choice semantics and thus has to be avoided. We therefore require the expressions to be partially evaluated to be linear, which can easily be achieved by a *linearization* replacing repeated occurrences of unbound variables by fresh ones. Note that this also implies a loss of information, and we will propose an improvement in Chapter 8.

Based on this consideration, we can define the closedness of FlatCurry expressions with respect to a set of expressions, which is used to ensure that all expressions in the right-hand side of pre-partial evaluations will later be covered by some program rule. We require the set of expressions to be linear as well as partially evaluable, where the latter requirement is necessary to avoid the later construction of renamings for expressions that have no pre-partial evaluation. Furthermore, this restriction reduces the non-determinism of the closedness property, since expressions that are not partially evaluable can now only be closed by recursive closedness of their subexpressions.

Definition 7.18 (Closedness)**.** *Let S be a finite set of linear and partially evaluable expressions. Then an expression e is S-closed if* $\mathsf{closed}(S, e)$ *holds, where* closed *is inductively defined as follows:*

$$\mathsf{closed}(S,e) :\Leftrightarrow \begin{cases} \mathsf{true} & \text{if } e \in \mathcal{V} \\ \bigwedge_{i=1}^{k} \mathsf{closed}(S, e_i) & \text{if } e = c(\overline{e_k}) \text{ with } c \in \mathcal{C} \cup \mathcal{P} \\ \mathsf{closed}(S, f(\overline{x_n})) \wedge \bigwedge_{i=1}^{k} \mathsf{closed}(S, e_i) & \text{if } e = f(\overline{e_k}) \text{ with } f \in \mathcal{F}^{(n)} \\ & \text{and } k < n \\ \bigwedge_{i=0}^{k} \mathsf{closed}(S, e_i) & \text{if } e = \texttt{let } \{\overline{x_k = e_k}\} \texttt{ in } e_0 \\ \mathsf{closed}(S, e') & \text{if } e = \texttt{let } \overline{x_k} \texttt{ free in } e' \\ \mathsf{closed}(S, e_1) \wedge \mathsf{closed}(S, e_2) & \text{if } e = e_1 \texttt{ ? } e_2 \\ \bigwedge_{i=0}^{k} \mathsf{closed}(S, e_i) & \text{if } e = \texttt{case } e_0 \texttt{ of } \{ \overline{p_k \to e_k} \} \\ \bigwedge_{e' \in \mathcal{R}an(\sigma)} \mathsf{closed}(S, e') & \text{if } \exists \sigma, \exists s \in S \text{ such that } e = \sigma(s) \end{cases}$$

A set (sequence) of expressions E is S-closed, written as $\mathsf{closed}(S, E)$*, if* $\mathsf{closed}(S, e)$ *holds for every $e \in E$. A configuration $\Gamma : e$ is said to be S-closed if e and $\mathcal{R}an(\Gamma)$ are S-closed, and a program P is S-closed if all right-hand sides in P are S-closed.*

An important property of this definition is the transitivity of closedness, which is expressed in the following lemma and has also been shown in the context of term rewriting systems [AFV98].

Lemma 7.19 (Transitivity of Closedness)**.** *If an expression e is S_1-closed and S_1 is S_2-closed, then e is S_2-closed.*

Note that this definition of closedness considers partial function applications to be closed only if the function call applied to variables and the respective arguments are closed, which ensures the definedness of partial function calls with varying arguments. Function calls need to be an instance of some expression in the set S with a closed substitution, while all other expressions can either be closed if they are an instance of some $s \in S$ and the substitution is recursively closed, or by closedness of their subexpressions.

Example 7.20 (Closedness)**.** *Consider the expression*

$$e = \texttt{case not(not(True)) of True} \to \texttt{False}$$

and the set

$$S = \{\texttt{case not(x) of True} \to \texttt{False}, \texttt{not(not(True))}\}\,.$$

Note that e is an instance of `case not(x) of True` $\to$ `False` *with* $\sigma = \{\texttt{x} \mapsto \texttt{not(True)}\}$*, but $\mathcal{R}an(\sigma)$ is not S-closed. However, the expression e is S-closed since its subexpressions* `not(not(True))` *and* `False` *are S-closed.*

After the computation of the pre-partial evaluations, their left-hand sides are renamed to obtain valid left-hand sides of FlatCurry function definitions. The notion of an independent renaming (Definition 6.8) can be transferred to our context without modifications, so that it remains to extend the recursive renaming function to be applicable to FlatCurry expressions.

Definition 7.21 (Renaming Function)**.** *Let E be a finite set of linear and partially evaluable expressions and ρ an independent renaming for E. Then the renaming of e under ρ, denoted by $\mathsf{ren}_\rho(e)$, is defined as*

$$\mathsf{ren}_\rho(e) = \begin{cases} e & \text{if } e \in \mathcal{V} \\ c(\overline{\mathsf{ren}_\rho(e_k)}) & \text{if } e = c(\overline{e_k}) \text{ with } c \in \mathcal{C} \\ f'(\overline{\mathsf{ren}_\rho(e_k)}) & \text{if } e = f(\overline{e_k}) \text{ with } f \in \mathcal{F}^{(n)},\ k < n, \text{ and } f(\overline{x_n}) \in E, \\ & \text{where } f'(\overline{x_n}) = \rho(f(\overline{x_n})) \\ \sigma'(\rho(e')) & \text{if } \exists \sigma, \exists e' \in E \text{ such that } e = \sigma(e') \text{ and } \mathsf{closed}(E, \mathcal{R}an(\sigma)), \\ & \text{where } \sigma' = \{x \mapsto \mathsf{ren}_\rho(\sigma(x)) \mid x \in \mathcal{D}om(\sigma)\} \\ \mathsf{ren}'_\rho(e) & \text{otherwise} \end{cases}$$

The recursive renaming ren'_ρ is defined as

$$\begin{aligned} \mathsf{ren}'_\rho(\phi(\overline{e_k})) &= \phi(\overline{\mathsf{ren}_\rho(e_k)}) \\ \mathsf{ren}'_\rho(\texttt{let } \{\overline{x_k = e_k}\} \texttt{ in } e) &= \texttt{let } \{\overline{x_k = \mathsf{ren}_\rho(e_k)}\} \texttt{ in } \mathsf{ren}_\rho(e) \\ \mathsf{ren}'_\rho(\texttt{let } \overline{x_k} \texttt{ free in } e) &= \texttt{let } \overline{x_k} \texttt{ free in } \mathsf{ren}_\rho(e) \\ \mathsf{ren}'_\rho(e_1 \texttt{ ? } e_2) &= \mathsf{ren}_\rho(e_1) \texttt{ ? } \mathsf{ren}_\rho(e_2) \\ \mathsf{ren}'_\rho(\texttt{case } e \texttt{ of } \{\ \overline{p_k \to e_k}\ \}) &= \texttt{case } \mathsf{ren}_\rho(e) \texttt{ of } \{\ \overline{p_k \to \mathsf{ren}_\rho(e_k)}\ \} \end{aligned}$$

The renaming can furthermore be extended to configurations as follows:

$$\begin{aligned} \mathsf{ren}_\rho(\Gamma : e) &= \mathsf{ren}_\rho(\Gamma) : \mathsf{ren}_\rho(e) \\ \mathsf{ren}_\rho(\Gamma) &= \{(x, \mathsf{ren}_\rho(b)) \mid (x, b) \in \Gamma\} \\ \mathsf{ren}_\rho(\mathrm{free}) &= \mathrm{free} \\ \mathsf{ren}_\rho(\blacksquare) &= \blacksquare \end{aligned}$$

The different cases are considered to be tried in their textual order, such that this definition is only non-deterministic if the expression e to be renamed is an instance of more than one expression $e' \in E$. In this case, we assume an appropriate heuristic such that the renaming operation becomes deterministic. Furthermore, an expression that is an instance of an expression in E is only renamed if the range of the corresponding substitution is also closed, otherwise its subexpressions are recursively renamed. This restriction ensures that for an expression e that is closed with respect to the set E, its renaming $\mathsf{ren}_\rho(e)$ is closed with respect to the set $\rho(E)$, which is necessary to ensure that every renamed expression is defined in the renamed program.

Example 7.22 (Closedness After Renaming)**.** *Consider again the expression*

$$e = \texttt{case not(not(True)) of True} \to \texttt{False}\,,$$

the set of expressions

$$E = \{\texttt{case not(x) of True} \to \texttt{False}, \texttt{not(not(True))}\}\,,$$

the following independent renaming ρ

$$\begin{aligned} \texttt{case not(x) of True} \to \texttt{False} &\mapsto \texttt{f(x)} \\ \texttt{not(not(True))} &\mapsto \texttt{g} \end{aligned}$$

and the pre-partial evaluation

$$\begin{aligned} (\texttt{case not(x) of True} \to \texttt{False}) &\to (\texttt{case x of False} \to \texttt{False}) \\ \texttt{not(not(True))} &\to \texttt{True} \end{aligned}$$

Note that $\mathsf{closed}(E, e)$ *holds according to Example 7.20. If we would omit the restriction on closedness of the substitution, then* $\mathsf{ren}_\rho(e) = \texttt{f(not(True))}$ *would be a valid renaming. However, the renaming is not* $\rho(E)$*-closed since there is no pre-partial evaluation for* `not(True)`*. With the restriction, the renaming instead yields* $\mathsf{ren}_\rho(e) = \texttt{case g of True} \to \texttt{False}$*, which is* $\rho(E)$*-closed and, thus, well-defined.*

The following lemma states the strong correspondence between the closedness of an expression and the closedness of its renaming, as demonstrated in Example 7.22.

Lemma 7.23 (Correspondence of Closedness under Renaming)**.** *Let E be a finite set of linear and partially evaluable expressions, ρ an independent renaming for E, and $E' = \rho(E)$. Then an expression e is E-closed if and only if $e' = \mathsf{ren}_\rho(e)$ is E'-closed.*

Based on the refined definitions of pre-partial evaluations and the recursive renaming, we can proceed to define the partial evaluation of a set of linear and partially evaluable FlatCurry expressions.

Definition 7.24 (Partial Evaluation)**.** *Let P be a program, E a finite set of linear and partially evaluable expressions, P' a pre-partial evaluation of E in P, and ρ an independent renaming for E. Then the partial evaluation P'' of E in P under ρ is defined as follows:*

$$P'' = \bigcup_{e \in E} \{\rho(e) = \mathsf{ren}_\rho(e') \mid (e \to e') \in P' \text{ is the pre-partial evaluation of } e\}$$

Thus, the definition of partial evaluation for FlatCurry programs is analogous to the definition of partial evaluation based on narrowing for term rewriting systems, so that the obtained specializations are comparable (modulo program representation, sharing and linearization).

Example 7.25 (Partial Evaluation)**.** *For the set of expressions $E = \{$`xs ++ ys`$\}$, the independent renaming $\rho =$* `xs ++ ys` $\mapsto$ `app(xs, ys)`*, and the pre-partial evaluation*

```
xs ++ ys → case xs of { [] -> ys; (z:zs) -> z : (zs ++ ys) }
```

we obtain the partial evaluation

```
app(xs, ys) = case xs of { [] -> ys; (z:zs) -> z : app(zs, ys) }
```

and thus a result comparable to the result obtained by narrowing-based partial evaluation.

7.3.4 Extensions of Configurations

In the process of renaming, expressions are renamed to function calls, where some subexpressions are converted to arguments of the function. For instance, if we consider an expression $e = \sigma(e')$ which is an instance of a linear expression e' with $\rho(e') = f(\overline{x_n})$ and $\sigma = \{\overline{x_n \mapsto e_n}\}$, then e is renamed to $f(\overline{\mathsf{ren}_\rho(e_n)})$. If we furthermore assume that the expressions $\overline{e_n}$ cannot be further renamed and that e' has been partially evaluated to itself, we obtain $\mathsf{ren}_\rho(e) = f(\overline{e_n})$ where f is defined as $f(\overline{x_n}) = e'$. In the context of narrowing, this renaming does not affect the semantics, since the call $f(\overline{e_n})$ will be rewritten to $\sigma(e') = e$ in a single narrowing step. For the operational semantics, however, this is not true since the non-variable arguments of a function call are inserted into the heap due to flattening. If σ contains only non-variable substitutions, then the evaluation of $f(\overline{e_n})$ with respect to the heap Γ leads to the derivation

$$\dfrac{\dfrac{\Gamma[\overline{x_n \mapsto e_n}] : e' \Downarrow_1 \Delta : v}{\Gamma[\overline{x_n \mapsto e_n}] : f(\overline{x_n}) \Downarrow_1 \Delta : v}}{\Gamma : f(\overline{e_n}) \Downarrow_1 \Delta : v}$$

In contrast, evaluation of the original expression $e = \sigma(e')$ leads to a derivation for the statement $\Gamma : \sigma(e') \Downarrow_1 \Delta' : v'$. Because the configurations $\Gamma[\overline{x_n \mapsto e_n}] : e'$ and $\Gamma : \sigma(e')$ differ both in the heap and the expression, we conclude that the out-configurations

$\Delta : v$ and $\Delta' : v'$ may also be different. However, the only difference between $\Gamma[\overline{x_n \mapsto e_n}] : e'$ and $\Gamma : \sigma(e')$ is that some subexpressions of $\sigma(e')$ have been extracted into the extended heap $\Gamma[\overline{x_n \mapsto e_n}]$, and we expect both configurations to yield the same value in general, so that the configurations can be thought of equivalent.

To be able to formally express this equivalence, we establish the notion of an *extension of a configuration*, which formalizes the extraction of subexpressions from the expression of a configuration into fresh bindings of the corresponding heap. For this purpose, we define the application of substitutions to configurations as

$$\begin{aligned} \sigma^*(\Gamma : e) &= \sigma^*(\Gamma) : \sigma^*(e) \\ \sigma^*(\Gamma) &= \{(x, \sigma^*(b) \mid (x, b) \in \Gamma\} \\ \sigma^*(\text{free}) &= \text{free} \\ \sigma^*(\blacksquare) &= \blacksquare \end{aligned}$$

We can then define the extension of a configuration with respect to a single variable, as well as the equality of configurations up to multiple extensions.

Definition 7.26 (Extension of Configuration)**.** *A configuration $\Gamma[x \mapsto e''] : e$ such that $e'' \notin \mathcal{V}$, $x \notin \mathcal{V}ar(e'')$, and x occurs at most once in $\mathcal{V}ar_M(\Gamma : e)$ is called an x-extension of the configuration $\Gamma' : e' = \sigma(\Gamma) : \sigma(e)$ for $\sigma = \{x \mapsto e''\}$, denoted by $\Gamma[x \mapsto e''] : e >_x \Gamma' : e'$.*

A configuration C is equal up to x-extension to a configuration C', denoted by $C \geqslant_x C'$, if and only if $C = C'$ or $C >_x C'$. The equality up to extension is generalized to a sequence of pairwise different variables by

$$\begin{aligned} C \geqslant_\varepsilon C' &:\Leftrightarrow C = C' \,, \\ C \geqslant_{x \cdot \overline{x_n}} C' &:\Leftrightarrow C \geqslant_x C'' \wedge C'' \geqslant_{\overline{x_n}} C' \,. \end{aligned}$$

If the sequence $\overline{x_n}$ is of no further interest, we may write $C \geqslant_{\mathsf{ext}} C'$ instead of $C \geqslant_{\overline{x_n}} C'$.

Lemma 7.27 (Transitivity of Equality Up To Multiple Extension)**.** *If $C \geqslant_{\overline{x_n}} C'$ and $C' \geqslant_{\overline{y_m}} C''$, then $C \geqslant_{\overline{x_n} \cdot \overline{y_m}} C''$.*

Proof. By the definition of equality up to multiple extension. □

Two configurations that are equal up to multiple extension can then be considered as equivalent, since they can be evaluated to out-configurations which in turn are equal up to multiple extension.

Theorem 7.28 (Correctness of Equality Up To Multiple Extension)**.** *Let $\Gamma' : e'$ and $\Gamma : e$ be two configurations such that $\Gamma' : e' \geqslant_{\overline{x_n}} \Gamma : e$. Then $\Gamma' : e' \Downarrow_1 \Delta' : v$ if and only if $\Gamma : e \Downarrow_1 \Delta : v$ such that $\Delta' : v \geqslant_{\overline{x_n}} \Delta : v$.*

7.3.5 Correctness

Based on the previous considerations, we can state the formal results for the correctness of the refined definition of partial evaluation. We begin with the (partial)

correctness of partial evaluations, which states that if there exists a derivation in the original program, then there exists a corresponding derivation in the specialized program and vice versa, where the correspondence involves both equality up to multiple extension as well as the renaming applied in the specialized program.

Theorem 7.29 (Correctness of Partial Evaluation)**.** *Let P be a program, E a finite set of linear and partially evaluable expressions, ρ an independent renaming for E, P' a partial evaluation of E in P under ρ such that P' is E'-closed where $E' = \rho(E)$, and $\Gamma : e$ a configuration such that $\mathsf{ren}_\rho(\Gamma : e)$ is E'-closed. Then $\Gamma : e \Downarrow_1 \Delta : v$ in P if and only if $\mathsf{ren}_\rho(\Gamma : e) \Downarrow_1 \mathsf{ren}_\rho(\Delta' : v)$ in P' such that $\mathsf{ren}_\rho(\Delta' : v)$ is E'-closed and $\Delta' : v \geqslant_{\mathsf{ext}} \Delta : v$.*

Since the operational semantics evaluates an expression only to a head normal form, we furthermore like to show that an expression is not only evaluated to the same head normal forms in both the original program and its specialization (modulo renaming), but also that the bindings referenced in the head normal forms are evaluated to the same extent. However, we cannot employ the abstract semantics of expressions (Definition 5.16) for this purpose, since its definition considers `let` expressions by an insertion of the contained bindings into the heap. Because `let` expressions may be partially evaluable and thus affected by the recursive renaming, the order in which the abstract semantics computation and the recursive renaming are applied is relevant. To overcome this problem, we constrain the definition of the abstract semantics to only consider bindings of variables to *values* in the computed result, whereas a variable bound to an expression which is no value is abstracted. In consequence, only values are extracted from the heap, which is why we call it the abstract *value* semantics of an expression.

Definition 7.30 (Abstract Value Semantics)**.** *The abstract value semantics $[\![e]\!]_i^P$ of an expression e with respect to a program P and an operational semantics $\Downarrow_i$ is defined as*

$$[\![e]\!]_i^P := \{[\![\Gamma : v]\!] \mid [] : e \Downarrow_i \Gamma : v\}$$

$$[\![\Gamma : v]\!] := \begin{cases} \phi(\overline{\Gamma^*(x_k)}) & \text{if } v = \phi(\overline{x_k}) \text{ with } \phi \in \mathcal{C} \text{ or } k < \mathsf{arity}(\phi) \\ v & \text{otherwise} \end{cases}$$

where the recursive heap lookup operation $\Gamma^*(x)$ is defined as

$$\Gamma^*(x) = x \quad \text{if } x \notin \mathcal{D}om(\Gamma) \vee \Gamma(x) \in \{\mathsf{free}, \blacksquare\}$$

$$\Gamma[x \mapsto y]^*(x) = \begin{cases} x & \text{if } x = y \\ \Gamma^*(y) & \text{otherwise} \end{cases}$$

$$\Gamma[x \mapsto \phi(\overline{e_k})]^*(x) = \begin{cases} \phi(\overline{\Gamma[\overline{y_l \mapsto e'_l}]^*(x_k)}) & \text{if } \phi \in \mathcal{C} \text{ or } k < \mathsf{arity}(\phi) \\ x & \text{otherwise} \end{cases}$$

$$\text{where } (\overline{y_l}, \overline{e'_l}, \overline{x_k}) = \mathsf{splitArgs}(\overline{e_k})$$

$$\Gamma[x \mapsto \texttt{let } \{\overline{x_k = e_k}\} \texttt{ in } e]^*(x) = x$$

$$\Gamma[x \mapsto \texttt{let } \overline{x_k} \texttt{ free in } e]^*(x) = x$$
$$\Gamma[x \mapsto e_1 \texttt{ ? } e_2]^*(x) = x$$
$$\Gamma[x \mapsto \texttt{case } e \texttt{ of \{ } \overline{p_k \to e_k} \texttt{ \}}]^*(x) = x$$

Note that for the recursive cases of $\Gamma^*(x)$, the binding of x is either removed or replaced by bindings for subexpressions, such that the operation is well-defined.

To be able to show the equivalence of the abstract value semantics of an expression and its partial evaluation, we need the following auxiliary result stating that two configurations that are equivalent up to multiple extension share the same abstract value semantics.

Lemma 7.31 (Abstract Value Semantics under Equality up to Multiple Extension)**.** *Let $\Gamma' : v$ and $\Gamma : v$ be two configurations such that $\Gamma' : v \geqslant_{\overline{x_n}} \Gamma : v$ and v is a value. Then $[\![\Gamma' : v]\!] = [\![\Gamma : v]\!]$.*

The same applies for the process of renaming, which does not affect the abstract value semantics except for the renaming of partial function applications.

Lemma 7.32 (Abstract Value Semantics under Renaming)**.** *Let E be a finite set of linear and partially evaluable expressions, ρ an independent renaming for E, and $\Gamma : v$ a configuration such that v is a value and $\mathsf{ren}_\rho(\Gamma : v)$ is E'-closed, where $E' = \rho(E)$. Then $\mathsf{ren}_\rho([\![\Gamma : v]\!]) = [\![\mathsf{ren}_\rho(\Gamma : v)]\!]$.*

We can then show that partial evaluation preserves the abstract value semantics of an expression modulo renaming of partial function applications.

Theorem 7.33 (Abstract Value Semantics under Partial Evaluation)**.** *Let P be a program, E a finite set of linear and partially evaluable expressions, ρ an independent renaming for E, P' the partial evaluation of E in P under ρ such that P' is E'-closed for $E' = \rho(E)$, and e an expression such that $\mathsf{ren}_\rho(e)$ is E'-closed. Then $\mathsf{ren}_\rho([\![e]\!]_1^P) = [\![\mathsf{ren}_\rho(e)]\!]_1^{P'}$.*

Proof.

$$\begin{aligned}
& \mathsf{ren}_\rho([\![e]\!]_1^P) \\
= \quad & \text{(Definition 7.30)} \\
& \mathsf{ren}_\rho(\{[\![\Gamma : v]\!] \mid [] : e \Downarrow_1^P \Gamma : v\}) \\
= \quad & \text{(Theorem 7.29)} \\
& \mathsf{ren}_\rho(\{[\![\Gamma : v]\!] \mid [] : e \Downarrow_1^P \Gamma : v \wedge \Gamma' : v \geqslant_{\mathsf{ext}} \Gamma : v \wedge \mathsf{closed}(E', \mathsf{ren}_\rho(\Gamma' : v))\}) \\
= \quad & \text{(Lemma 7.31)} \\
& \mathsf{ren}_\rho(\{[\![\Gamma' : v]\!] \mid [] : e \Downarrow_1^P \Gamma : v \wedge \Gamma' : v \geqslant_{\mathsf{ext}} \Gamma : v \wedge \mathsf{closed}(E', \mathsf{ren}_\rho(\Gamma' : v))\}) \\
= \quad & \text{(renaming of set)} \\
& \{\mathsf{ren}_\rho([\![\Gamma' : v]\!]) \mid [] : e \Downarrow_1^P \Gamma : v \wedge \Gamma' : v \geqslant_{\mathsf{ext}} \Gamma : v \wedge \mathsf{closed}(E', \mathsf{ren}_\rho(\Gamma' : v))\}
\end{aligned}$$

$$
\begin{aligned}
&= && \text{(Lemma 7.32)}\\
&&& \{[\![\mathsf{ren}_\rho(\Gamma' : v)]\!] \mid [] : e \Downarrow_1^P \Gamma : v \wedge \Gamma' : v \geqslant_{\mathsf{ext}} \Gamma : v \wedge \mathsf{closed}(E', \mathsf{ren}_\rho(\Gamma' : v))\}\\
&= && \text{(Theorem 7.29)}\\
&&& \{[\![\mathsf{ren}_\rho(\Gamma' : v)]\!] \mid \mathsf{ren}_\rho([] : e) \Downarrow_1^{P'} \mathsf{ren}_\rho(\Gamma' : v)\}\\
&= && \text{(renaming of configuration)}\\
&&& \{[\![\mathsf{ren}_\rho(\Gamma' : v)]\!] \mid [] : \mathsf{ren}_\rho(e) \Downarrow_1^{P'} \mathsf{ren}_\rho(\Gamma' : v)\}\\
&= && \text{(Definition 7.30)}\\
&&& [\![\mathsf{ren}_\rho(e)]\!]_1^{P'} \qquad \square
\end{aligned}
$$

7.4 Partial Evaluation Procedure

We complete the development of the partial evaluation scheme with the presentation of the global specialization algorithm, based on the residualizing semantics and the notions introduced so far. Just like the original partial evaluation scheme presented in Chapter 6, the partial evaluation scheme is parametric with respect to

1. an *unfolding rule* to evaluate expressions using the residualizing semantics, and
2. an *abstraction operator* to ensure finiteness and closedness of the set of expressions to be partially evaluated.

We can directly adapt the definition of an unfolding rule from the narrowing-driven partial evaluation, whereas for the abstraction operator we additionally require the computed sequence to contain only linear and partially evaluable expressions.

Definition 7.34 (Unfolding Rule for FlatCurry)**.** *An unfolding rule $U(e, P)$ is a mapping which yields for a partially evaluable expression e its pre-partial evaluation $e \to e'$ in the program P. Given a set of expressions E, we denote by $U(E, P)$ the union of the pre-partial evaluations $U(e, P)$ for all $e \in E$.*

Definition 7.35 (Abstraction Operator for FlatCurry)**.** *Let q be a sequence of linear and partially evaluable expressions, E a finite set of expressions, and let S_q denote the set of expressions in q. An abstraction operator* abstract *is a mapping which returns a sequence $q' = \mathsf{abstract}(q, E)$ such that*

1. *if $e' \in S_{q'}$, then e' is linear and partially evaluable, and*
2. *if $e' \in S_{q'}$, then there exists an expression $e \in (S_q \cup E)$ such that $e|_p = \sigma(e')$ for some non-variable position p and substitution σ, and*
3. *for all $e \in (S_q \cup E)$, e is closed with respect to $S_{q'}$.*

Based on these definitions, we can adapt the concepts of the partial evaluation transition relation and the partial evaluation algorithm to our context as follows.

Definition 7.36 (Partial Evaluation Transition Relation for FlatCurry)**.** *We define the partial evaluation transition relation* $\longmapsto_{\mathcal{PE}}$ *as the smallest relation satisfying*

$$\frac{P' = U(S_q, P)}{q \longmapsto_{\mathcal{PE}} \mathsf{abstract}(q, \{e' \mid e \to e' \in P'\})}$$

Definition 7.37 (Partial Evaluation Algorithm)**.** *Let P be a program and E a finite set of expressions. We define the partial evaluation function $\mathcal{PE}$ as follows:*

$$\mathcal{PE}(P, E) = S_q \quad \textit{if } \mathsf{abstract}(\varepsilon, E) \longmapsto^*_{\mathcal{PE}} q \textit{ and } q \longmapsto_{\mathcal{PE}} q$$

where ε denotes the empty sequence of expressions.

The partial evaluation procedure basically operates as follows. Given a program P and a set of expressions E, it first extracts from E a sequence of linear and partially evaluable expressions, for which the corresponding pre-partial evaluations are computed. To achieve the closedness condition necessary for the correctness of partial evaluation, this process is iteratively repeated for the right-hand sides of the pre-partial evaluations by considering those subexpressions that are not closed with respect to the set of expressions already evaluated. Just like the basic algorithm presented in Chapter 6, this algorithm involves two levels of termination control. The local level addresses termination of the evaluation step (unfolding), while the global level addresses termination of the global iterative loop.

7.4.1 Local Termination

To come to a decision whether to defer or continue an evaluation in the residualizing semantics, the operation proceed takes as its arguments the current configuration and the dependency sequence that led to the configuration. In principle, sophisticated techniques such as the use of well-founded orderings as presented in Section 6.3 can be applied as a criterion. However, the presence of mutually recursive bindings and an infinite signature (e. g., integer numbers in Curry) may restrict their applicability. Therefore, we use the simpler approach of *one-step unfolding* allowing only one function call to be unfolded in every derivation.

Definition 7.38 (One-Step Unfolding)**.** *Let $\overline{(\Gamma_k : e_k)} \cdot (\Gamma : f(\overline{x_n}))$ be a non-empty dependency sequence with $f \in \mathcal{F}^{(n)}$. Then the operation* $\mathsf{proceed}_1$ *implementing a one-step unfolding is defined as:*

$$\mathsf{proceed}_1(\overline{(\Gamma_k : e_k)} \cdot (\Gamma : f(\overline{x_n}))) :\Leftrightarrow \nexists i \in \{1, \ldots, k\} : e_i = g(\overline{y_m}) \textit{ with } g \in \mathcal{F}^{(m)}$$

We then define $U_1(e, P)$ as the pre-partial evaluation of e under $\mathsf{proceed}_1$.

A crucial requirement for an unfolding rule is its termination, and we state that an evaluation in the residualizing semantics using $\mathsf{proceed}_1$ terminates.

Lemma 7.39 (Termination of One-Step-Unfolding)**.** *Let P be a program and e an expression. Then the computation of $U_1(e, P)$ terminates.*

Proof. Since the complexities of the in-configurations of a derivation without applications of rule (FunEval) strictly decrease according to Lemma 7.10 and the depth of every right-hand side in P must be finite, every infinite derivation must contain an infinite number of applications of rule (FunEval). This, however, is excluded by the assumption that $\mathsf{proceed}_1$ yields true only once. Furthermore, due to Lemma 7.10 there can only exist finitely many different derivations and the heaps of the resulting configurations must be finite, so that discomb as well as drf terminate. □

7.4.2 Global Termination

To ensure the global termination of the partial evaluation algorithm, we follow the approach of Alpuente, Falaschi, and Vidal [AFV98] to employ an abstraction operator based on the homeomorphic embedding relation in conjunction with the computation of most-specific generalizations to ensure finiteness of the set of expressions to be evaluated. However, due to the presence of additional syntactic constructs and non-determinism, we have to adapt these notions to our context.

Homeomorphic Embedding

The homeomorphic embedding as presented in Definition 6.22 is only defined for terms so that it has to be extended to cover FlatCurry expressions. The basic idea of this extension is to reduce any additional construct to a new special symbol, such that the original relation can be applied to this extended set of symbols. However, an important restriction of the homeomorphic embedding relation we have to obey is that it forms a well-founded quasi-ordering only for a *finite* set of symbols.

- In addition to fully saturated applications, FlatCurry also allows the *partial* application of function or constructor symbols. Since every symbol has a finite arity, it can only be applied to a finite number of argument expressions, which must be smaller or equal to the symbol's arity. We can therefore distinguish applications of a function or constructor symbol to different numbers of arguments by associating to an application $\phi(\overline{e_k})$ a symbol ϕ_k for all $k \in \{0, \ldots, \mathsf{arity}(\phi)\}$.
- For primitive types such as integer numbers, there exists an infinite number of constructor symbols. We therefore consider each number as a list of its digits, such that the number of different constructors becomes finite. For instance, we consider the number `123` as an abbreviation for the structure `1 : 2 : 3 : []`, and a negative number like `-42` as an abbreviation for `-(4 : 2 : [])`.
- A non-deterministic expression of the form e_1 `?` e_2 is considered as the application of a special symbol `?` to the expressions e_1 and e_2.
- A `case` expression of the form `case` e `of {` $\overline{p_k \to e_k}$ `}` is interpreted as an application of the special symbol $\mathtt{case}_{\overline{p_k}}$ to the arguments e and $\overline{e_k}$. Since the number of

constructor symbols is finite, so is the number of constructors of a certain type, and therefore the number of different pattern sequences $\overline{p_k}$ that can occur in `case` expressions.

- ▷ Unfortunately, it turns out that it is not possible to assign a special symbol to every `let` expression reflecting the number of bindings, since there is no fixed upper bound. Although the maximum number of bindings occurring in a program can easily be computed, the heap dereferencing mechanism may repeatedly introduce `let` bindings of increasing sizes. An ad-hoc solution would be to split every `let` binding into a sequence of mutually recursive bindings, such that the number of declarations in a single `let` expression gets considerably smaller. However, there is still the potential of an infinitely increasing number of mutually recursive bindings constructed by strict unifications, such that this approach may not work either. We therefore consider `let` expressions as an application of a special symbol `let` to a varying number of arguments.
- ▷ Although the introduction of free variables can be split into nested introductions of single variables, we adopt the idea for `let` bindings for consistency.

Based on this considerations, we refine the definition of comparable terms to the definition of *comparable expressions*.

Definition 7.40 (Comparable Expressions)**.** *Two expressions e and e' are comparable, denoted by* $\mathsf{comparable}(e, e')$*, if and only if the outermost constructor, function, or special symbols of e and e' coincide.*

To be able to cope with the varying number of arguments for `let` bindings, we use the concept of an *extended* homeomorphic embedding relation [Leu98] which can be parameterized over a well-founded binary relation for symbols as well as for sequences of terms.

Definition 7.41 (Extended Homeomorphic Embedding Relation [Leu98])**.** *Given a well-founded binary relation $\preceq_F$ on the set of function symbols and a well-founded binary relation $\preceq_S$ on sequences of terms, we define the extended homeomorphic embedding relation $\trianglelefteq^*$ on terms as the smallest relation satisfying the following conditions:*

1. $x \trianglelefteq^* y$ *for all* $x, y \in \mathcal{V}$,
2. $e \trianglelefteq^* \phi(\overline{e_k})$ *if* $e \trianglelefteq^* e_i$ *for some* $i \in \{1, \ldots, k\}$,
3. $\phi(\overline{e_m}) \trianglelefteq^* \psi(\overline{e'_n})$ *if* $\phi \preceq_F \psi$, $\overline{e_m} \preceq_S \overline{e'_n}$, *and* $\exists 1 \leqslant i_1 < \ldots < i_m \leqslant n$ *such that* $\forall j \in \{1, \ldots, m\} : e_j \trianglelefteq^* e'_{i_j}$.

Theorem 7.42 ($\trianglelefteq^*$ is a Well-Founded Binary Relation [Leu98])**.** *$\trianglelefteq^*$ is a well-founded binary relation on expressions. Additionally, if $\preceq_F$ and $\preceq_S$ are well-founded quasi orders, then so is $\trianglelefteq^*$.*

Since we have a finite alphabet of (special) symbols, we choose $\preceq_F$ to be the equality on symbols, and define the relation $\preceq_S$ on sequences as $\overline{e_m} \preceq_S \overline{e'_n} :\Leftrightarrow m \leqslant n$. Both $\preceq_F$ and $\preceq_S$ are well-founded quasi orders, and therefore $\trianglelefteq^*$ is also a well-founded quasi order according to Theorem 7.42. Furthermore, we can reformulate the extended homeomorphic embedding on FlatCurry expressions based on the previously discussed representation of syntactic constructs. For this purpose, we need to be able to obtain the direct subexpressions of an expression, which is achieved by means of the following operation.

Definition 7.43 (Direct Subexpressions)**.** *We define the set of all direct subexpressions of an expression e, denoted by $\mathcal{S}ub_1(e)$, inductively as:*

$$\begin{aligned}
\mathcal{S}ub_1(x) &= \varnothing \\
\mathcal{S}ub_1(\phi(\overline{e_k})) &= \{\overline{e_k}\} \\
\mathcal{S}ub_1(\texttt{let } \{\overline{x_k = e_k}\} \texttt{ in } e) &= \{e, \overline{e_k}\} \\
\mathcal{S}ub_1(\texttt{let } \overline{x_k} \texttt{ free in } e) &= \{e\} \\
\mathcal{S}ub_1(e_1 \texttt{ ? } e_2) &= \{e_1, e_2\} \\
\mathcal{S}ub_1(\texttt{case } e \texttt{ of \{ } \overline{p_k \to e_k} \texttt{ \}}) &= \{e, \overline{e_k}\}
\end{aligned}$$

We can then use this definition to reformulate the extended homeomorphic embedding on FlatCurry expressions.

Definition 7.44 (Homeomorphic Embedding on FlatCurry Expressions)**.** *The extended homeomorphic embedding relation $\trianglelefteq^*$ on FlatCurry expressions is defined as the smallest relation satisfying the following conditions:*

1. $x \trianglelefteq^* y$ *for all* $x, y \in \mathcal{V}$,

2. $e \trianglelefteq^* e'$ *if* $e \trianglelefteq^* e''$ *for some* $e'' \in \mathcal{S}ub_1(e')$,

3. $\phi(\overline{e_k}) \trianglelefteq^* \phi(\overline{e'_k})$ *if* $e_i \trianglelefteq^* e'_i$ *for all* $i \in \{1, \ldots, k\}$,

4. $\texttt{let } \{\overline{x_m = e_m}\} \texttt{ in } e \trianglelefteq^* \texttt{let } \{\overline{y_n = e'_n}\} \texttt{ in } e'$ *if* $e \trianglelefteq^* e'$, $m \leqslant n$, *and* $\exists 1 \leqslant i_1 < \ldots < i_m \leqslant n$ *such that* $\forall j \in \{1, \ldots, m\} : e_j \trianglelefteq^* e'_{i_j}$,

5. $\texttt{let } \overline{x_m} \texttt{ free in } e \trianglelefteq^* \texttt{let } \overline{y_n} \texttt{ free in } e'$ *if* $m \leqslant n$ *and* $e \trianglelefteq^* e'$,

6. $e_1 \texttt{ ? } e_2 \trianglelefteq^* e'_1 \texttt{ ? } e'_2$ *if* $e_1 \trianglelefteq^* e'_1$ *and* $e_2 \trianglelefteq^* e'_2$,

7. $\texttt{case } e \texttt{ of \{ } \overline{p_k \to e_k} \texttt{ \}} \trianglelefteq^* \texttt{case } e' \texttt{ of \{ } \overline{p'_k \to e'_k} \texttt{ \}}$ *if* $\overline{p_k} = \overline{p'_k}$, $e \trianglelefteq^* e'$, *and* $e_i \trianglelefteq^* e'_i$ *for all* $i \in \{1, \ldots, k\}$.

Generalization

In the narrowing-based partial evaluation framework, the concept of the *most-specific generalization* is used to generalize two comparable expressions related by the homeomorphic embedding. The generalization as well as the ranges of the corresponding substitutions are then independently considered for abstraction. Since we rely on the linearity of expressions for the renaming operation and the closedness condition, we restrict the generalization of two expressions to be linear as well.

Definition 7.45 (Most-Specific Linear Generalization)**.** *A generalization (g, σ, θ) of two expressions is linear if and only if g is a linear expression. A linear generalization (g, σ, θ) is the most specific linear generalization (mslg) if, for every other linear generalization (g', σ', θ') of e_1 and e_2, g' is more general than g.*

Lemma 7.46 (Uniqueness of Most Specific Linear Generalization)**.** *The most specific linear generalization of two expressions is unique up to variable renaming.*

Proof. Since every linear generalization of two expressions e_1 and e_2 is a generalization, and there exist only finitely many generalizations for two expressions [LMM88], there also exist only finitely many linear generalizations. Thus, let $\{g_1, \ldots, g_n\}$ denote the linear generalizations of e_1 and e_2. Then the greatest common instance g of the expressions $g_1, \ldots, g_n$ can be computed using the unification algorithm, which is a generalization of e_1 and e_2. Since $g_1, \ldots, g_n$ are all linear expressions, then so must be g, and g is therefore unique up to variable renaming. □

A further complication in the usage of the most specific (linear) generalization arises from the presence of `let` and `case` expressions. For the simpler structure of terms, the computation of the most specific generalization of two *comparable* terms leads to a *non-trivial generalization*, i. e., a generalization that is no variable. This is ensured by the fact that comparable terms share the same root symbol and the same number of arguments. In our context, however, two `let` expressions containing a different number of bindings are also considered as comparable, although there exists no non-trivial generalization due to the different number of bindings. For instance, there is no non-trivial generalization of the two comparable expressions `let {x = 1} in x` and `let {x = 1; y = 2} in x`. Furthermore, even in the case of the same number of bindings, the generalization may be trivial due to the occurrence of locally bound variables. For instance, consider the two comparable expressions

```
e1 = let {x = 1} in x + x
e2 = let {x = 1} in x * x
```

for which the generalization is $(g, \sigma, \theta) = (\mathtt{y}, \{\mathtt{y} \mapsto e_1\}, \{\mathtt{y} \mapsto e_2\})$ since the bound variable `x` must not occur in the range of the substitutions. In general, this problem may only occur for `let` and `case` expressions, but we still have to consider the possibility that the generalization of two comparable expressions yields a variable.

Non-Embedding Abstraction Operator

Based on the extended homeomorphic embedding relation on FlatCurry expressions and the most-specific linear generalization, we can now define the non-embedding abstraction operator for FlatCurry expressions. Note that in the residualizing semantics, deferred expressions and the branches of residual `case` expressions are annotated since they may be further evaluated, so that these annotations serve as a guidance for the abstraction operator to identify expressions suitable for further evaluation.

Definition 7.47 (Non-Embedding Abstraction Operator)**.** *Given a finite sequence of linear and partially evaluable expressions q and a finite set of expressions E, the abstraction operator $\mathsf{abs}^*_{\mathsf{emb}}(q, E)$ based on the extended homeomorphic embedding is defined as follows:*

$$\mathsf{abs}^*_{\mathsf{emb}}(q, E) = \begin{cases} q & \text{if } E = \varnothing \\ \mathsf{abs}^*_{\mathsf{emb}}(\ldots \mathsf{abs}_{\mathsf{emb}}(q, e_1), \ldots, e_n) & \text{if } E = \{e_1, \ldots, e_n\},\ n > 0 \end{cases}$$

The deterministic abstraction of a single expression is defined as

$$\mathsf{abs}_{\mathsf{emb}}(q, e) = \begin{cases} \mathsf{part}_{\mathsf{emb}}(q, f(\overline{e_k})) & \text{if } e = f(\overline{e_k}) \text{ or } e = \langle\!\langle f(\overline{e_k}) \rangle\!\rangle,\ f \in \mathcal{F}^{(n)},\ k < n \\ \mathsf{add}_{\mathsf{emb}}(q, \mathsf{lin}(e)) & \text{if } e = f(\overline{e_n}) \text{ with } f \in \mathcal{F}^{(n)} \\ \mathsf{add}_{\mathsf{emb}}(q, \mathsf{lin}(e')) & \text{if } e = \langle\!\langle e' \rangle\!\rangle \\ \mathsf{abs}^*_{\mathsf{emb}}(q, \mathcal{NVS}ub_1(e)) & \text{otherwise} \end{cases}$$

where $\mathsf{lin}(e)$ denotes the linearization of e and $\mathcal{NVS}ub_1(e)$ the set of its direct non-variable subexpressions. The abstraction of partial function calls is defined as

$$\mathsf{part}_{\mathsf{emb}}(q, f(\overline{e_k})) = \mathsf{abs}^*_{\mathsf{emb}}(q, \{f(\overline{x_n})\} \cup \{\langle\!\langle e_i \rangle\!\rangle \mid e_i \in \mathcal{NVS}ub_1(f(\overline{e_k}))\}) \\ \text{where } n = \mathsf{arity}(f) \text{ and } \overline{x_n} \text{ are fresh variables}$$

and the addition of an expression e to the sequence q is defined as

$$\mathsf{add}_{\mathsf{emb}}(q, e) = \begin{cases} q & \text{if } e \equiv e' \in q \\ q \cdot e & \text{if } \mathsf{peval}(e) \text{ and there is no } q_i \\ \mathsf{abs}^*_{\mathsf{emb}}(q, \{\langle\!\langle e' \rangle\!\rangle \mid e' \in \mathcal{NVS}ub_1(e)\}) & \text{if not } \mathsf{peval}(e) \text{ or } g \in \mathcal{V} \\ \mathsf{abs}^*_{\mathsf{emb}}(q \setminus q_i, S) & \text{otherwise} \end{cases}$$

where q_i is the last expression in q such that $\mathsf{comparable}(q_i, e)$ and $q_i \trianglelefteq^* e$, $(g, \sigma, \theta) = \mathsf{mslg}(q_i, e)$, $S = \{\langle\!\langle s \rangle\!\rangle \mid s \in \{g\} \cup \mathcal{R}an(\sigma) \cup \mathcal{R}an(\theta), s \notin \mathcal{V}\}$, and $q \setminus q_i$ denotes the removal of q_i from q.

The abstraction operator behaves similar to the initial abstraction operator, and proceeds as follows for a sequence q and an expression e to be added:

- ▷ (Annotated) partial function applications are handled by splitting them into a fully saturated function call applied to fresh variables, and considering this new function call and the annotated non-variable arguments recursively.

- ▷ Fully saturated function applications or annotated expressions are linearized and considered to be added to the sequence of expressions.
- ▷ For other expressions, their non-variable subexpressions are considered.

In consequence, only (partial) function calls and annotated expressions are considered for partial evaluation. The addition of a linear expression e to a sequence q of linear and partially evaluable expressions then proceeds as follows.

- ▷ An expressions that is a variant of another expression in the sequence is ignored.
- ▷ For an expression that is not partially evaluable, its non-variable subexpressions are annotated and recursively considered.
- ▷ Otherwise, the last expression q_i in the sequence that is comparable to e and embedded in e is determined. If there does not exist such an expression q_i and e is partially evaluable, then e is added to the sequence.
- ▷ If there exists such an expression q_i and e is partially evaluable, then the most-specific linear generalization (g, σ, θ) of q_i and e is computed, and the following decision is made based on g:
  - ▷ If the generalization g is a variable, which may occur if e is a `let` or `case` expression, then the non-variable subexpressions of e are annotated and recursively considered.
  - ▷ Otherwise, the expression q_i is removed from the sequence, and the generalization as well as the range of the substitutions are recursively considered. Like in the original abstraction operator presented in Definition 6.31, this prohibits an infinite loop in the abstraction process. Furthermore, the expressions to be further considered are annotated.

Note that applications of user-defined functions are always considered as partially evaluable, so that this check does not conflict with the requirement for closedness. For primitive operations, this definition furthermore may allow their evaluation if appropriate. The examination of annotated expressions regarding whether or not they are partially evaluable is necessary to avoid the consideration of expressions for which no pre-partial evaluation can be constructed, as the following example shows.

Example 7.48. *Consider the following expression:*

```
case x of { True → ⟪False⟫ }
```

If we would add `False` *to the set of expressions to be evaluated and consider the independent renaming* $\rho =$ `False` $\mapsto$ `false`*, then no pre-partial evaluation could be computed, and every occurrences of* `False` *would be renamed to the undefined function* `false`*. The hypothetical pre-partial evaluation* `False` $\rightarrow$ `False` *would not help either, since then the process of renaming would produce the partial evaluation*

```
false = false
```

and the constructor would thus be renamed to an infinite loop.

We turn our attention towards the correctness of the abstraction operator, where the following proposition states that the operation $\mathsf{abs}_{\mathsf{emb}}$ indeed is an abstraction operator in the sense of Definition 7.35, i. e., it adds only linear and partially evaluable expressions, does not invent new expressions, and ensures the correct propagation of the closedness property.

Proposition 7.49 ($\mathsf{abs}^*_{\mathsf{emb}}$ is an Abstraction Operator)**.** *The function* $\mathsf{abs}^*_{\mathsf{emb}}$ *is an abstraction operator in the sense of Definition 7.35, i. e., given a sequence q of linear and partially evaluable expressions and a finite set of expressions E, it satisfies the following conditions for* $q' = \mathsf{abs}^*_{\mathsf{emb}}(q, E)$*:*

1. *if $e' \in S_{q'}$, then e' is linear and partially evaluable, and*
2. *if $e' \in S_{q'}$, then there exists an expression $e \in (S_q \cup E)$ such that $e|_p = \sigma(e')$ for some non-variable position p and substitution σ, and*
3. *for all $e \in (S_q \cup E)$, e is closed with respect to $S_{q'}$*

where S_q denotes the set of expressions contained in the sequence q.

In addition to the correctness of the abstraction operator, its termination, which is stated in the following lemma, is also necessary for global termination.

Lemma 7.50 (Termination of $\mathsf{abs}^*_{\mathsf{emb}}$)**.** *The computation of the operator* $\mathsf{abs}^*_{\mathsf{emb}}$ *terminates.*

Proof. By well-founded induction on the complexity of the multiset union of the input arguments of $\mathsf{abs}^*_{\mathsf{emb}}$, which strictly decreases for every recursive invocation. □

We continue to address the termination of the global specialization process, using the above introduced abstraction operator. Since global termination relies on the finiteness of the sequences of comparable and non-embedding expressions, the operation $\mathsf{abs}^*_{\mathsf{emb}}$ must preserve the following non-embedding property.

Definition 7.51 (Non-Embedding Property [AFV98])**.** *Let $q = \overline{q_n}$ be a finite sequence of expressions. We say that q satisfies the non-embedding property if it holds that*

$$\forall k, l \in \{1, \ldots, n\} \text{ with } k < l : \mathsf{comparable}(q_k, q_l) \implies q_k \not\trianglelefteq^* q_l$$

This property basically states that in the sequence of partially evaluated expressions, no element is embedded by a subsequent element, so that the subsequence of comparable expressions must be finite since the extended homeomorphic embedding is a well-founded quasi-order.

Lemma 7.52 ($\mathsf{abs}^*_{\mathsf{emb}}$ preserves the Non-Embedding Property)**.** *Let q be a finite sequence of linear and partially evaluable expressions satisfying the non-embedding property, and E a finite set of expressions. Then $q' = \mathsf{abs}^*_{\mathsf{emb}}(q, E)$ satisfies the non-embedding property.*

We can furthermore generalize this lemma to the sequence of expression sequences that are computed by the partial evaluation algorithm.

Lemma 7.53. *Let $q_0, \ldots, q_n$ be the sequence of finite expression sequences computed by*

$$q_i \longmapsto_{\mathcal{PE}} \mathsf{abs}_{\mathsf{emb}}(q_i, E_i)$$

where $E_i = \{e' \mid e \in q_i \wedge e \to e' = \mathcal{U}(e, P)\}$ for a program P and unfolding rule $\mathcal{U}$, and $q_0 = \varepsilon$. Then q_i satisfies the non-embedding property for all $i \in 0, \ldots, n$.

Proof. By natural induction on the number n of sequences. For the base case of $n = 0$, $q_0 = \varepsilon$ trivially satisfies the non-embedding property. For the inductive case of q_{n+1}, we assume as the induction hypothesis that the claim holds for n. Thus, for $q_{n+1} = \mathsf{abs}^*_{\mathsf{emb}}(q_n, E_n)$, q_n satisfies the non-embedding property by the induction hypothesis. Since q_n is finite so must be E_n, and q_{n+1} then also satisfies the non-embedding property according to Lemma 7.52. □

Finally, we can state the termination of the entire partial evaluation algorithm, based on the correctness and termination results presented so far.

Theorem 7.54 (Termination of Partial Evaluation Algorithm)**.** *Let P be a program and E a finite set of expressions. The computation of the partial evaluation algorithm $\mathcal{PE}(P, E)$ terminates with the unfolding rule $\mathcal{U}_1$ and the abstraction operator $\mathsf{abs}^*_{\mathsf{emb}}$.*

Proof. This theorem directly follows from the following facts:

1. For every program P, the set of symbols is finite.

2. The number of sets of incomparable expressions that can be built from a finite set of symbols is finite.

3. Since the embedding relation $\trianglelefteq^*$ is a well-founded quasi order, the subsequences of comparable expressions satisfying the non-embedding property are finite.

4. The operation $\mathsf{abs}^*_{\mathsf{emb}}$ preserves the non-embedding property, so that the subsequences of comparable expressions of any sequence computed by $\mathcal{PE}(P, E)$ are finite.

5. The unfolding rule $\mathcal{U}_1$ terminates by Lemma 7.39.

6. The computation of the operation $\mathsf{abs}^*_{\mathsf{emb}}$ terminates by Lemma 7.50. □

7.4.3 Total Correctness

As the final step, we combine the correctness of partial evaluation, the correctness of the abstraction operator and the termination of the partial evaluation algorithm to finally state the total correctness of the partial evaluation algorithm.

Theorem 7.55 (Total Correctness of Partial Evaluation Algorithm)**.** *Let P be a program, e an expression, $q = \mathcal{PE}(P, \{e\})$ the sequence of linear and partially evaluable expressions computed using $\mathsf{abs}^*_{\mathsf{emb}}$ and U_1, E the set of expressions in q, ρ an independent renaming for E, and P' the partial evaluation of E in P under ρ. Then $\mathsf{ren}_\rho(\llbracket e \rrbracket_1^P) = \llbracket \mathsf{ren}_\rho(e) \rrbracket_1^{P'}$.*

Proof. By Theorem 7.54, the computation of $q = \mathcal{PE}(P, \{e\})$ terminates with $q = \mathsf{abs}^*_{\mathsf{emb}}(q, S)$, where $S = \{e'' \mid e' \to e'' \in U_1(E, P)\}$ by Definition 7.36 and Definition 7.37. Because $\mathsf{abs}^*_{\mathsf{emb}}$ is an abstraction operator by Proposition 7.49, the set S is E-closed. Furthermore, for $q_1 = \mathsf{abs}^*_{\mathsf{emb}}(\varepsilon, \{e\})$ it holds that e is S_{q_1}-closed, and $q_i \longmapsto q_{i+1}$ implies $S_{q_{i+1}}$-closedness of S_{q_i} by the definition of an abstraction operator, so that it follows from the transitivity of closedness (Lemma 7.19) that e is E-closed. Thus e and S are E-closed, so that $\mathsf{ren}_\rho(e)$ and P' are E'-closed by Lemma 7.23. Finally, $\mathsf{ren}_\rho(\llbracket e \rrbracket_1^P) = \llbracket \mathsf{ren}_\rho(e) \rrbracket_1^{P'}$ follows from Theorem 7.33. □

7.5 Summary

In this chapter we have presented a partial evaluation scheme for FlatCurry programs which can be considered as an instance of the original partial evaluation framework of Alpuente, Falaschi, and Vidal [AFV98] using the residualizing semantics for evaluation of expressions. Furthermore, we have shown the correctness and termination of the proposed partial evaluation scheme. The residualizing semantics and the general partial evaluation algorithm have been previously published in the *Proceedings of the 23rd International Workshop on Functional and (Constraint) Logic Programming (WFLP 2014)* as "A Partial Evaluator for Curry" [HP14], whereas the algorithm has only been informally sketched. Since then, the residualizing semantics has been revised, and the partial evaluation algorithm has been formalized and accompanied with the theoretical results regarding the correctness and termination of the algorithm.

The residualizing semantics follows the ideas of Albert, Hanus, and Vidal [AHV03], and the extraction of residual programs has been previously discussed by Fischer, Silva, Tamarit, and Vidal [FST+07], although for partial evaluation of a first-order functional language. Some adjustments with respect to the representation of programs were also present in the previous partial evaluator [AHV00; AHV02], whereas most adjustments arise from the additional consideration of non-deterministic operations and thus have not been covered before. Despite other work in the field of partial evaluation, the presented scheme is also close to existing work in the field of (positive) supercompilation of functional languages [Mit10; BP10], where the effects of one-step

unfolding and abstraction is achieved by means of driving and folding, respectively. Interestingly, Bolingbroke and Peyton Jones [BP10] consider configurations instead of expressions in their supercompilation scheme and argue that this eliminates some form of syntactic variance while simplifying the generalization mechanism, but the presented dereferencing mechanism in conjunction with the optimizations presented in the next chapter seem to achieve comparable results.

Since Curry programs can be normalized to the representation of FlatCurry, the partial evaluation scheme is already applicable for realistic Curry programs and can serve as the basis for a practical implementation of a partial evaluator. However, to achieve a good level of specializations, various optimizations have to be made, such as improvements in the process itself or in the representation of expressions. These optimizations and a practical implementation will be described in the next chapter.

Chapter 8

A Practical Partial Evaluator for Curry

Premature optimization is the root of all evil.

Donald Knuth

This chapter presents a practical partial evaluator for Curry programs, based on the partial evaluation scheme for FlatCurry introduced in Chapter 7. We first introduce the necessary extensions of the residualizing semantics to deal with primitive operations in general, and the application of higher-order functions, strict unification, and linear functional patterns in particular. We then continue to describe some important optimizations applied to the partial evaluation scheme that increase the efficiency of the obtained specializations. Finally, we sketch the implementation of the practical partial evaluator and present experimental results that advocate its applicability and usefulness.

8.1 Extensions of the Residualizing Semantics

The presented partial evaluation scheme can already be used for the evaluation of high-level Curry programs, thanks to the normalization process presented in Chapter 4 which translates Curry programs into the representation of FlatCurry. However, to achieve a good level of specialization for realistic Curry programs the explicit consideration of primitive functions is often necessary, as the following example demonstrates.

Example 8.1 (Need for Consideration of Primitives)**.** *If we consider the FlatCurry program*

```
foldr(f, e, xs) = case xs of { []       → e
                             ; (y : ys) → apply(apply(f, y), foldr(f, e, ys)) }
```

and the expression `foldr(+, 0, 1 : 2 : [])`*, we obtain the following resultant after evaluation of the function call and the subsequent* `case` *expression:*[1]

```
foldr(+, 0, 1 : 2 : [])  →  apply(apply(+, 1), foldr(+, 0, 2 : []))
```

[1] We inline the `let`-bindings introduced by dereferencing for better readability.

If we would treat the primitive operations `apply` *and* `+` *like constructors, only the nested call to* `foldr` *could be further evaluated, and we would obtain the following resultants:*

```
foldr(+, 0, 1 : 2 : [])  →  apply(apply(+, 1), foldr(+, 0, 2 : []))
foldr(+, 0, 2 : [])      →  apply(apply(+, 2), foldr(+, 0, []))
foldr(+, 0, [])          →  0
```

In consequence, the best result we can expect is the partial evaluation

```
sum = apply(apply(+, 1), apply(apply(+, 2), 0))
```

which leaves much room for improvements.

We will therefore extend the residualizing semantics to consider primitive operations in the following. In the presentation of the operational semantics in Chapter 5, we have already covered primitive operations for higher-order application, strict unification, and linear functional patterns. In principle, their definitions can be directly transferred to the residualizing semantics, but some minor adjustments are necessary due to the additional requirements of partial evaluation.

Termination While the evaluation of most primitives such as integer arithmetics is known to terminate once the arguments are evaluated to head normal forms, this is not true for all primitives. Consider, for instance, the following expression:

```
let ones = 1 : ones in ones =:= ones
```

If we consider the strict equality `ones =:= ones`, its evaluation will result in the constraint `1 =:= 1 & ones =:= ones` to be evaluated. While the first constraint is immediately solvable, the second equals the initial constraint, which thus may cause an infinite loop. In consequence, the operation `proceed` must also be applied to restrict the evaluation of certain primitives.

Deferred Expressions If a primitive operation is applied to an argument whose evaluation was deferred, the obvious choice is to defer the evaluation of the application as well, just like in the rule (CaseDefer) for `case` expressions.

Residual Case Expressions In the residualizing semantics, bindings originating from residual `case` expressions are only propagated by means of rule (CaseCase), i. e., if a residual `case` is inspected by another `case` expression. However, in the context of primitive operations, it is sometimes crucial to propagate these bindings also among different arguments to achieve a good level of specialization. For instance, the expression `x + (case x of {1 → ⟪2⟫})` could be further evaluated if the binding `x ↦ 1` of the second argument is propagated to the first. We therefore lift residual `case` expressions from argument positions of primitive operations upwards if admissible. For example, the expression `x + (case x of {1 → ⟪2⟫})` can be transformed to `case x of {1 → ⟪x + 2⟫}`, and using some other optimizations, we may obtain the specialization `case x of {1 → 3}`.

Note that this lifting of residual `case` expressions cannot always be applied, due to the lazy semantics of FlatCurry. Thus, it will only be applied if the residual expression occurs at a *strict* argument position, i. e., as an argument whose evaluation is required for the function to yield a result.

Furthermore, this optimization is only possible if calls to primitive functions are not *flattened*, since flattening would replace the residual `case` expression by a fresh variable which then evaluates to itself. Therefore, we assume in the following that for primitive functions, flattening is only applied for arguments which are shared in the respective internal implementation.

Partial Information Finally, the implementation of primitive operations has to consider partial information in the form of unbound variables. Since the application of a primitive operation to unbound variables might not be expressible as a residual `case` expression, we extend the representation of residual values to consider this case:

$$\mathit{RValue} \ ::= \ \ldots \mid f(\overline{e_n}) \quad (\text{residual primitive}, f \in \mathcal{P}^{(n)})$$

Furthermore, we add two more rules covering the cases that either a variable binding or an expression scrutinized by a `case` expression evaluates to a residual primitive:

$$\text{(VarPrim)} \quad \frac{\Gamma[x \mapsto \blacksquare] : e \Downarrow_{PE} \Delta : f(\overline{e_n})}{\Gamma[x \mapsto e] : x \Downarrow_{PE} \Delta[x \mapsto f(\overline{e_n})] : \langle\!\langle x \rangle\!\rangle} \quad \text{where } e \notin \{\text{free}, \blacksquare\} \text{ and } f \in \mathcal{P}^{(n)}$$

$$\text{(CasePrim)} \quad \frac{\Gamma : e \Downarrow_{PE} \Delta : f(\overline{e_n})}{\Gamma : \texttt{case}\ e\ \texttt{of}\ \{\ \overline{p_k \to e_k}\ \} \Downarrow_{PE} \Delta : \texttt{case}\ f(\overline{e_n})\ \texttt{of}\ \{\ \overline{p_k \to \langle\!\langle e_k \rangle\!\rangle}\ \}} \quad \begin{array}{l}\text{where} \\ f \in \mathcal{P}^{(n)}\end{array}$$

Based on these considerations, we can extend the residualizing semantics to cover the evaluation of primitive operations as follows.

8.1.1 Primitive Operations

We generally assume that primitive operations require the prior evaluation of their arguments to a head normal form. Operations that deviate from this assumption can be implemented similarly, and the requirement for complete evaluation of arguments can be realized by means of the auxiliary primitive `nf` presented in Section 5.4.1. Hence, we restrict ourselves to the general case, and provide the following scheme for primitive operations:

$$\text{(Prim)} \quad \frac{\Gamma_0 : e_1 \Downarrow_{PE} \Gamma_1 : v_1 \quad \ldots \quad \Gamma_{n-1} : e_n \Downarrow_{PE} \Gamma_n : v_n}{\Gamma_0 : f(\overline{e_n}) \Downarrow_{PE} \Gamma_n : \mathsf{prim}(\Gamma_n, f, \overline{v_n})} \quad \begin{array}{l}\text{where } f \in \mathcal{P}^{(n)} \text{ and} \\ \mathsf{proceed}(\Gamma_0 : f(\overline{e_n}))\end{array}$$

Unlike in the operational semantics, we do not apply the auxiliary operation `hnf`, but directly evaluate the arguments from left to right to be able to access residual values in the definition of the auxiliary operation prim.

$$\mathsf{prim}(\Gamma, f, \overline{v_n}) = \begin{cases} \langle\!\langle f(\overline{v_n}) \rangle\!\rangle & \text{if } \exists i \in \{1,\ldots,n\} \text{ such that } v_i = \langle\!\langle e \rangle\!\rangle \\ \texttt{case}\ e'\ \texttt{of} & \text{if } i \text{ is the first } j \in \{1,\ldots,n\} \text{ such that} \\ \quad \{\ \overline{p_k \to \langle\!\langle f(es_k) \rangle\!\rangle}\ \} & v_i = \texttt{case}\ e'\ \texttt{of}\ \{\ \overline{p_k \to \langle\!\langle e'_k \rangle\!\rangle}\ \}, \text{ where} \\ & es_k = v_1,\ldots,v_{i-1},e'_k,v_{i+1},\ldots,v_n \\ f(\overline{v_n}) & \text{if } \exists i \in \{1,\ldots,n\} \text{ such that } v_i = g(\overline{e_n}) \text{ with} \\ & g \in \mathcal{P}^{(n)} \text{ or } v_i \in \mathcal{V} \text{ with } v_i \notin \mathcal{D}om(\Gamma) \\ f_{\mathcal{A}}(\Gamma, \overline{v_n}) & \text{otherwise, where } f_{\mathcal{A}} \text{ computes the} \\ & \text{semantics of } f \end{cases}$$

If any argument evaluates to a deferred expression, then the entire function call is deferred. Otherwise, if any argument evaluates to a residual `case` expression, then the leftmost residual `case` expression is lifted upwards, and if any argument evaluates to a residual primitive or an unbound variable, the entire call is residualized. Only if all arguments evaluate to head normal forms, the function call is evaluated according to the semantics $f_{\mathcal{A}}$ of the primitive operation. This operation may access the heap Γ to dereference any logic variable v_i that may be bound to a value in Γ because of strict unification or narrowing in the evaluation of an argument e_j with $j > i$.

8.1.2 Higher-Order Application

The definition of higher-order application is straightforward, thanks to the general considerations presented above. Note that the primitive function `apply` is strict only in its first argument so that the rule to be applied is determined by the value of the first argument. In particular, we have to consider the cases of partial calls, deferred expressions, residual expressions, unbound variables, and residual primitives.

(Apply) $$\frac{\Gamma : e_1 \Downarrow_{PE} \Delta : \phi(\overline{x_k}) \quad \Delta : \phi(\overline{x_k}, e_2) \Downarrow_{PE} \Theta : v}{\Gamma : \texttt{apply}(e_1, e_2) \Downarrow_{PE} \Theta : v} \quad \text{where } k < \mathsf{arity}(\phi)$$

(ApplyDefer) $$\frac{\Gamma : e_1 \Downarrow_{PE} \Delta : \langle\!\langle e \rangle\!\rangle}{\Gamma : \texttt{apply}(e_1, e_2) \Downarrow_{PE} \Delta : \langle\!\langle \texttt{apply}(e, e_2) \rangle\!\rangle}$$

(ApplyCase) $$\frac{\Gamma : e_1 \Downarrow_{PE} \Delta : \texttt{case}\ e'\ \texttt{of}\ \{\ \overline{p_k \to \langle\!\langle e'_k \rangle\!\rangle}\ \}}{\Gamma : \texttt{apply}(e_1, e_2) \Downarrow_{PE} \Delta : \texttt{case}\ e'\ \texttt{of}\ \{\ \overline{p_k \to \langle\!\langle \texttt{apply}(e'_k, e_2) \rangle\!\rangle}\ \}}$$

(ApplyPartial) $$\frac{\Gamma : e_1 \Downarrow_{PE} \Delta : v}{\Gamma : \texttt{apply}(e_1, e_2) \Downarrow_{PE} \Delta : \texttt{apply}(v, e_2)} \quad \begin{array}{l} \text{where } v = f(\overline{e_n}) \text{ with } f \in \mathcal{P}^{(n)}, \\ \text{or } v \in \mathcal{V} \text{ and } v \notin \mathcal{D}om(\Delta) \end{array}$$

The rule (Apply) for partial applications resembles the corresponding rule of the operational semantics, and since there is no risk of non-termination, the rule can be applied without restrictions. If the first argument evaluates to either a deferred or residual `case` expression, the annotation or `case` structure is lifted upwards, and in case of missing information, the evaluation is residualized.

8.1.3 Strict Unification

Since the unification operator is strict in both arguments, we can choose an arbitrary order for the evaluation of its arguments, and we decide to evaluate them in left to right order to comply with the definition in Section 5.4.3. However, we do not define the strict unification operator by means of `hnf` and `prim_su`, but instead evaluate the arguments directly to access any residual values. Note that each of the two arguments may evaluate to one of six kinds of residual values (logic variable, unbound variable, constructor call, deferred expression, residual primitive, residual `case` expression), resulting in 36 combinations in total. To tighten their presentation, we combine some cases and proceed in such an order that the most general cases come first. To begin with, we consider the case that the first argument is evaluated to either a deferred or residual `case` expression, and we lift the annotation or the `case` structure upwards by the following rules.

(EqDefer)
$$\frac{\Gamma : e_1 \Downarrow_{PE} \Delta : \langle\!\langle e \rangle\!\rangle}{\Gamma : e_1 \texttt{ =:= } e_2 \Downarrow_{PE} \Delta : \langle\!\langle e \texttt{ =:= } e_2 \rangle\!\rangle}$$

(EqCase)
$$\frac{\Gamma : e_1 \Downarrow_{PE} \Delta : \texttt{case } e' \texttt{ of \{ } \overline{p_k \rightarrow \langle\!\langle e'_k \rangle\!\rangle} \texttt{ \}}}{\Gamma : e_1 \texttt{ =:= } e_2 \Downarrow_{PE} \Delta : \texttt{case } e' \texttt{ of \{ } \overline{p_k \rightarrow \langle\!\langle e'_k \texttt{ =:= } e_2 \rangle\!\rangle} \texttt{ \}}}$$

In all other constellations, the progress of evaluation also depends on the value of the second argument. If it evaluates to a deferred or residual `case` expression, we lift it upwards like for the previous cases.

(EqDefer2)
$$\frac{\Gamma : e_1 \Downarrow_{PE} \Delta : v \quad \Delta : e_2 \Downarrow_{PE} \Theta : \langle\!\langle e \rangle\!\rangle}{\Gamma : e_1 \texttt{ =:= } e_2 \Downarrow_{PE} \Theta : \langle\!\langle v \texttt{ =:= } e \rangle\!\rangle}$$
where $v \in \mathcal{V}$, or $v = \phi(\overline{x_n})$ with $\phi \in \mathcal{C}^{(n)} \cup \mathcal{P}^{(n)}$

(EqCase2)
$$\frac{\Gamma : e_1 \Downarrow_{PE} \Delta : v \quad \Delta : e_2 \Downarrow_{PE} \Theta : \texttt{case } e' \texttt{ of \{ } \overline{p_k \rightarrow \langle\!\langle e'_k \rangle\!\rangle} \texttt{ \}}}{\Gamma : e_1 \texttt{ =:= } e_2 \Downarrow_{PE} \Theta : \texttt{case } e' \texttt{ of \{ } \overline{p_k \rightarrow \langle\!\langle v \texttt{ =:= } e'_k \rangle\!\rangle} \texttt{ \}}}$$
where $v \in \mathcal{V}$, or $v = \phi(\overline{x_n})$ with $\phi \in \mathcal{C}^{(n)} \cup \mathcal{P}^{(n)}$

If no argument evaluates to a deferred or residual `case` expression, but at least one argument evaluates to a residual primitive expression, then partial evaluation can make no further progress, and the strict unification is residualized.

(EqPrim)
$$\frac{\Gamma : e_1 \Downarrow_{PE} \Delta : v_1 \quad \Delta : e_2 \Downarrow_{PE} \Theta : v_2}{\Gamma : e_1 \texttt{ =:= } e_2 \Downarrow_{PE} \Theta : v_1 \texttt{ =:= } v_2}$$
where $v_1 = f(\overline{e_n})$ or $v_2 = f(\overline{e_n})$ with $f \in \mathcal{P}^{(n)}$,
$v_1, v_2 \neq \langle\!\langle e' \rangle\!\rangle$, and $v_1, v_2 \neq \texttt{case } e' \texttt{ of \{ } \overline{p_k \rightarrow \langle\!\langle e'_k \rangle\!\rangle} \texttt{ \}}$

We continue with those cases where at least one argument evaluates to an unbound variable. If the other argument also evaluates to a variable (either logic or unbound), then no further evaluation is possible and we residualize the entire unification.

$$\text{(EqUnbound)}\quad \frac{\Gamma : e_1 \Downarrow_{PE} \Delta : v_1 \quad \Delta : e_2 \Downarrow_{PE} \Theta : v_2}{\Gamma : e_1 \texttt{ =:= } e_2 \Downarrow_{PE} \Theta : v_1 \texttt{ =:= } v_2}$$
$$\text{where } v_1, v_2 \in \mathcal{V}, \text{ and } v_1 \notin \mathcal{D}om(\Theta) \text{ or } v_2 \notin \mathcal{D}om(\Theta)$$

If one argument evaluates to an unbound variable but the second argument evaluates to a constructor-rooted expression, then we are able to generate a residual `case` expression resembling the binding performed by the strict unification.

$$\text{(EqRes)}\quad \frac{\Gamma : e_1 \Downarrow_{PE} \Delta : x \quad \Delta : e_2 \Downarrow_{PE} \Theta : c(\overline{y_n})}{\Gamma : e_1 \texttt{ =:= } e_2 \Downarrow_{PE} \Theta : \texttt{case } x \texttt{ of \{ } c(\overline{x_n}) \rightarrow \langle\!\langle x_1 \texttt{ =:= } y_1 \texttt{ \& ... \& } x_n \texttt{ =:= } y_n \rangle\!\rangle \texttt{ \}}}$$
$$\text{where } x \in \mathcal{V} \text{ with } x \notin \mathcal{D}om(\Delta), \overline{x_n} \text{ fresh}$$

$$\text{(EqRes2)}\quad \frac{\Gamma : e_1 \Downarrow_{PE} \Delta : c(\overline{x_n}) \quad \Delta : e_2 \Downarrow_{PE} \Theta : y}{\Gamma : e_1 \texttt{ =:= } e_2 \Downarrow_{PE} \Theta : \texttt{case } y \texttt{ of \{ } c(\overline{y_n}) \rightarrow \langle\!\langle x_1 \texttt{ =:= } y_1 \texttt{ \& ... \& } x_n \texttt{ =:= } y_n \rangle\!\rangle \texttt{ \}}}$$
$$\text{where } y \in \mathcal{V} \text{ with } y \notin \mathcal{D}om(\Theta), \overline{y_n} \text{ fresh}$$

What remains is the consideration of those cases where both arguments evaluate to a logic variable or a constructor-rooted expression. These cases proceed in the same manner as in the operational semantics, i. e., they can be defined similar to `prim_su`, so that we skip their formal presentation. Note that an application of these rules must be controlled by the operation `proceed` to avoid the risk of non-termination (unless two logic variables should be unified). Finally, the concurrent constraint conjunction can be implemented analogously (lifting of deferral annotation and residual `case` expression), so that we omit its formal description as well.

8.1.4 Functional Patterns

The extension for functional patterns is analogous to the extension for strict unification, where the only noteworthy difference is the asymmetric evaluation of the arguments, since the lazy unification operator is strict only in its first argument. The rules from the operational semantics (again augmented with a termination check using `proceed`) can then be applied without further changes, and the additional rules to cover the special cases originating from partial evaluation are identical to those for the strict unification.

8.2 Optimizations of the Partial Evaluation Scheme

In addition to the extension of the residualizing semantics, there are still some optimizations necessary to obtain *good* specializations. These address different parts of the partial evaluation process, such as dereferencing, the construction of partial evaluations, or the abstraction mechanism, and we discuss them in the following.

8.2.1 Limited Flattening

The omission of flattening for primitive operations is justified by the fact that these operations do not share their arguments, i. e., they are assumed to be representable by a linear right-hand side. In consequence, residual case expressions as values of the respective arguments can be taken into account, which leads to better specializations. Naturally, this optimization is not limited to primitive operations, but can also be applied to user-defined operations, provided that the corresponding argument variable is not shared in the respective function body.

Example 8.2 (Limited Flattening)**.** *Consider the following definition of the Boolean negation:*

```
not(x) = case x of { True → False; False → True }
```

If we evaluate the expression `not(not(x))` *using the residualizing semantics, we may obtain the resultant*

```
not(not(x)) → let { y = case x of {True → ⟪False⟫; False → ⟪True⟫} }
                in  case y of {True → ⟪False⟫; False → ⟪True⟫}
```

where no annotated subexpression is partially evaluable. If we omit the flattening for the arguments of `not`*, we instead can obtain the resultant*

```
not(not(x)) → case x of
                 { True  → ⟪case False of { True → False; False → True }⟫
                 ; False → ⟪case True  of { True → False; False → True }⟫
                 }
```

where both annotated subexpressions can be further evaluated, enabling a much better specialization.

We therefore assume that the flattening of function calls is only applied for argument positions where the corresponding argument variable occurs more than once in the function's right-hand side. Furthermore, flattening can also be omitted for any argument expression that is known to be deterministic so that its duplication does not affect the set of values, although it might duplicate work. A simple approximation is to allow only the duplication of constructor expressions, i. e., expressions that only consist of variables, constructors and partial function calls:

$$\begin{array}{rcll} ce & ::= & x & (\text{variable}) \\ & | & c(\overline{ce_n}) & (\text{constructor application}, n = \mathsf{arity}(c)) \\ & | & \phi(\overline{ce_k}) & (\text{partial application}, k < \mathsf{arity}(\phi)) \end{array}$$

In fact, constructor expressions may partially be duplicated anyway if they are bound to a parameter variable which is repeatedly evaluated. However, the duplication of these expressions still destroys sharing, and thus may break cyclic structures and affect the run time behavior of programs (e. g., their memory consumption). Note that a better approximation can quite easily be achieved by a separate program analysis, and we will discuss this topic in more detail in Section 9.2.2.

8.2.2 Improved Dereferencing

The dereferencing operation converts a configuration into an expression by addition of the heap bindings to the configuration's expression using `let` expressions. However, the operation does not consider the occurrence of the bindings in the subjacent expression, so that all bindings are inserted into a single mutually-recursive declaration group, including unused bindings. To obtain simpler expressions, it is therefore reasonable to limit the set of inserted bindings to those that are referenced in the subjacent expressions, and furthermore separate them into individual declaration groups if appropriate. This can be seen as a form of *let-floating* [PPS96] and allows the independent consideration of subexpressions that reference disjoint sets of variables.

Example 8.3 (Improved Dereferencing)**.** *For instance, the configuration*

$$[\mathtt{x} \mapsto \mathtt{not(True)}, \mathtt{y} \mapsto \mathtt{not(False)}, \mathtt{z} \mapsto \mathtt{0}] : \mathtt{(x,\ y)}$$

could be dereferenced to the expression

```
(let x = not(True) in x, let y = not(False) in y)
```

so that the binding for `z` *is omitted and both subexpressions can be considered individually.*

This can be achieved by the addition of bindings at the most inwards position such that no binding is duplicated and every bound variable still refers to the original binding. For this purpose, the heap Γ can be split into a part Γ_S containing bindings shared by at least two subexpressions, and the remaining heap $\Gamma_{\overline{S}}$ such that $\Gamma = \Gamma_S \uplus \Gamma_{\overline{S}}$. The shared bindings are then introduced around the expression, whereas the unshared bindings are recursively added to the subexpressions. This is achieved by the following operation, where add denotes the original operation that inserts all bindings from a heap to an expression, and $\mathsf{reachable}(\Gamma, e)$ selects the subset of Γ that contains all bindings in Γ that are transitively reachable from e.

$$\mathsf{add}^*(\Gamma, e) = \begin{cases} \mathsf{add}(\mathsf{reachable}(\Gamma, e), e) & \text{if } e \in \mathcal{V} \\ \mathsf{add}(\Gamma_S, C[\overline{\mathsf{add}^*(\Gamma_{\overline{S}}, e_k)}]) & \text{if } e = C[\overline{e_k}] \text{ with direct subexpressions } \overline{e_k}, \\ & \Gamma = \Gamma_S \uplus \Gamma_{\overline{S}}, \text{ and } \Gamma_S \text{ contains only those} \\ & \text{bindings reachable from } e_i, e_j \in \overline{e_k} \text{ with } i \neq j \end{cases}$$

For the dereferencing of a residual `case` expression scrutinizing an unbound variable, it is furthermore beneficial to locally augment the heap with a binding of the unbound variable to the respective pattern [FST+07].

Example 8.4 (Heap Augmentation during Dereferencing)**.** *For the configuration*

$$\Gamma : \mathtt{case}\ x\ \mathtt{of\ \{\ True} \to \langle\!\langle \mathtt{not(x)} \rangle\!\rangle\ \mathtt{\}}$$

where $x \notin \mathcal{D}om(\Gamma)$ it is beneficial to add the binding $x \mapsto$ `True` *to the heap Γ before the dereferencing of the nested expression $\langle\!\langle$*`not(x)`*$\rangle\!\rangle$, so that dereferencing yields the expression*

```
case x of { True → ⟪let y = True in not(y)⟫ }
```

This implements the propagation of information for unbound variables like in positive supercompilation, and increases the quality of specializations. Both optimizations can be integrated into the dereferencing operation by

$$\mathsf{drf}(\Gamma, e) = \begin{cases} \langle\!\langle \mathsf{drf}(\Gamma, e') \rangle\!\rangle & \text{if } e = \langle\!\langle e' \rangle\!\rangle \\ \texttt{case}\ x\ \texttt{of} & \text{if } e = \texttt{case}\ x\ \texttt{of}\ \{\ \overline{p_k \to e_k}\ \},\ x \notin \mathcal{D}om(\Gamma) \\ \quad \{\ \overline{p_k \to \mathsf{drf}(\Gamma[x \mapsto p_k], e_k)}\ \} & \\ \mathsf{add}^*(\Gamma, e) & \text{otherwise} \end{cases}$$

where we assume a renaming of bound variables to obey the variable convention.

8.2.3 Normalization of Expressions

After the process of dereferencing and the combination of multiple resultants by means of the operation discomb, we obtain a single expression as the pre-partial evaluation. This expression is then supplied to the abstraction operator, which identifies subexpressions for further evaluation and ensures closedness of the set of partially evaluated expressions. To achieve a good level of specialization, it is desirable to avoid unnecessary complex expressions to increase the chances for the identification of variants. We therefore apply the following normalization steps until a fixpoint is reached, so that a unified representation of expressions is obtained.

Removal of Unused Bindings

The removal of unused bindings, sometimes called *garbage collection*, is beneficial since it increases the possibility to detect variants of expressions. Furthermore, a decreased number of bindings may speed-up the entire process of partial evaluation, since less expressions have to be considered. Note that this idea has already been integrated into the dereferencing operation, but may still be applied for the results of other transformations. We remove unused bindings by the transformation

$$\begin{aligned} \texttt{let}\ \overline{x_k}\ \texttt{free in}\ e &\to \texttt{let}\ \{\overline{x_k}\} \cap \mathcal{UV}(e)\ \texttt{free in}\ e \\ \texttt{let}\ \{\overline{x_j = e_j}; x = e'; \overline{y_k = e'_k}\}\ \texttt{in}\ e &\to \texttt{let}\ \{\overline{x_j = e_j}; \overline{y_k = e'_k}\}\ \texttt{in}\ e \\ &\qquad \text{if } x \notin \mathcal{UV}(\{e, \overline{e_j}, \overline{e_k}\}), \text{ where } j, k \geqslant 0 \end{aligned}$$

where $\texttt{let}\ \varepsilon\ \texttt{free in}\ e$ and $\texttt{let}\ \{\varepsilon\}\ \texttt{in}\ e$ are considered as complex notations for e.

Example 8.5 (Removal of Unused Bindings)**.** *Consider the program*

```
const(x, y) = x
```

and the pre-partial evaluation

$$\texttt{const True exp} \to \texttt{let y = exp in True}$$

where exp *is expensive to compute. Then the abstraction operator might consider the evaluation of* exp*, but not for the simplified pre-partial evaluation* const True exp → True*.*

Inlining of Bindings

Another reasonable optimization is the inlining of `let` bindings [PM02]. Note that inlining is not limited to non-shared bindings, but may also be applied to non-recursive bindings to constructor expressions, which is sometimes necessary to obtain a good specialization. The inlining of `let` bindings is achieved by the transformation

$$\texttt{let } \{\overline{x_j = e_j}; x = e'; \overline{y_k = e'_k}\} \texttt{ in } e \to \texttt{let } \{\overline{x_j = \sigma(e_j)}; \overline{y_k = \sigma(e'_k)}\} \texttt{ in } \sigma(e)$$
if x occurs at most once in $e \cdot \overline{e_j} \cdot \overline{e'_k}$, or e' is a constructor expression with $x \notin \mathcal{V}ar(e')$, where $\sigma = \{x \mapsto e'\}$

The usefulness of this transformation is demonstrated by the following example, where the inlining enables a much better specialization.

Example 8.6 (Inlining)**.** *Consider the program*

```
map(f, xs) = case xs of { [] → [] ; (y : ys) → apply(f, y) : map(f, ys) }
square(x)  = x * x
```

and the following simplified pre-partial evaluation of the expression `map(square, xs)`*:*

```
case xs of { [] → []; (y : ys) → ⟪let {f = square} in apply(f, y) : map(f, ys)⟫}
```

Since the annotated expression is not partially evaluable, abstraction will recursively consider the expressions `square`, `apply(f, y)` *and* `map(f, ys)`*, leading to a poor specialization. If we instead inline the binding for* `f`*, we obtain the pre-partial evaluation*

```
case xs of { [] → []; (y : ys) → ⟪apply(square, y) : map(square, ys)⟫}
```

so that abstraction will now consider the expressions `apply(square, y)` *and* `map(square, ys)`*, enabling a much better specialization.*

Order of Bindings

Another source of variance between equivalent expressions is the order of local bindings. For instance, the expressions

```
let {x = 1; y = 2} in x + y
let {y = 2; x = 1} in x + y
```

are not variants of each other, but can nevertheless be considered as equivalent. We therefore rearrange declarations to the order of their occurrence in the subjacent expression. Furthermore, the alternatives of `case` expressions are reordered so that the patterns occur in the order of their definition in the respective data type.

Removal of Failing Alternatives

Further normalizations are possible for `case` expressions and the non-deterministic choice, where failing alternatives can be removed. If all branches of a `case` of non-

deterministic expression fail, then so does the entire expression, which is thus transformed to `failed` accordingly.

$$\texttt{case}\ e\ \texttt{of}\ \{\ \overline{p_k \to e_k}\ \} \to \texttt{case}\ e\ \texttt{of}\ \{\ (p_i \to e_i \in \overline{p_k \to e_k} \mid e_i \neq \texttt{failed})\ \}$$
$$\texttt{case}\ e\ \texttt{of}\ \{\ \varepsilon\ \} \to \texttt{failed}$$
$$\texttt{failed}\ \texttt{?}\ e_2 \to e_2$$
$$e_1\ \texttt{?}\ \texttt{failed} \to e_1$$

Application of Rule (Select)

If a `case` expression scrutinizes a constructor-rooted expression, then an application of rule (Select) can be simulated to transform the expression to the respective alternative. Formally, we perform the following transformation, where $c \in \mathcal{C}^{(n)}$:

$$\texttt{case}\ c(\overline{e_n})\ \texttt{of}\ \{\ \overline{p_k \to e'_k}\ \} \to \begin{cases} \texttt{let}\ \{\overline{x_n = e_n}\}\ \texttt{in}\ e'_i & \text{if } \exists i \in \{1,\ldots,k\} : p_i = c(\overline{x_n}) \\ \texttt{failed} & \text{otherwise} \end{cases}$$

Lifting of Expressions below Non-Determinism

During the partial evaluation of non-deterministic programs, the situation may arise that different resultants of the same expression are constrained by the same condition, i. e., the resultants $\overline{e_n}$ are of the form $c\ \texttt{\&>}\ e'_i$ where the condition c is identical for each resultant e_i. Because resultants are combined by a non-deterministic choice, the pre-partial evaluation is then of the form $c\ \texttt{\&>}\ e_1\ \texttt{?}\ \ldots\ \texttt{?}\ c\ \texttt{\&>}\ e_n$. Since the condition c is evaluated if any of the alternatives is evaluated, it can be safely moved upwards by the transformation

$$(c\ \texttt{\&>}\ e_1)\ \texttt{?}\ (c\ \texttt{\&>}\ e_2) \to c\ \texttt{\&>}\ (e_1\ \texttt{?}\ e_2)$$

which does not only reduce the size of the expression, but may also allow further optimizations of the alternatives.

Another important optimization for non-deterministic alternatives is the lifting of subjacent `case` expressions scrutinizing the same expression. That is, for an expression

$$\texttt{case}\ e\ \texttt{of}\ \{\ \overline{p_k \to e_k}\ \}\ \texttt{?}\ \texttt{case}\ e\ \texttt{of}\ \{\ \overline{p'_k \to e'_k}\ \}\ ,$$

we can lift the `case` expression upwards by means of the transformation

$$\texttt{case}\ e\ \texttt{of}\ \{\ \overline{p_k \to e_k}\ \}\ \texttt{?}\ \texttt{case}\ e\ \texttt{of}\ \{\ \overline{p'_k \to e'_k}\ \} \to \texttt{case}\ e\ \texttt{of}\ \{\ \mathsf{merge}(\overline{p_k \to e_k}, \overline{p'_k \to e'_k})\ \}$$

where the operation `merge` is defined as

$$\mathsf{merge}(\varepsilon, bs') = bs'$$
$$\mathsf{merge}((p \to e) \cdot bs, bs') = \begin{cases} (p \to e\ \texttt{?}\ e') \cdot \mathsf{merge}(bs, bs' \setminus (p \to e')) & \text{if } p \to e' \in bs' \\ (p \to e) \cdot \mathsf{merge}(bs, bs') & \text{otherwise} \end{cases}$$

Note that if the sets of patterns of the `case` expressions are disjoint, then this optimization may transform a non-deterministic expression into a deterministic one.

8.2.4 Non-Linear Expressions

A major disadvantage of the current partial evaluation scheme is its restriction to linear expressions in order to avoid the unintended introduction of sharing for subexpressions during renaming. While this restriction is generally necessary for correctness, it does not apply if the respective subexpression is deterministic, and hence in particular for constructor expressions. We therefore relax the recursive renaming to also rename an expression $e = \sigma(e')$ to a function call of $f_{e'}$ if for every variable $x \in \mathcal{D}om(\sigma)$ it holds that $\sigma(x)$ is a constructor expression if x occurs more than once in e'. In addition, the abstraction operator can be improved to no longer require the linearization of expressions, and for the most-specific generalization the requirement for linearity can be relaxed so that the generalization is required to be linear only in those variables not substituted by constructor expressions.

Example 8.7 (Non-Linear Generalization)**.** *For instance, the two expressions* `f(True, True)` *and* `f(False, False)` *can be generalized to*

$$(\texttt{f(x, x)}, \{\texttt{x} \mapsto \texttt{True}\}, \{\texttt{x} \mapsto \texttt{False}\})$$

instead of the less specific linear generalization

$$(\texttt{f(x, y)}, \{\texttt{x} \mapsto \texttt{True}, \texttt{y} \mapsto \texttt{True}\}, \{\texttt{x} \mapsto \texttt{False}, \texttt{y} \mapsto \texttt{False}\})\,.$$

8.2.5 Abstraction of Partial Function Calls

In the definition of the non-embedding abstraction operator, partial function calls are abstracted by separately considering the function call and the provided non-variable arguments. Although this definition leads to a correct specialization, it causes a loss of information, as the following example demonstrates.

Example 8.8 (Loss of Information for Partial Function Calls)**.** *Consider the program*

```
f(x, y, z) = case x == y of { True → z }
```

with the expression `f(1, 1, z)` *and its pre-partial evaluation* `f(1, 1, z)` → `z`*. If we consider the partial call* `f(1, 1)` *during abstraction, then* `f(x, y, z)` *might be evaluated. The partial evaluation then is*

```
f1(x)       = x
f2(x, y, z) = case x == y of { True → z }
```

and `f(1, 1)` *is renamed to* `f2(1, 1, z)`*, although the renaming to* `f1` *is more desirable.*

To eliminate this disadvantage, we modify the abstraction operator to not split up partial function calls but consider them in entirety. For this purpose, we introduce a *completion* operation to extend a partial function call using fresh variables.

Definition 8.9 (Completion of Partial Function Call)**.** *The completion of a partial function call $f(\overline{e_k})$ is defined as*

$$\mathsf{complete}(f(\overline{e_k})) = f(\overline{e_k}, \overline{x_{n-k}}) \quad \textit{where } f \in \mathcal{F}^{(n)} \textit{ and } \overline{x_{n-k}} \textit{ fresh}$$

The abstraction operator can then be changed to consider *completed* partial function calls instead of their deconstruction:

$$\mathsf{part}_{\mathsf{emb}}(q, f(\overline{e_k})) = \mathsf{add}_{\mathsf{emb}}(q, \mathsf{complete}(f(\overline{e_k})))$$

To further guarantee the correctness of the partial evaluation scheme, the definition of closedness and the recursive renaming operation have to be modified accordingly. That is, a partial function call is closed if and only if its completion is closed, and the renaming of a partial function call is the renaming of its completion, where the fresh variables added during completion are afterwards removed again. Thus, for the example above, `f(1, 1)` is first completed to `f(1, 1, z)`, which is renamed to `f1(z)`, and then `z` is removed to obtain the partial call `f1`.

Note that this changed renaming implies that a partial function call might be renamed differently than its flattened counterpart. In consequence, the renaming of the value of an expression might differ from the value of the renamed expression, which conflicts with the correctness results of partial evaluation stated in Chapter 7. However, this only affects partial function calls, so that different renamings are not observable in practice.

8.2.6 Compression of Residual Functions

The one-step unfolding tends to produce multiple residual functions of which some can be considered simple, i. e., they have a trivial right-hand side, or intermediate, i. e., they directly call another residual function with the same arguments. Furthermore, there may also exist residual functions that are duplicates of each other, which arises if the right-hand sides of the pre-partial evaluations of two distinct expressions coincide. We therefore apply an additional post-processing phase to reduce the number of residual functions, which is achieved by a removal of duplicate functions and inlining of simple functions.

To start with, we define the conditions under which are residual function can be considered as a duplicate. Duplicates are then removed from the partial evaluation, while the independent renaming is updated accordingly, such that this compression does not affect closedness. Note that the removal of duplicates has only a positive effect on the resulting size of the program, but not on its efficiency.

Definition 8.10 (Duplicate Function)**.** *A function f is considered a duplicate of a function $g \neq f$ if and only if f and g are defined as $f(\overline{x_n}) = e$ and $g(\overline{x_n}) = e'$ and $e_{f \to g} = e'$, where $e_{f \to g}$ denotes the replacement of every function symbol f with g in e.*

After the removal of duplicate functions, there may still exist residual functions with a trivial right-hand side. For instance, partial evaluation of the expression `map(square, xs)` will produce the specialized program

```
mapSquare(xs) = case xs of { [] → []; (y : ys) → sq(x) : mapSquare(xs) }
sq(x)         = x * x
```

where the auxiliary function `sq` is used only once and has a simple right-hand side. Furthermore, some residual functions may also evaluate to constants such as `True`. We therefore perform an additional inlining phase where function calls to *inlineable* functions are replaced with the instantiated right-hand side, so that a call $f(\overline{e_n})$ to an inlineable function defined as $f(\overline{x_n}) = e$ is replaced by `let` $\{\overline{x_n = e_n}\}$ `in` e. In the literature on partial evaluation, this optimization is often referred to as *post-unfolding*, and it generally improves the efficiency of the resulting program since the number of function calls is decreased.

Definition 8.11 (Inlineable Function)**.** *In a program P, a function $f(\overline{x_n}) = e \in P$ is called inlineable if and only if it satisfies at least one of the following conditions:*

- *there are no (partial) function calls in e, or*
- *f is an alias to a function $g \neq f$, i. e., $e = g(\overline{y_m})$ with $\{\overline{y_m}\} \subseteq \{\overline{x_n}\}$, or*
- *f is non-recursive (there is no call to f in e) and f is called only once in P.*

Note that the resulting expression may be further optimized by the transformations presented before, such as the inlining of `let` bindings. Furthermore, after inlining of function calls there may exist residual functions that are neither called in the program nor occur in the independent renaming, and can thus be safely removed.

8.3 The Practical Partial Evaluator

The practical partial evaluator is supposed to specialize a set of expressions with respect to a given input program in order to create a residual program. To be able to provide the initial set of expressions, we assume these expressions to be annotated within the input program by means of the special unary function `PEVAL :: a -> a`. The partial evaluator then recognizes the set of annotated expressions and produces a residual program for this set. To also allow the regular evaluation of annotated programs, `PEVAL` behaves like the identity function so that annotations with `PEVAL` do not change the semantics of the original program, i. e., it is defined in the module `Prelude` as

```
PEVAL :: a -> a
PEVAL x = x
```

Example 8.12 (Annotation of Expression)**.** *In the Curry program*

```
map :: (a -> b) -> [a] -> [b]
map _ []       = []
map f (x : xs) = f x : map f xs

square :: Int -> Int
square x = x * x

main :: [Int] -> [Int]
main xs = PEVAL (map square xs)
```

the expression `map square xs` *is annotated and will thus be considered for partial evaluation.*

The annotations have to be provided manually by the user, which allows experiments regarding the usefulness of partial evaluation for certain expressions. Nevertheless, it would also be possible to automatically annotate a given program based on a heuristic strategy. To allow the partial evaluation of functions defined using functional patterns, an auxiliary function is necessary. For instance, the definition of the function `last` can be partially evaluated using the auxiliary function

```
last' :: [a] -> a
last' xs = PEVAL (last xs)
```

Since the introduction of such auxiliary functions may sometimes be cumbersome, the partial evaluator is furthermore configurable to consider all functions defined using functional patterns for partial evaluation.

After annotation of the program, the process of partial evaluation is fully automatic, and will result in the computation of the residual program. The process itself consists of the following phases, which are depicted in Figure 8.1.

Normalization The partial evaluator is called with the name of an arbitrary Curry program, which should contain annotated expressions as described above. This source program is first converted into its FlatCurry representation, according to the normalization process presented in Chapter 4.

Extraction The process continues with an extraction of the set of annotated expressions and creates a copy of the original program without annotations.

Evaluation Both the annotated expressions and the unannotated program form the input for the partial evaluation phase, in which the expressions are evaluated with respect to the program by means of an unfolding rule, and the process of abstraction ensures closedness of the set of evaluated expressions.

Renaming The output of the partial evaluation phase is a set of pre-partial evaluations. By the post-processing step of *renaming*, these pre-partial evaluations are converted into definitions of residual functions, so that the partial evaluation is obtained.

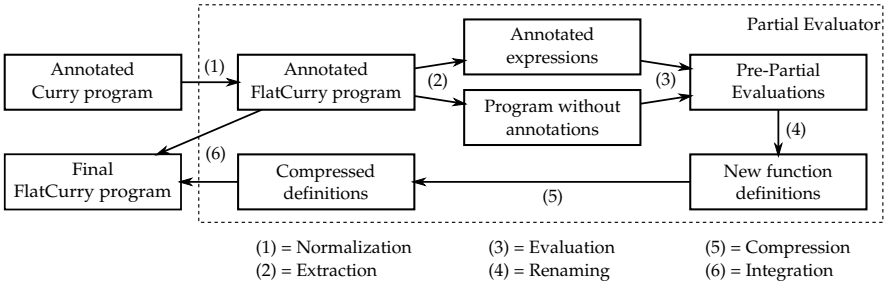


Figure 8.1. Overview of the Partial Evaluation Process

Compression The partial evaluation is then compressed by removal of duplicate functions and the inlining of simple functions to obtain a better specialization.

Integration Finally, the annotated expressions in the original program are renamed to calls of the residual functions, which in turn are appended to the original program. Note that the original function definitions are kept in the specialized program in order to allow the evaluation of functions which have not been annotated, and this program is then stored as a new FlatCurry program.

To illustrate the different phases of the partial evaluation process, we consider the specialization of the program of Example 8.12 and present the intermediate results.

Example 8.13 (Partial Evaluation Process)**.** *For the Curry program of Example 8.12, the partial evaluation process yields then following intermediate results.*

Normalization: The resulting FlatCurry program is

```
map(f, xs) = case xs of { [] → []; (y : ys) → apply(f, y) : map(f, ys) }
square(x)  = x * x
main(xs)   = PEVAL(map(square, xs))
```

Extraction: The set of annotated expressions is {`map(square, xs)`}*, and the unannotated program equals the program above with the call to* `PEVAL` *removed.*

Evaluation: The evaluation phase follows the partial evaluation algorithm presented in Chapter 7 in conjunction with the above mentioned extensions and optimizations and yields the following set of simplified pre-partial evaluations:

```
map(square, xs) → case xs of {
                    [] → [];
                    (y : ys) → square(y) : map(square, ys) }
square(y) → y * y
```

Renaming: For the independent renaming

$$\rho = \{\texttt{map(square, xs)} \mapsto \texttt{mapSquare(xs)},\ \texttt{square(y)} \mapsto \texttt{sq(y)}\},$$

the process of renaming yields the following partial evaluation P:

```
mapSquare(xs) = case xs of { [] → []; (y : ys) → sq(y) : mapSquare(ys) }
sq(y)         = y * y
```

Compression: The specialized program P is compressed by inlining of the definition of `sq` *and a subsequent simplification, and we obtain the following compressed program:*

```
mapSquare(xs) = case xs of { [] → []; (y : ys) → (y * y) : mapSquare(ys) }
```

Integration: The annotated expression `map(square, xs)` *is replaced by a call to the residual function* `mapSquare`*, thus yielding the following result program:*

```
map(f, xs)    = case xs of { [] → []; (y : ys) → apply(f, y) : map(f, ys) }
square(x)     = x * x
main(xs)      = mapSquare(xs)
mapSquare(xs) = case xs of { [] → []; (y : ys) → (y * y) : mapSquare(ys) }
```

The evaluation phase is parametric with respect to the employed unfolding rule and abstraction operator, which are responsible to ensure local and global termination, respectively. Regarding local termination, the partial evaluator provides the strategy of one-step unfolding as presented in Definition 7.38, as well as two supplemental strategies.

One At most one function call is unfolded in each evaluation. This strategy may lead to many expressions to be evaluated, and thus increases the reuse of residual functions, since subexpressions already evaluated in another context may be identified more often.

Each At most one call for each user-defined function is unfolded in each evaluation, and subsequent calls to a function are deferred. This strategy tends to create less residual functions, which may limit their reuse.

All All function calls are unfolded, so that this strategy cannot guarantee termination of the evaluation process. However, it may still be useful for experiments to determine the impact of the unfolding rule. Furthermore, it may become useful for partial evaluation of functions that are known to terminate but require a large number of intermediate function calls.

At the first sight, a strategy that allows less function calls to be evaluated in one derivation might seem inferior. If an evaluation is split into multiple derivations with deferred subexpressions, then each of these subexpressions has to be evaluated anew, which leads to a new residual function. Hence, more derivations will produce more

deferred subexpressions and, therefore, more residual functions. Nevertheless, there are better chances that some of these expressions have already been encountered before, reducing the overall number of expressions to be evaluated. Furthermore, the final compression phase will eliminate intermediate functions, so that even the simple strategy of one-step unfolding performs well in practice. Note that, regardless of the chosen strategy, built-in functions are always evaluated if they are known to terminate, such as the function `apply` or arithmetic functions.

Like the unfolding rule, the abstraction operator can be parameterized by a strategy to decide the option taken for expressions that are considered for further evaluation (add or generalize an expression). These strategies are usually based on some well-founded binary relation, and the partial evaluator supports the following strategies.

Homeomorphic Embedding Abstraction This strategy corresponds to the non-embedding abstraction operator of Definition 7.47, where termination is ensured by the generalization of expressions that embed a comparable expression which has been evaluated before.

Size-Based Abstraction This strategy uses a well-founded ordering on comparable expressions based on the strict ordering on their *syntactic size*, i. e., an expression is generalized if its size is strictly greater than the size of a comparable expression already evaluated. Because comparable expressions are also comparable with respect to their size, this abstraction operator is computationally less expensive than the homeomorphic embedding, since it suffices to compare the size of a new expression with the size of the last comparable expression previously evaluated.

No Generalization No generalization is performed, so that an expression is always added to the sequence of expressions to be evaluated unless it is a variant of some other expression. Naturally, this strategy implies the risk of non-termination and should therefore be used with care. On the other hand, the lack of generalization also avoids a loss of information so that the results will be most accurate. Hence, this strategy may be useful to determine the impact of generalization on the quality of the specialized program.

8.4 Experimental Results

We evaluate the implementation of our partial evaluator with some selected benchmarks grouped by the language features employed in the respective programs. We compiled both the partial evaluator and the benchmarks with the PAKCS Curry system (version 1.13.1, based on SICStus Prolog 4.3.2), and all benchmarks were executed on a Linux machine running Debian Jessie 8.2 with an Intel(R) Core(TM) i5-440 processor clocked at 3.10 GHz and 16 GiB of memory.

The time for partial evaluation of a program has been measured by the `time` command, and the timings for evaluation of an expression with respect to the original and specialized program were obtained using the profiling operation `profileTimeNF` of the PAKCS library `Profile`. This operation determines the time required for complete evaluation of the results, where the arguments have been completely evaluated beforehand. Thus, the timings reflect the time needed for the actual computation of the result to identify the speedup obtained by partial evaluation. To allow comparisons with the results of the original partial evaluator [AHV02], we also include the timings of the specialized program computed by the "`:peval`" command of the PAKCS system. All timings are given in seconds and were computed as the average of five subsequent runs to eliminate incidental effects.

The benchmark examples have generally been specialized with the strategy of one-step unfolding and with the non-embedding abstraction operator. Experiments with different unfolding strategies combined with the non-embedding abstraction, as well as different abstraction operators combined with the one-step unfolding, are considered afterwards.

8.4.1 First-Order Programs

We start our evaluation with some first-order programs and consider typical examples of partial deduction and functional program transformations such as supercompilation. Of these benchmarks specified below, the first three examples are functions working on lists or trees as (intermediate) data structures, while the last three functions operate on fixed input which is interpreted by the respective function.

- `doubleApp` is the concatenation of three lists presented in Example 6.1,
- `lengthApp` computes the length of the concatenation of two input lists,
- `doubleFlip` [Wad90] flips a binary tree twice, thus returning the initial tree,
- `power4` takes an integer number to the fourth power, based on the general exponentiation function presented in Example 1.1,
- `kmp` implements a generic string pattern matcher [SGJ96] and is specialized w.r.t. a fixed pattern containing three characters, producing a pattern matcher similar to one produced by the Knuth-Morris-Pratt algorithm [KMP77], and
- `automaton` [CL11] executes a state machine with two states.

The list-consuming functions are applied to an input list containing 500,000 elements (only `power4` is applied to a list containing 50,000 elements), and `doubleFlip` is applied to a balanced binary tree of depth 18. The results of the benchmarks are presented in Figure 8.2, where we show for each benchmark the time needed for partial evaluation (PE Time), the run time of the original program (Original) and of

Benchmark	PE Time	Original	NN-PE	Specialized	Speedup
`doubleApp`	0.80	0.26	0.21	0.21	1.24
`lengthApp`	0.79	0.20	0.14	0.14	1.43
`doubleFlip`	0.75	0.31	0.24	0.24	1.29
`power4`	1.03	1.21	n/a[a]	0.87	1.39
`kmp`	4.42	1.10	0.21	0.21	5.24
`automaton`	5.36	1.20	1.17	0.94	1.28

Figure 8.2. Benchmark Results for First-Order Programs

[a]The partial evaluator failed to produce a specialization within 60 seconds.

the specialized program obtained from partial evaluation based on needed narrowing (NN-PE), as well as the run time of the specialized program obtained from the partial evaluator presented in this work (Specialized) together with its obtained speedup (Speedup) compared to the original program.

First of all, we like to emphasize that our partial evaluator is on par with the narrowing-based partial evaluator [AHV02], and in most cases the specialized programs coincide up to renaming. This is especially noteworthy since the current implementation is considerably more complex compared to the simpler approach of narrowing, and achieved by the carefully chosen optimizations presented in Section 8.2. The speedup of the list- and tree-consuming functions is caused by the removal of intermediate data structures. For instance, `doubleFlip` defined as

```
data Tree         = Leaf Int | Node Int Tree Tree
flip (Leaf     n) = Leaf n
flip (Node n l r) = Tree n (flip r) (flip l)
main t            = PEVAL (flip (flip t))
```

has been specialized to the following definition without an intermediate tree:[2]

```
main  t            = flip2 t
flip2 (Leaf     n) = Leaf n
flip2 (Node n l r) = Node n (flip2 l) (flip2 r)
```

Regarding the second group of benchmarks, the speedup is achieved by the removal of interpretative overhead, since the programs are specialized with respect to some fixed input. For instance, the `kmp` pattern matcher [SGJ96], defined as

```
match p       s              = loop p s p s
loop []       _        _  _  = True
loop (_ : _ ) []       _  _  = False
loop (p : ps) (s : ss) op os = if p == s then loop ps ss op os else next op os
next _        []             = False
next op       (_ : ss)       = loop op ss op ss
```

[2]We present the results as Curry programs for readability.

is specialized for the fixed pattern `[A,A,B]` and the alphabet $\{$`A,B`$\}$. Its specialization is equivalent to the following Curry program:

```
matchAAB s       = loopAAB s
loopAAB []       = False
loopAAB (c : cs) = if c == A then loopAB cs else loopAAB cs
loopAB  []       = False
loopAB  (c : cs) = if c == A then loopB  cs else loopAAB (B : cs)
loopB   []       = False
loopB   (c : cs) = if c == B then True      else loopAAB (A : A : cs)
```

Note that this program is not optimal since the calls `loopAAB (B : cs)` and `loopAAB (A : A : cs)` could be replaced by `loopAAB cs` and `loopB cs`, respectively. This lack of efficiency is caused by three generalization steps performed to ensure global termination so that some recursive calls were evaluated with generalized patterns.

8.4.2 Higher-Order Programs

In the second group of benchmarks we consider some more realistic examples using higher-order functions.

- `sum` computes the sum of its input list and is defined as `sum xs = foldr (+) 0 xs`,
- `twiceSquare` applies the square function twice to each element of a list using `map`,
- `iterate f n = if n == 0 then f else iterate (f . f) (n - 1)` applies the function `f` to an argument 2^n times where we consider $n = 2$, and
- `deforest` is the introductory example `sum (map square (upto 1 n))` of Wadler's deforestation paper [Wad90].

The benchmarks are again applied to lists containing 500,000 elements, and the results are presented in Figure 8.3. In these cases, the speedup is generally achieved by the transformation of functions with higher-order definitions into first-order functions. For instance, the function `sum` is specialized to the following definition:

```
sum []       = 0
sum (x : xs) = x + sum xs
```

The possible improvements go beyond those achievable by a treatment of higher-order functions as "higher-order macros" [Wad90], i. e., the instantiation of higher-order functions with a specific argument function, which is demonstrated by `iterate`. Furthermore, they do not require specific formats like other transformations such as short-cut deforestation [GLP93], which demonstrates both the quality and applicability of the partial evaluator.

Note that the specialization for `twiceSquare` obtained from the new partial evaluator performs better than its counterpart obtained from the narrowing-based partial evaluator. This is caused by the consideration of sharing, since the original definition

Benchmark	PE Time	Original	NN-PE	Specialized	Speedup
`sum`	0.77	0.81	0.59	0.57	1.42
`twiceSquare`	0.83	2.19	2.02	1.77	1.24
`iterate`	1.20	4.02	2.54	3.25	1.24
`deforest`	34.84	3.45	4.06	2.61	1.32

Figure 8.3. Benchmark Results for Higher-Order Programs

```
square  x = x * x
twice f x = f (f x)
main   xs = PEVAL (map (twice square) xs)
```

is transformed to the specialization

```
main        xs       = twiceSquare xs
twiceSquare []       = []
twiceSquare (x : xs) = (let y = x * x in y * y) : twiceSquare xs
```

Due to the sharing of the multiplication `x * x`, the program performs only two multiplications for each element, while the narrowing-based specialization computes `(x * x) * (x * x)` and thus performs three multiplications. However, the effect of sharing does not always improve the efficiency of the specialization. For instance, during specialization of the program `iterate`, the argument `f` is shared and can thus not be inlined, which causes more complex expressions and thus a generalization step. For the narrowing-based partial evaluator, this generalization is not necessary.

The drastic increase in the partial evaluation time for the `deforest` benchmark is explained by the consideration of natural numbers in the homeomorphic embedding. In this case, the definition

```
upTo a b = if a > b then [] else a : upTo (a + 1) b
main   n = PEVAL (sum (mapSquare (upTo 1 n)))
```

leads to the evaluation of many expressions that are almost identical to the annotated expression, but where `1` is subsequently replaced by `2`, ..., `9`, until `10` finally embeds the initial number `1`. Consequently, the resulting program contains some intermediate function definitions, which are reflected in the following compressed specialization:

```
main     n = deforest n
deforest n = case 1 > n of
  True  -> 0
  False -> ... case 10 > n of
     True  -> 285
     False -> 1 + (4 + (... + (100 + deforest' 11 n) ... ))
deforest' a b = if a > b then 0 else (a * a) + deforest' (a + 1) b
```

For this example, the narrowing-based partial evaluator produces a much simpler program since it does not perform any additions.

Furthermore, the considerable partial evaluation time is also explained by the fact that the developed partial evaluator is a purely functional program compiled with the PAKCS system. Since the implementation does not employ any logic features, the PAKCS system is not the best choice to achieve fast execution. In fact, the partial evaluator compiled using KiCS2 only takes roughly 0.35 seconds for the specialization of `deforest`, and is thus about 100 times faster for this particular example. Nevertheless, we chose the PAKCS system for the benchmarks since it produces significant timings for smaller input sizes and thus avoids some additional problems such as memory space limitations. Furthermore, the KiCS2 system performs some additional optimizations, which may interfere with the optimizations obtained from partial evaluation and make the analysis of the partial evaluation results more difficult.

8.4.3 Non-Deterministic Programs

One of the main motivations for the development of the partial evaluator was the correct consideration of *non-deterministic* operations, which is examined by the next series of benchmarks.

- `choose` implements a non-deterministic choice of an arbitrary element for a given list by `choose xs = foldr (?) failed xs`,
- `headPerm` implements the same function in a more complicated fashion by taking the first element of a non-deterministic permutation of the input list,
- `last` computes the last element of a list and is defined by `last (_ ++ [x]) = x`,
- `some` is a variant of `choose` defined as `some (_ ++ (x : _)) = x`,
- `prefix` non-deterministically computes a prefix of the input list and is defined as `prefix (xs ++ _) = xs`, and
- `treemirror` mirrors a binary tree once, while the definition uses (trivial) smart constructors for functional pattern matching.

The results of these benchmarks are given in Figure 8.4, where the definitions are applied to lists of length 100,000 (`choose`), 10,000 (`headPerm`, `some`), 500,000 (`last`), 1,000 (`prefix`) and a balanced binary tree of depth 18 (`treemirror`). In the last four examples, we consider operations defined by means of linear functional patterns. Since the partial evaluator based on needed narrowing does not consider functional patterns, we exclude it from the comparison as it does not achieve any considerable effects.

The achieved speedups for the first group are still satisfactory, although no big improvements are made. Like for the benchmarks presented before, the main enhancements regard the elimination of intermediate structures and the transformation of

Benchmark	PE Time	Original	NN-PE	Specialized	Speedup
`choose`	0.82	0.32	0.27	0.27	1.19
`headPerm`	1.16	1.78	n/a[a]	1.81	0.98
`last`	0.88	2.30	-	0.06	38.33
`some`	0.94	8.64	-	0.03	288.00
`prefix`	1.79	0.58	-	0.53	1.09
`treemirror`	1.67	2.23	-	0.26	8.58

Figure 8.4. Benchmark Results for Non-Deterministic Programs

[a] The produced specialization has a different semantics.

higher-oder definitions to first-order ones. For instance, the definition of the operation `choose` is transformed to the more efficient variant

```
choose (x : xs)    = choose' x xs
choose' x _        = x
choose' _ (y : ys) = choose' y ys
```

The slight performance regression for benchmark `headPerm` is caused by a generalization step, and we will return to this example when we compare the different abstraction strategies. Furthermore, the narrowing-based partial evaluator fails to compute a correct specialization for this example, since it does not consider the effects of sharing.

The achieved speedups for the second group impressively demonstrate the usefulness of partial evaluation of (linear) functional patterns. For instance, the definition of `last` is transformed to the *deterministic* definition

```
last (x : xs)    = last' x xs
last' x []       = x
last' x (y : ys) = last' y ys
```

and the speedup is explained by the complete removal of non-determinism. The improvement for the benchmark `some` is caused by the removal of the intermediate list structure and lazy unifications, and the specialization is identical to the specialization of `choose` shown above. Compared to these two examples, the speedup achieved for the operation `prefix` is much smaller because during partial evaluation of the functional pattern

```
prefix (xs ++ _) = xs
```

the computed bindings for `xs` have to be considered in the right-hand side and cannot be discarded. Therefore, partial evaluation continues with bindings representing a list `xs` of increasing length, and thus a generalization step is necessary to ensure termination. For this example, further optimization techniques may be necessary to achieve better results. Finally, for the program `treemirror` defined as

Benchmark	One		Each		All	
	PE Time	Run Time	PE Time	Run Time	PE Time	Run Time
`power4`	1.03	0.87	1.02	0.88	0.76	0.19
`kmp`	4.42	0.21	2.61	0.14	1.61	0.09
`automaton`	5.36	0.94	3.73	0.96	1.56	0.19
`iterate`	1.20	3.25	1.24	3.14	0.92	2.77

Figure 8.5. Benchmark Results for Different Unfolding Strategies

```
data Tree = Leaf Int | Node Int Tree Tree

leaf x     = Leaf x
node n a b = Node n a b

mirror (leaf     n) = leaf n
mirror (node n a b) = node n (mirror b) (mirror a)
```

that mirrors a tree using smart constructors in functional patterns, the functional patterns are completely replaced by constructor patterns:

```
mirror (Leaf     n) = Leaf n
mirror (Node n a b) = Node n (mirror b) (mirror a)
```

While this example may seem artificial, it demonstrates the potential of efficient pattern matching on abstract data types that can be (de-)constructed by means of smart constructor functions. In consequence, this may enable an even more declarative style of programming using abstract data types.

8.4.4 Impact of Control Strategies

We finally like to investigate the impact of the different control strategies on the quality of the computed specialization. Regarding the *local* control strategy, Figure 8.5 contains an excerpt of the previous benchmarks where different unfolding strategies led to different specializations. For these examples, we show both the time necessary for partial evaluation (PE Time) as well as execution of the specialized program (Run Time) for abstraction based on the homeomorphic embedding and different unfolding strategies. The presented strategies either allow the unfolding of at most one function call in every derivation (One), one function call for every function (Each), or an unlimited unfolding (All).

While the run times do not show a significant difference between the strategies One and Each, the strategy All clearly outperforms the other ones. Although it also incorporates the risk of non-termination, it may still serve as a guideline for the quality of specializations achieved using different strategies. For instance, strategy

Benchmark	Hom. Embedding		Size-Based		None	
	PE Time	Run Time	PE Time	Run Time	PE Time	Run Time
`power4`	1.03	0.87	1.02	0.92	1.07	0.19
`kmp`	4.42	0.21	1.82	0.59	2.48	0.09
`automaton`	5.36	0.94	5.58	1.22	2.65	0.20
`iterate`	1.20	3.25	1.21	3.23	1.26	2.69
`headPerm`	1.16	1.81	1.11	1.81	0.87	0.03
`treemirror`	1.67	0.26	1.63	4.72	1.22	0.26

Figure 8.6. Benchmark Results for Different Abstraction Operators

All is capable to produce the following optimal specialization for `power4`

```
power4 x = let y = x * x in y * y
```

Furthermore, it produces an optimal specialization for `kmp`, so that the partial evaluator passes the KMP-test [SGJ96] and is thus capable to achieve results comparable to those obtained from positive supercompilation.

Regarding the impact of the *global* control strategy, Figure 8.6 shows both the partial evaluation time and run time for one-step unfolding and the different abstraction operators based on the homeomorphic embedding (Hom. Embedding), size-based abstraction (Size-Based), as well as no generalization (None). The figure again mentions only those benchmarks where the abstraction strategy had a significant influence. With the exception of `kmp`, the time needed for partial evaluation is on par for abstraction based on the homeomorphic embedding and sized-based abstraction, while the strategy employing the homeomorphic embedding consistently achieves better specializations. Like for the local control strategy, dropping the requirement for termination once again reveals the loss of efficiency caused by generalization steps. For instance, the specialization of `headPerm` without generalizations finally led to the same program as for `choose` and `some`. Although termination of the entire process is still essential in general, the strategy None may be thus useful for the improvement of other abstraction operators.

8.5 Summary

In this chapter we have presented a sophisticated partial evaluator for the functional logic language Curry, capable of achieving efficient specializations. The partial evaluator has been developed based on the general partial evaluation scheme for FlatCurry programs presented in Chapter 7, extended to cover additional language constructs such as higher-order functions, strict unification or linear functional patterns, and carefully optimized to increase the efficiency of the obtained specializations.

The partial evaluator shows promising results and usually performs on par or even better compared to the narrowing-based partial evaluator [AHV02], while preserving correctness also for non-confluent programs. Furthermore, it can achieve powerful optimizations such as deforestation [Wad90], the transformation of higher-order functions to first-order ones, or the transformation of non-deterministic operations to deterministic ones. Finally, the partial evaluator shows encouraging results for the optimization of linear functional patterns, which may encourage their broader application. In summary, the partial evaluator is able to significantly speedup the considered programs, and its easy integration into existing compilation chains even more advocates its broad application in the development of highly declarative and efficient functional logic Curry programs.

Chapter 9

Conclusion

It's more fun to arrive a conclusion than to justify it.

Malcolm Forbes

In this thesis we have subsequently developed a practical online partial evaluator for programs written in the functional logic language Curry. The implementation can be seen as an instance of the generic narrowing-based partial evaluation framework of Alpuente, Falaschi, and Vidal [AFV98], which was the first general framework for partial evaluation of functional logic programming languages. Early instances of this framework considered the partial evaluation of programs based on innermost narrowing [AFV98], lazy narrowing [AFI+97], and needed narrowing [ALH+05]. However, these instances did not directly lead to a practical implementation of a partial evaluator for realistic functional logic languages such as Curry or $\mathcal{TOY}$, since various language constructs have not been considered. To address this shortcoming, the framework has later been extended to support an abstract program representation [AHV00] together with a non-standard residualizing semantics for partial evaluation [AHV03]. These ideas finally led to the development of the partial evaluator of Albert, Hanus, and Vidal [AHV02], which was the first practical implementation of a partial evaluator for the language Curry. Nevertheless, at that time Curry programs were considered to be confluent, and the later integration of implicit non-deterministic operations limited the applicability of the partial evaluator since correctness was only achieved for confluent programs.

To allow the correct partial evaluation of non-deterministic operations, the implications of sharing have to be considered, which does not fit well into the basic framework of narrowing. Furthermore, the advent of language concepts such as (mutually recursive) `let` expressions requires an extension of the general partial evaluation framework. These shortcomings motivated to the development of the partial evaluator described in this thesis, which is capable of improving realistic Curry programs and supports contemporary language features such as mutually recursive bindings, non-deterministic operations, and linear functional patterns. Furthermore, the presented benchmarks document its potential to achieve powerful optimizations, which perform on par or even better than those obtained from partial evaluation based on needed narrowing.

9.1 Contributions

The development of the partial evaluator has been carried out in several subsequent steps, ranging from the normalization of Curry programs to the optimizations necessary to obtain reasonably efficient specializations. Each of these steps constitutes an individual contribution of this thesis, and we discuss them in the following.

9.1.1 Normalization of Functional Logic Languages

Current functional logic programming languages such as Curry provide a rich set of syntactic constructs allowing a high-level, declarative style of programming. During compilation, the majority of syntactic constructs is transformed into simpler constructs by a number of transformation steps. In the official language report, these transformation steps are presented in isolation, i. e., they do not take into account the effects of other transformation steps. While this eases their discussion and understanding, some language constructs such as non-linear and functional patterns interfere with each other, so that their combination has to be considered as well. We therefore presented an *comprehensive normalization scheme* for the normalization of Curry programs, specifying the order of independent transformation steps. Furthermore, we identified illegal combinations of language features and discussed subtle issues from the combination of certain constructs. Finally, the implementation of both flexible and rigid pattern matching has been formalized by the corresponding transformations.

This normalization scheme can therefore be seen as a reference for the transformations employed in a compiler, and may be useful to specify the integration of new language features in the future. A possible topic for future work is the compilation of rigid pattern matching in Curry, since there exist several suitable algorithms that produce different results of varying code size and efficiency. It may therefore be worthwhile to further investigate and assess those algorithms, and to identify the one best suitable for integration into contemporary Curry systems.

9.1.2 Generalized Operational Semantics and Extensions

Starting with the formal representation of FlatCurry programs, we developed a modified variant of its operational semantics [AHH+05] that eliminates the lack of blackhole detection, based on the ideas of Braßel [Bra11]. Afterwards, we generalized this variant to be applicable to non-flattened expressions, which has been shown to allow the same set of statements (modulo flattening) to be derived, and furthermore leads to the same abstract semantics of expressions. Afterwards, we extended the semantics by additional rules covering primitive operations in general, and in particular the primitives necessary to support higher-order application, strict unification

and linear functional patterns. In consequence, the resulting operational semantics serves as a formal specification for the evaluation of FlatCurry programs.

Nevertheless, although the presented rules for functional pattern unification resemble the algorithm proposed by Antoy and Hanus [AH05], they are limited to *linear* functional patterns. Non-linear functional pattern in addition require the generation of strict unification constraints on demand, which requires an additional bookkeeping of bound variables. Consequently, the formalization of the operational semantics of non-linear functional patterns is still an interesting topic for future work.

9.1.3 Residualizing Semantics and Partial Evaluation Scheme

Based on the extended operational semantics, we developed a residualizing semantics for partial evaluation capable to deal with additional requirements such as ensured termination or partial information in the form of unbound variables. We then extended the partial evaluation framework of Alpuente, Falaschi, and Vidal [AFV98] to deal with FlatCurry programs instead of term-rewriting systems. To start with, we refined the notions of resultants and pre-partial evaluations to deal with the implications of the residualizing semantics by providing mechanisms to convert configurations to expressions and multiple resultants to a single pre-partial evaluation. Furthermore, the notions of recursive renaming and abstraction were adjusted to cover the additional syntactic construct as well as consider the implications of sharing. In addition, we presented an extended homeomorphic embedding relation establishing a well-founded quasi order on the set of FlatCurry expressions. By providing several theoretical results, we proved the correctness of the residualizing semantics and the constructed partial evaluations, as well as the termination and correctness of the partial evaluation algorithm.

9.1.4 Practical Partial Evaluator

Based on the proposed partial evaluation scheme, we continued to develop a practical partial evaluator for realistic Curry programs. As the first step, we extended the residualizing semantics to cover primitive operations in general, and in particular higher-order application, strict unification and linear functional patterns. While this extension addressed the applicability of the partial evaluator, several optimizations were furthermore necessary to also obtain specialization of an satisfactory efficiency, which has been enabled by a number of optimization steps. Afterwards, the implementation and usage of the partial evaluator has been sketched, in conjunction with its parametricity with respect to the employed unfolding rule and abstraction operator. The general usefulness has then been documented by a number of selected benchmarks addressing different features of the language Curry.

The developed partial evaluator shows impressive results in general, and in particular achieves powerful transformations such as deforestation [Wad90]. Furthermore,

it is capable of specializing a pattern matching algorithm using a fixed pattern to a result comparable to the result obtained by the Knuth-Morris-Pratt pattern matching algorithm [KMP77]. This KMP-test is usually applied to assess the power of program transformations, and only regularly passed for positive supercompilers but not for partial evaluators. Thus, this result documents the general power of partial evaluation for functional logic languages. In addition, the presented partial evaluator achieves useful transformations such as the transformation of higher-order functions into first-order ones beyond simple macro instantiation, and the transformation of certain non-deterministic operations into deterministic ones.

Furthermore, the partial evaluator may also be used as a limited form of a theorem prover, since in certain cases it is able to show the equivalence of expressions by partial evaluation into the same specialization, an idea that has also been applied in the context of supercompilation [KR10].

9.2 Future Work

Although the partial evaluator developed in this thesis is fully applicable for the partial evaluation of Curry programs and already provides impressive results, there are still several topics worth of further investigation.

9.2.1 Full Support for Functional Patterns

The current implementation is restricted to the support of *linear functional patterns*, i. e., functional patterns that evaluate to linear values. The specializations obtained from partial evaluation of linear functional patterns are promising, and the support of *non-linear functional patterns* is therefore desirable as well. This requires an extension of the residualizing semantics to also cover non-linear functional patterns, which could be easily derived from the corresponding operational semantics.

However, the formalization of an operational semantics of the functional pattern unification operator "`=:<=`" is still an open topic, and recent investigations revealed that both the implementations incorporated into the PAKCS and KiCS2 systems lead to wrong results for certain constellations. Therefore, such a formalization is both desirable for the validation of the corresponding implementation in current Curry systems, as well as for extending the applicability of the partial evaluator. Nevertheless, especially in the context of partial evaluation the formalization is a challenging task, due to the following requirements it must satisfy:

- During evaluation of an expression e_1 `=:<=` e_2, all logic variables occurring repeatedly in a value of e_1 must be strictly unified. However, since the value should be incrementally computed, the already encountered variables need to be memorized, and only if a variable is repeatedly bound by a call of "`=:<=`" originating from the same functional pattern, a strict unification must be initiated.

▷ Due to the risk of non-termination, partial evaluation of a functional pattern might be stopped at any time and, thus, any intermediate structure necessary for the correct implementation of the functional pattern unification must be representable as an expression to fit into the partial evaluation scheme.

Until an appropriate formalization has been discovered, it might be reasonable to incorporate a static approximation to classify the functional patterns to either be linear or possibly non-linear. Such an approximation is already incorporated in the PAKCS system, and it should be easily transferable to our context.

9.2.2 Improvements in the Partial Evaluation Process

The support for non-linear expressions as well as locally bound variables introduced by `let` or `case` expressions required special considerations in the partial evaluation framework. For instance, non-linear expressions required the adaptation of the most-specific generalization, so that the generalization either must be linear or variables that repeatedly occur in the generalization must be substituted with constructor expressions. This requirement may be further relaxed to only require linearity for *possibly non-deterministic* substitutions, such that expressions that are known to be deterministic could be duplicated. For this purpose, an additional non-determinism analysis becomes necessary, which classifies an arbitrary expression as either known deterministic or potentially non-deterministic [BH05]. A simple variant of such an analysis can be implemented as an abstract interpretation [CC77] of FlatCurry programs, and there already exists a general framework for such kinds of program analysis [HS14]. In consequence, such an analysis might be easy to incorporate.

Another complication in the generalization of expressions arises from the existence of locally bound variables which must not occur in the range of a substitution. In consequence, certain comparable expressions can only be generalized to a trivial generalization. A possible solution might be to consider *higher-order* generalizations, i. e., generalizations that take an additional higher order argument such that diverging subexpressions containing locally bound variables can be extracted as the function body of the higher-order argument. However, it is not clear whether the more accurate generalization outweighs the possible overhead of higher-order functions. Furthermore, the invention of new function declarations may impose new challenges for termination of the partial evaluation process.

A further topic for future work is the development of more sophisticated termination criteria both for unfolding and abstraction. The unfolding strategy has a notably influence on the number of pre-partial evaluations, and a more generous strategy may avoid some pre-partial evaluations, although at the same time it may reduce the chances for the identification of variants which results in less polygenetic residual functions. The abstraction strategy in contrast has a notable influence on the quality of specializations, since every generalization step leads to some loss of

information. The work of Bolingbroke and Peyton Jones [BP10] reports good results for a termination criterion on "tag-bags" [Mit10], and it may be worthwhile to add this criterion as an additional strategy.

9.2.3 Integration of Algebraic Optimizations

Further improvements of the efficiency of the specializations may be achieved by the integration of optimization steps based on algebraic laws, which has already been mentioned as a transformation step in the early fold/unfold framework [BD77]. As an example, we consider the following inefficient definition of `scanl`, which behaves similar to `foldl` but produces a list of the successively computed values:

```
scanl :: (b -> a -> b) -> b -> [a] -> [b]
scanl f e xs = PEVAL (map (foldl f e) (inits xs))

foldl :: (a -> b -> a) -> a -> [b] -> a
foldl _ z []       = z
foldl f z (x : xs) = foldl f (f z x) xs

inits :: [a] -> [[a]]
inits []       = [] : []
inits (x : xs) = [] : map (x :) (inits xs)
```

For instance, `scanl (+) 0 [1, 2, 3]` evaluates to the list of successively summarized numbers `[0, 1, 3, 6]`. If we partially evaluate the above definition of `scanl`, we only obtain an inefficient specialization. However, if we apply the two algebraic laws

$$\texttt{map f (map g) xs} \equiv \texttt{map (f . g) xs}$$

for the fusion of subsequent applications of `map` and the law

$$\texttt{(foldl f e . (x :))} \equiv \texttt{foldl f (f e x)} \quad \text{for } f \in \mathcal{V}\,,$$

then partial evaluation would be able to produce the efficient specialization

```
scanl f e xs = fcase xs of
  []       -> e : []
  (y : ys) -> e : scanl f (f e y) ys
```

Experiments show that evaluation of `scanl (+) 0 [1 .. 10000]` only take 0.01 seconds for this efficient specialization, instead of 1.52 seconds for the previously obtained specialization. In consequence, the application of algebraic laws may be another instrument to achieve more efficient specializations. Note that the two laws stated above can easily be shown correct by an application of the partial evaluator itself, since for both equations the left- and right-hand side lead to the same specialization.

However, although the application of an algebraic law is a rather simple transformation, it requires the identification of valid equivalences between expressions,

which is a non-trivial task. While some simple equivalences, like the fusion of subsequent applications of `map`, are well-known and could be manually incorporated, an automatic approach is still desirable. For this purpose, one could consider the integration of *free theorems* [Wad89], which allow the derivation of statements over certain functions based on their type signature. The idea of using free theorems is not new, and Kehler Holst and Hughes [KH91] apply free theorems to improve the binding-time analysis for offline partial evaluation. Furthermore, the research field of free theorems for functional logic languages has just recently gained more attention. While the earlier work of Christiansen, Seidel, and Voigtländer [CSV10] investigates free theorems for Curry by a case study of selected examples to provide a first intuition, the more recent work of Mehner, Seidel, Straßburger, and Voigtländer [MSS+14] presents a formal foundation to derive free theorems for a limited subset of Curry, and we are confident that this work could be helpful to improve the partial evaluation of Curry programs.

9.2.4 Formal Investigation of Run Time Costs

In our evaluation, we have investigated the effects of partial evaluation by comparing the run times of the original and the specialized program for some fixed input. While this approach is simple and provides a first impression of the achieved speedup, the speedup cannot be contributed to certain transformations employed during partial evaluation or the later optimization of the program. Furthermore, other effects such as the memory consumption of the program were also not taken into account.

It is therefore desirable to develop a formal cost model for the evaluation of functional logic programs which allows a formal reasoning about the effects of certain transformations. Such a cost model has already been proposed for functional logic languages based on narrowing [AAV00] and should in principle be transferable to our context. This model would then allow the detailed investigation of both simple transformation steps as well as the attribution of an exact cost reduction for specific examples, so that the variation in the results obtained by measurement of the execution time can be avoided. In certain cases, it may then be possible to determine the speedup achieved by some pre-partial evaluations such that worthwhile evaluations can be kept and the general speedup of the partial evaluation can be estimated.

9.2.5 Further Automation

Another interesting topic worth of further research regards the automation of the partial evaluation process. Currently, the expressions to be partially evaluated have to be specified by the user via manual annotations. While this approach is flexible and allows the user to experiment with the usefulness of several partial evaluations, it also requires an accurate identification of expressions to obtain a satisfactory result.

It may thus be desirable to provide some heuristics for the automatic identification of good candidates for partial evaluation, leading to a fully automated partial evaluation process that could be incorporated into existing compilation chains. The provided benchmarks indicate that the specialization of higher-order functions to first-order ones and the specialization of functions defined using linear functional patterns seems to be advantageous, but more experiments are still necessary.

Appendix A

Syntax of Curry

The emotional intensity of debate on a language feature increases as one moves down the following scale: Semantics, Syntax, Lexical syntax, Comments.

Philip Wadler

We present the context-free grammar of the Curry language as accepted by the Curry systems KiCS2 and PAKCS, which is a slightly extended version of the grammar specified in the Curry Report [Han12]. In addition, a minor difference concerns the notation of numeric and character literals, where we adopt the notation of the language Haskell [Mar10].

A.1 Notational Conventions

The syntax is given in extended Backus-Naur-Form using the following notation:

$\mathit{NonTerm} ::= \alpha$	production
$\mathit{NonTerm}$	nonterminal symbol
`Term`	terminal symbol
$[\alpha]$	optional
$\{\alpha\}$	zero or more repetitions
(α)	grouping
$\alpha \mid \beta$	alternative
$\alpha_{\langle\beta\rangle}$	difference – elements generated by α without those generated by β

A.2 Lexicon

Comments

Comments either begin with "`--`" and terminate at the end of the line, or begin with "`{-`" and terminate with a matching "`-}`", i. e., the delimiters "`{-`" and "`-}`" act as parentheses and can be nested.

Identifiers and Keywords

The case of identifiers is important, i. e., the identifier "abc" is different from "ABC". Although the Curry Report specifies four different case modes (Prolog, Gödel, Haskell, free), the KiCS2 and PAKCS systems only support the *free* mode which puts no constraints on the case of identifiers in certain language constructs.

$$
\begin{array}{rcl}
\mathit{Letter} & ::= & \textit{any ASCII letter} \\
\mathit{Dashes} & ::= & \texttt{--}\ \{\texttt{-}\} \\
 & & \\
\mathit{Ident} & ::= & (\mathit{Letter}\ \{\mathit{Letter} \mid \mathit{Digit} \mid \_ \mid \texttt{'}\})_{\langle \mathit{ReservedID} \rangle} \\
\mathit{Symbol} & ::= & \texttt{~} \mid \texttt{!} \mid \texttt{@} \mid \texttt{\#} \mid \texttt{\$} \mid \texttt{\%} \mid \texttt{\^{}} \mid \texttt{\&} \mid \texttt{*} \mid \texttt{+} \mid \texttt{-} \mid \texttt{=} \mid \texttt{<} \mid \texttt{>} \mid \texttt{?} \mid \texttt{.} \mid \texttt{/} \mid \texttt{|} \mid \texttt{\textbackslash} \mid \texttt{:} \\
 & & \\
\mathit{ModuleID} & ::= & \{\mathit{Ident}\ \texttt{.}\}\ \mathit{Ident} \\
\mathit{TypeConstrID} & ::= & \mathit{Ident} \\
\mathit{TypeVarID} & ::= & \mathit{Ident} \mid \_ \\
\mathit{DataConstrID} & ::= & \mathit{Ident} \\
\mathit{InfixOpID} & ::= & (\mathit{Symbol}\ \{\mathit{Symbol}\})_{\langle \mathit{Dashes} \mid \mathit{ReservedSym} \rangle} \\
\mathit{FunctionID} & ::= & \mathit{Ident} \\
\mathit{VariableID} & ::= & \mathit{Ident} \\
\mathit{LabelID} & ::= & \mathit{Ident} \\
 & & \\
\mathit{QTypeConstrID} & ::= & [\mathit{ModuleID}\ \texttt{.}]\ \mathit{TypeConstrID} \\
\mathit{QDataConstrID} & ::= & [\mathit{ModuleID}\ \texttt{.}]\ \mathit{DataConstrID} \\
\mathit{QInfixOpID} & ::= & [\mathit{ModuleID}\ \texttt{.}]\ \mathit{InfixOpID} \\
\mathit{QFunctionID} & ::= & [\mathit{ModuleID}\ \texttt{.}]\ \mathit{FunctionID} \\
\mathit{QLabelID} & ::= & [\mathit{ModuleID}\ \texttt{.}]\ \mathit{LabelID}
\end{array}
$$

The following identifiers are recognized as keywords and cannot be used as regular identifiers:

$$
\begin{array}{rcl}
\mathit{ReservedID} & ::= & \texttt{case} \mid \texttt{data} \mid \texttt{do} \mid \texttt{else} \mid \texttt{external} \mid \texttt{fcase} \mid \texttt{foreign} \\
 & \mid & \texttt{free} \mid \texttt{if} \mid \texttt{import} \mid \texttt{in} \mid \texttt{infix} \mid \texttt{infixl} \mid \texttt{infixr} \\
 & \mid & \texttt{let} \mid \texttt{module} \mid \texttt{newtype} \mid \texttt{of} \mid \texttt{then} \mid \texttt{type} \mid \texttt{where}
\end{array}
$$

Note that the identifiers `as`, `hiding` and `qualified` are no keywords. They have only a special meaning in module headers and can thus be used as ordinary identifiers elsewhere. The following symbols also have a special meaning and cannot be used as an infix operator identifier:

$$
\mathit{ReservedSym} ::= \texttt{..} \mid \texttt{:} \mid \texttt{::} \mid \texttt{=} \mid \texttt{\textbackslash} \mid \texttt{|} \mid \texttt{<-} \mid \texttt{->} \mid \texttt{@} \mid \texttt{~}
$$

Numeric and Character Literals

In contrast to the Curry Report, we adopt Haskell's notation of literals for both numeric as well as character and string literals, extended with the ability to denote binary integer literals.

```
Int ::= Decimal
      | 0b Binary | 0B Binary
      | 0o Octal | 0O Octal
      | 0x Hexadecimal | 0X Hexadecimal

Float ::= Decimal . Decimal [Exponent]
        | Decimal Exponent
Exponent ::= (e | E) [+ | -] Decimal

Decimal ::= Digit {Digit}
Binary ::= Binit {Binit}
Octal ::= Octit {Octit}
Hexadecimal ::= Hexit {Hexit}

Digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
Binit ::= 0 | 1
Octit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7
Hexit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F | a | b | c | d | e | f
```

For character and string literals, the syntax is as follows:

```
Char ::= ' ( Graphic_⟨\⟩ | Space | Escape_⟨\&⟩ ) '
String ::= " { Graphic_⟨" | \⟩ | Space | Escape | Gap } "
Escape ::= \ ( CharEsc | AsciiEsc | Decimal | o Octal | x Hexadecimal )
CharEsc ::= a | b | f | n | r | t | v | \ | " | ' | &
AsciiAsc ::= ^ Cntrl | NUL | SOH | STX | ETX | EOT | ENQ | ACK
           | BEL | BS | HT | LF | VT | FF | CR | SO | SI | DLE
           | DC1 | DC2 | DC3 | DC4 | NAK | SYN | ETB | CAN
           | EM | SUB | ESC | FS | GS | RS | US | SP | DEL
Cntrl ::= A | ... | Z | @ | [ | \ | ] | ^ | _
Gap ::= \ WhiteChar {WhiteChar} \
Graphic ::= any graphical character
WhiteChar ::= any whitespace character
```

A.3 Layout

Like in Haskell, a Curry programmer can use layout information to define the structure of blocks. For this purpose, we define the indentation of a symbol as the column number indicating the start of this symbol, and the indentation of a line is the indentation of its first symbol.[1]

The layout (or "off-side") rule applies to lists of syntactic entities after the keywords `let`, `where`, `do`, or `of`. In the subsequent context-free syntax, these lists are enclosed with curly braces (`{ }`) and the single entities are separated by semicolons (`;`). Instead of

[1] In order to determine the exact column number, we assume a fixed-width font with tab stops at each 8th column.

using the curly braces and semicolons of the context-free syntax, a Curry programmer can also specify these lists by indentation: the indentation of a list of syntactic entities after `let`, `where`, `do`, or `of` is the indentation of the next symbol following the `let`, `where`, `do`, `of`. Any item of this list starts with the same indentation as the list. Lines with only whitespaces or an indentation greater than the indentation of the list continue the item in the previous line. Lines with an indentation less than the indentation of the list terminate the entire list. Moreover, a list started by `let` is terminated by the keyword `in`. Thus, the sentence

```
f x = h x where { g y = y + 1 ; h z = (g z) * 2 }
```

which is valid w.r.t. the context-free syntax, can be written with the layout rules as

```
f x = h x
 where g y = y + 1
       h z = (g z) * 2
```

or also as

```
f x = h x  where
  g y = y + 1
  h z = (g z)
          * 2
```

To avoid an indentation of top-level declarations, the keyword `module` and the end-of-file token are assumed to start in column 0.

A.4 Context-Free Grammar

$$
\begin{array}{rcll}
\mathit{Module} & ::= & \texttt{module}\ \mathit{ModuleID}\ [\mathit{Exports}]\ \texttt{where}\ \mathit{Block} & \\
 & \mid & \mathit{Block} & \\
 & & & \\
\mathit{Block} & ::= & \texttt{\{}\ [\mathit{ImportDecls}\ \texttt{;}]\ \mathit{BlockDecl}_1\ \texttt{;}\ \ldots\ \texttt{;}\ \mathit{BlockDecl}_n\ \texttt{\}} & (n \geqslant 0) \\
 & & & \\
\mathit{Exports} & ::= & \texttt{(}\ \mathit{Export}_1\ \texttt{,}\ \ldots\ \texttt{,}\ \mathit{Export}_n\ \texttt{)} & (n \geqslant 0) \\
\mathit{Export} & ::= & \mathit{QFunction} & \\
 & \mid & \mathit{QTypeConstrID}\ [\texttt{(}\ \mathit{ConsLabel}_1\ \texttt{,}\ \ldots\ \texttt{,}\ \mathit{ConsLabel}_n\ \texttt{)}] & (n \geqslant 0) \\
 & \mid & \mathit{QTypeConstrID}\ \texttt{(..)} & \\
 & \mid & \texttt{module}\ \mathit{ModuleID} & \\
\mathit{ConsLabel} & ::= & \mathit{DataConstr} \mid \mathit{Label} & \\
 & & & \\
\mathit{ImportDecls} & ::= & \mathit{ImportDecl}_1\ \texttt{;}\ \ldots\ \texttt{;}\ \mathit{ImportDecl}_n & (n \geqslant 1) \\
\mathit{ImportDecl} & ::= & \texttt{import}\ [\texttt{qualified}]\ \mathit{ModuleID}\ [\texttt{as}\ \mathit{ModuleID}]\ [\mathit{ImportSpec}] & \\
\mathit{ImportSpec} & ::= & \texttt{(}\ \mathit{Import}_1\ \texttt{,}\ \ldots\ \texttt{,}\ \mathit{Import}_n\ \texttt{)} & (n \geqslant 0) \\
 & \mid & \texttt{hiding}\ \texttt{(}\ \mathit{Import}_1\ \texttt{,}\ \ldots\ \texttt{,}\ \mathit{Import}_n\ \texttt{)} & (n \geqslant 0) \\
\mathit{Import} & ::= & \mathit{Function} & \\
 & \mid & \mathit{TypeConstrID}\ [\texttt{(}\ \mathit{ConsLabel}_1\ \texttt{,}\ \ldots\ \texttt{,}\ \mathit{ConsLabel}_n\ \texttt{)}] & (n \geqslant 0) \\
 & \mid & \mathit{TypeConstrID}\ \texttt{(..)} &
\end{array}
$$

$$
\begin{array}{rcll}
\mathit{BlockDecl} & ::= & \mathit{TypeSynDecl} & \\
& | & \mathit{DataDecl} & \\
& | & \mathit{NewtypeDecl} & \\
& | & \mathit{FixityDecl} & \\
& | & \mathit{FunctionDecl} & \\
\\
\mathit{TypeSynDecl} & ::= & \texttt{type}\ \mathit{SimpleType}\ \texttt{=}\ \mathit{TypeExpr} & \\
\mathit{SimpleType} & ::= & \mathit{TypeConstrID}\ \mathit{TypeVarID}_1\ \ldots\ \mathit{TypeVarID}_n & (n \geqslant 0) \\
\\
\mathit{DataDecl} & ::= & \texttt{data}\ \mathit{SimpleType} & (\textit{external data type}) \\
& | & \texttt{data}\ \mathit{SimpleType}\ \texttt{=}\ \mathit{ConstrDecl}_1\ \texttt{|}\ \ldots\ \texttt{|}\ \mathit{ConstrDecl}_n & (n \geqslant 1) \\
\mathit{ConstrDecl} & ::= & \mathit{DataConstr}\ \mathit{SimpleTypeExpr}_1\ \ldots\ \mathit{SimpleTypeExpr}_n & (n \geqslant 0) \\
& | & \mathit{TypeConsExpr}\ \mathit{ConOp}\ \mathit{TypeConsExpr} & (\textit{infix data constructor}) \\
& | & \mathit{DataConstr}\ \texttt{\{}\ \mathit{FieldDecl}_1\ \texttt{,}\ \ldots\ \texttt{,}\ \mathit{FieldDecl}_n\ \texttt{\}} & (n \geqslant 0) \\
\mathit{FieldDecl} & ::= & \mathit{Label}_1\ \texttt{,}\ \ldots\ \texttt{,}\ \mathit{Label}_n\ \texttt{::}\ \mathit{TypeExpr} & (n \geqslant 1) \\
\\
\mathit{NewtypeDecl} & ::= & \texttt{newtype}\ \mathit{SimpleType}\ \texttt{=}\ \mathit{NewConstrDecl} & \\
\mathit{NewConstrDecl} & ::= & \mathit{DataConstr}\ \mathit{SimpleTypeExpr} & \\
& | & \mathit{DataConstr}\ \texttt{\{}\ \mathit{Label}\ \texttt{::}\ \mathit{TypeExpr}\ \texttt{\}} & \\
\\
\mathit{TypeExpr} & ::= & \mathit{TypeConsExpr}\ [\texttt{->}\ \mathit{TypeExpr}] & \\
\mathit{TypeConsExpr} & ::= & \mathit{QTypeConstrID}\ \mathit{SimpleTypeExpr}_1\ \ldots\ \mathit{SimpleTypeExpr}_n & (n \geqslant 1) \\
& | & \mathit{SimpleTypeExpr} & \\
\mathit{SimpleTypeExpr} & ::= & \mathit{TypeVarID} & \\
& | & \mathit{QTypeConstrID} & \\
& | & \texttt{()} & (\textit{unit type}) \\
& | & \texttt{(}\ \mathit{TypeExpr}_1\ \texttt{,}\ \ldots\ \texttt{,}\ \mathit{TypeExpr}_n\ \texttt{)} & (\textit{tuple type},\ n \geqslant 2) \\
& | & \texttt{[}\ \mathit{TypeExpr}\ \texttt{]} & (\textit{list type}) \\
& | & \texttt{(}\ \mathit{TypeExpr}\ \texttt{)} & (\textit{parenthesized type}) \\
\\
\mathit{FixityDecl} & ::= & \mathit{Fixity}\ [\mathit{Int}]\ \mathit{Op}_1\ \texttt{,}\ \ldots\ \texttt{,}\ \mathit{Op}_n & (n \geqslant 1) \\
\mathit{Fixity} & ::= & \texttt{infixl} \mid \texttt{infixr} \mid \texttt{infix} & \\
\\
\mathit{FunctionDecl} & ::= & \mathit{Signature} \mid \mathit{ExternalDecl} \mid \mathit{Equation} & \\
\mathit{Signature} & ::= & \mathit{Functions}\ \texttt{::}\ \mathit{TypeExpr} & \\
\mathit{ExternalDecl} & ::= & \mathit{Functions}\ \texttt{external} & (\textit{externally defined functions}) \\
\mathit{Functions} & ::= & \mathit{Function}_1\ \texttt{,}\ \ldots\ \texttt{,}\ \mathit{Function}_n & (n \geqslant 1) \\
\mathit{Equation} & ::= & \mathit{FunLhs}\ \mathit{Rhs} & \\
\mathit{FunLhs} & ::= & \mathit{Function}\ \mathit{SimplePat}_1\ \ldots\ \mathit{SimplePat}_n & (n \geqslant 0) \\
& | & \mathit{ConsPattern}\ \mathit{FunOp}\ \mathit{ConsPattern} & \\
& | & \texttt{(}\ \mathit{FunLhs}\ \texttt{)}\ \mathit{SimplePat}_1\ \ldots\ \mathit{SimplePat}_n & (n \geqslant 1) \\
\\
\mathit{Rhs} & ::= & \texttt{=}\ \mathit{Expr}\ [\texttt{where}\ \mathit{LocalDecls}] & \\
& | & \mathit{CondExprs}\ [\texttt{where}\ \mathit{LocalDecls}] & \\
\mathit{CondExprs} & ::= & \texttt{|}\ \mathit{InfixExpr}\ \texttt{=}\ \mathit{Expr}\ [\mathit{CondExprs}] & \\
\\
\mathit{LocalDecls} & ::= & \texttt{\{}\ \mathit{LocalDecl}_1\ \texttt{;}\ \ldots\ \texttt{;}\ \mathit{LocalDecl}_n\ \texttt{\}} & (n \geqslant 0) \\
\mathit{LocalDecl} & ::= & \mathit{FunctionDecl} & \\
& | & \mathit{PatternDecl} &
\end{array}
$$

	|	*Variable*$_1$, ... , *Variable*$_n$ `free`	($n \geqslant 1$)
	|	*FixityDecl*	
PatternDecl	::=	*Pattern Rhs*	
Pattern	::=	*ConsPattern* [*QConOp Pattern*]	(*infix constructor pattern*)
ConsPattern	::=	*GDataConstr SimplePat*$_1$ ... *SimplePat*$_n$	(*constructor pattern*, $n \geqslant 1$)
	|	`-` *Int*	(*negative integer pattern*)
	|	`-.` *Float*	(*negative float pattern*)
	|	*SimplePat*	
SimplePat	::=	*Variable*	(*variable*)
	|	_	(*wildcard*)
	|	*GDataConstr*	(*constructor*)
	|	*Literal*	(*literal*)
	|	`(` *Pattern* `)`	(*parenthesized pattern*)
	|	`(` *Pattern*$_1$, ... , *Pattern*$_n$ `)`	(*tuple pattern*, $n \geqslant 2$)
	|	`[` *Pattern*$_1$, ... , *Pattern*$_n$ `]`	(*list pattern*, $n \geqslant 1$)
	|	*Variable* `@` *SimplePat*	(*as-pattern*)
	|	`~` *SimplePat*	(*irrefutable pattern*)
	|	`(` *QFunction SimplePat*$_1$ ... *SimplePat*$_n$ `)`	(*functional pattern*, $n \geqslant 1$)
	|	`(` *ConsPattern QFunOp Pattern* `)`	(*infix functional pattern*)
	|	*QDataConstr* `{` *FieldPat*$_1$, ... , *FieldPat*$_n$ `}`	(*labeled pattern*, $n \geqslant 0$)
FieldPat	::=	*QLabel* `=` *Pattern*	
Expr	::=	*InfixExpr* `::` *TypeExpr*	(*expression with type signature*)
	|	*InfixExpr*	
InfixExpr	::=	*NoOpExpr QOp InfixExpr*	(*infix operator application*)
	|	`-` *InfixExpr*	(*unary int minus*)
	|	`-.` *InfixExpr*	(*unary float minus*)
	|	*NoOpExpr*	
NoOpExpr	::=	`\` *SimplePat*$_1$ ... *SimplePat*$_n$ `->` *Expr*	(*lambda expression*, $n \geqslant 1$)
	|	`let` *LocalDecls* `in` *Expr*	(*let expression*)
	|	`if` *Expr* `then` *Expr* `else` *Expr*	(*conditional*)
	|	`case` *Expr* `of` `{` *Alt*$_1$ `;` ... `;` *Alt*$_n$ `}`	(*case expression*, $n \geqslant 1$)
	|	`fcase` *Expr* `of` `{` *Alt*$_1$ `;` ... `;` *Alt*$_n$ `}`	(*fcase expression*, $n \geqslant 1$)
	|	`do` `{` *Stmt*$_1$ `;` ... `;` *Stmt*$_n$ `;` *Expr* `}`	(*do expression*, $n \geqslant 0$)
	|	*FuncExpr*	
FuncExpr	::=	[*FuncExpr*] *BasicExpr*	(*application*)
BasicExpr	::=	*Variable*	(*variable*)
	|	_	(*anonymous free variable*)
	|	*QFunction*	(*qualified function*)
	|	*GDataConstr*	(*general constructor*)
	|	*Literal*	(*literal*)
	|	`(` *Expr* `)`	(*parenthesized expression*)
	|	`(` *Expr*$_1$, ... , *Expr*$_n$ `)`	(*tuple*, $n \geqslant 2$)
	|	`[` *Expr*$_1$, ... , *Expr*$_n$ `]`	(*finite list*, $n \geqslant 1$)
	|	`[` *Expr* [`,` *Expr*] `..` [*Expr*] `]`	(*arithmetic sequence*)

$$
\begin{array}{rcll}
 & | & \texttt{[}\ \mathit{Expr}\ \texttt{|}\ \mathit{Qual}_1\ \texttt{,}\ \ldots\ \texttt{,}\ \mathit{Qual}_n\ \texttt{]} & (\textit{list comprehension},\ n \geqslant 1) \\
 & | & \texttt{(}\ \mathit{InfixExpr}\ \mathit{QOp}\ \texttt{)} & (\textit{left section}) \\
 & | & \texttt{(}\ \mathit{QOp}_{\langle \texttt{-}, \texttt{-}. \rangle}\ \mathit{InfixExpr}\ \texttt{)} & (\textit{right section}) \\
 & | & \mathit{QDataConstr}\ \texttt{\{}\ \mathit{FBind}_1\ \texttt{,}\ \ldots\ \texttt{,}\ \mathit{FBind}_n\ \texttt{\}} & (\textit{record construction},\ n \geqslant 0) \\
 & | & \mathit{BasicExpr}_{\langle \mathit{QDataConstr} \rangle}\ \texttt{\{}\ \mathit{FBind}_1\ \texttt{,}\ \ldots\ \texttt{,}\ \mathit{FBind}_n\ \texttt{\}} & (\textit{record update},\ n \geqslant 1)
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{Alt} & ::= & \mathit{Pattern}\ \texttt{->}\ \mathit{Expr}\ [\texttt{where}\ \mathit{LocalDecls}] \\
 & | & \mathit{Pattern}\ \mathit{GdAlts}\ [\texttt{where}\ \mathit{LocalDecls}] \\
\mathit{GdAlts} & ::= & \texttt{|}\ \mathit{InfixExpr}\ \texttt{->}\ \mathit{Expr}\ [\mathit{GdAlts}]
\end{array}
$$

$$
\mathit{FBind} ::= \mathit{QLabel}\ \texttt{=}\ \mathit{Expr}
$$

$$
\begin{array}{rcll}
\mathit{Qual} & ::= & \mathit{Pattern}\ \texttt{<-}\ \mathit{Expr} & (\textit{generator}) \\
 & | & \texttt{let}\ \mathit{LocalDecls} & (\textit{local declarations}) \\
 & | & \mathit{Expr} & (\textit{guard})
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{Stmt} & ::= & \mathit{Pattern}\ \texttt{<-}\ \mathit{Expr} \\
 & | & \texttt{let}\ \mathit{LocalDecls} \\
 & | & \mathit{Expr}
\end{array}
$$

$$
\mathit{Literal} ::= \mathit{Int} \mid \mathit{Float} \mid \mathit{Char} \mid \mathit{String}
$$

$$
\begin{array}{rcll}
\mathit{GDataConstr} & ::= & \texttt{()} & (\textit{unit}) \\
 & | & \texttt{[]} & (\textit{empty list}) \\
 & | & \texttt{(,\{,\})} & (\textit{tuple}) \\
 & | & \mathit{QDataConstr} &
\end{array}
$$

$$
\begin{array}{rcll}
\mathit{Variable} & ::= & \mathit{VariableID} \mid \texttt{(}\ \mathit{InfixOpID}\ \texttt{)} & (\textit{variable}) \\
\mathit{Function} & ::= & \mathit{FunctionID} \mid \texttt{(}\ \mathit{InfixOpID}\ \texttt{)} & (\textit{function}) \\
\mathit{QFunction} & ::= & \mathit{QFunctionID} \mid \texttt{(}\ \mathit{QInfixOpID}\ \texttt{)} & (\textit{qualified function}) \\
\mathit{DataConstr} & ::= & \mathit{DataConstrID} \mid \texttt{(}\ \mathit{InfixOpID}\ \texttt{)} & (\textit{constructor}) \\
\mathit{QDataConstr} & ::= & \mathit{QDataConstrID} \mid \texttt{(}\ \mathit{QInfixOpID}\ \texttt{)} & (\textit{qualified constructor}) \\
\mathit{Label} & ::= & \mathit{LabelID} \mid \texttt{(}\ \mathit{InfixOpID}\ \texttt{)} & (\textit{label}) \\
\mathit{QLabel} & ::= & \mathit{QLabelID} \mid \texttt{(}\ \mathit{QInfixOpID}\ \texttt{)} & (\textit{qualified label})
\end{array}
$$

$$
\begin{array}{rcll}
\mathit{VarOp} & ::= & \mathit{InfixOpID} \mid \texttt{`}\ \mathit{VariableID}\ \texttt{`} & (\textit{variable operator}) \\
\mathit{FunOp} & ::= & \mathit{InfixOpID} \mid \texttt{`}\ \mathit{FunctionID}\ \texttt{`} & (\textit{function operator}) \\
\mathit{QFunOp} & ::= & \mathit{QInfixOpID} \mid \texttt{`}\ \mathit{QFunctionID}\ \texttt{`} & (\textit{qualified function operator}) \\
\mathit{ConOp} & ::= & \mathit{InfixOpID} \mid \texttt{`}\ \mathit{DataConstrID}\ \texttt{`} & (\textit{constructor operator}) \\
\mathit{QConOp} & ::= & \mathit{GConSym} \mid \texttt{`}\ \mathit{QDataConstrID}\ \texttt{`} & (\textit{qualified constructor operator}) \\
\mathit{LabelOp} & ::= & \mathit{InfixOpID} \mid \texttt{`}\ \mathit{LabelID}\ \texttt{`} & (\textit{label operator}) \\
\mathit{QLabelOp} & ::= & \mathit{QInfixOpID} \mid \texttt{`}\ \mathit{QLabelID}\ \texttt{`} & (\textit{qualified label operator})
\end{array}
$$

$$
\begin{array}{rcll}
\mathit{Op} & ::= & \mathit{FunOp} \mid \mathit{ConOp} \mid \mathit{LabelOp} & (\textit{operator}) \\
\mathit{QOp} & ::= & \mathit{VarOp} \mid \mathit{QFunOp} \mid \mathit{QConOp} \mid \mathit{QLabelOp} & (\textit{qualified operator}) \\
\mathit{GConSym} & ::= & \texttt{:} \mid \mathit{QInfixOpID} & (\textit{general constructor symbol})
\end{array}
$$

Appendix B

Proofs

Faith is different from proof; the latter is human, the former is a Gift from God.

Blaise Pascal

In this chapter we provide the missing proofs for the correctness and termination results stated in Chapter 7 as well as some auxiliary results.

B.1 Residualizing Semantics

To show the correctness of the residualizing semantics, we show its soundness and completeness individually. We like to note that Lemma 5.15 also holds for the residualizing semantics by the same proof and that annotations are ignored in $\Downarrow_1$.

Lemma B.1 (Soundness of Residualizing Semantics)**.** $\Gamma_S : e \Downarrow_{PE} \Theta : r$ *and* $(\Theta \uplus \Gamma_D) : r \Downarrow_1 \Delta : v$ *imply* $(\Gamma_S \uplus \Gamma_D) : e \Downarrow_1 \Delta : v$*.*

Proof. Note that $\Gamma_S : e \Downarrow_{PE} \Theta : r$ implies $\mathcal{D}om(\Gamma_S) \subseteq \mathcal{D}om(\Theta)$ since there is no rule such that a variable bound in the in-configuration is not bound in the respective out-configuration, so that $(\Gamma_S \uplus \Gamma_D) : e$ is a valid configuration. We show the claim by structural induction on the derivation for $\Gamma_S : e \Downarrow_{PE} \Theta : r$ and start with the base cases.

(Value) We have to show that $\Gamma_S : r \Downarrow_{PE} \Gamma_S : r$ and $(\Gamma_S \uplus \Gamma_D) : r \Downarrow_1 \Delta : v$ imply $(\Gamma_S \uplus \Gamma_D) : r \Downarrow_1 \Delta : v$, which directly holds.

(FunDefer) We have to show that $\Gamma_S : f(\overline{x_n}) \Downarrow_{PE} \Gamma_S : \langle\!\langle f(\overline{x_n}) \rangle\!\rangle$ where $f \in \mathcal{F}^{(n)}$ and $(\Gamma_S \uplus \Gamma_D) : f(\overline{x_n}) \Downarrow_1 \Delta : v$ imply $(\Gamma_S \uplus \Gamma_D) : f(\overline{x_n}) \Downarrow_1 \Delta : v$, which directly holds.

We continue with the inductive cases and assume as the induction hypothesis that the claim holds for the premises of the assumed derivation.

(VarExp) We have to show that $\Gamma_S[x \mapsto e] : x \Downarrow_{PE} \Theta[x \mapsto r] : r$ and $(\Theta[x \mapsto r] \uplus \Gamma_D) : r \Downarrow_1 \Delta : v$ imply $(\Gamma_S[x \mapsto e] \uplus \Gamma_D) : x \Downarrow_1 \Delta : v$, where $e \notin \{\text{free}, \blacksquare\}$ and $r \in \mathcal{V}$ with $\Theta(r) = \text{free}$ or $r = \phi(\overline{x_k})$ with $\phi \in \mathcal{C}$ or $k < \text{arity}(\phi)$. The induction hypothesis states that $\Gamma_S[x \mapsto \blacksquare] : e \Downarrow_{PE} \Theta : r$ and $(\Theta \uplus \Gamma_D) : r \Downarrow_1 \Delta' : v'$ imply

$(\Gamma_S[x \mapsto \blacksquare] \uplus \Gamma_D) : e \Downarrow_1 \Delta' : v'$. Since r is a value, $(\Theta[x \mapsto r] \uplus \Gamma_D) : r \Downarrow_1 \Delta : v$ must hold by rule (Value) with $\Delta : v = (\Theta[x \mapsto r] \uplus \Gamma_D) : r$, and thus $(\Theta \uplus \Gamma_D) : r \Downarrow_1 \Delta' : v'$ also holds by rule (Value) with $\Delta' : v' = (\Theta \uplus \Gamma_D) : r$. By application of rule (VarExp), we can then construct the derivation

$$\dfrac{\begin{array}{rc} & (\Gamma_S[x \mapsto \blacksquare] \uplus \Gamma_D) : e \Downarrow_1 \Delta' : v' \\ = & (\Gamma_S \uplus \Gamma_D)[x \mapsto \blacksquare] : e \Downarrow_1 (\Theta \uplus \Gamma_D) : r \end{array}}{\begin{array}{rc} & (\Gamma_S \uplus \Gamma_D)[x \mapsto e] : x \Downarrow_1 (\Theta \uplus \Gamma_D)[x \mapsto r] : r \\ = & (\Gamma_S[x \mapsto e] \uplus \Gamma_D) : x \Downarrow_1 \Delta : v \end{array}}$$

where the premise follows from the induction hypothesis.

(VarDefer) For this rule we have to show that $\Gamma_S[x \mapsto e] : x \Downarrow_{PE} \Theta[x \mapsto r] : \langle\!\langle x \rangle\!\rangle$ and $(\Theta[x \mapsto r] \uplus \Gamma_D) : x \Downarrow_1 \Delta[x \mapsto v] : v$ imply $(\Gamma_S[x \mapsto e] \uplus \Gamma_D) : x \Downarrow_1 \Delta[\overline{x \mapsto v}] : v$, where $e \notin \{\text{free}, \blacksquare\}$ and $r \in \mathcal{V}$ with $r \notin \mathcal{D}om(\Theta)$ or $r = \texttt{case}\ e'\ \texttt{of}\ \{\ \overline{p_k \to \langle\!\langle e_k \rangle\!\rangle}\ \}$ or $r = \langle\!\langle e' \rangle\!\rangle$. The induction hypothesis states that $\Gamma_S[x \mapsto \blacksquare] : e \Downarrow_{PE} \Theta : r$ and $(\Theta \uplus \Gamma_D) : r \Downarrow_1 \Delta : v$ imply $(\Gamma_S[x \mapsto \blacksquare] \uplus \Gamma_D) : e \Downarrow_1 \Delta : v$. Note that $\Gamma_S[x \mapsto \blacksquare] : e \Downarrow_{PE} \Theta : r$ implies $\Theta(x) = \blacksquare$ by Lemma 5.15, so that $(\Theta[x \mapsto r] \uplus \Gamma_D) : x \Downarrow_1 \Delta[x \mapsto v] : v$ implies $(\Theta \uplus \Gamma_D) : r \Downarrow_1 \Delta : v$ by rule (VarExp). We can then apply rule (VarExp) to construct the derivation

$$\dfrac{\begin{array}{rc} & (\Gamma_S[x \mapsto \blacksquare] \uplus \Gamma_D) : e \Downarrow_1 \Delta : v \\ = & (\Gamma_S \uplus \Gamma_D)[x \mapsto \blacksquare] : e \Downarrow_1 \Delta : v \end{array}}{\begin{array}{rc} & (\Gamma_S \uplus \Gamma_D)[x \mapsto e] : x \Downarrow_1 \Delta[x \mapsto v] : v \\ = & (\Gamma_S[x \mapsto e] \uplus \Gamma_D) : x \Downarrow_1 \Delta[x \mapsto v] : v \end{array}}$$

where the premise follows from the induction hypothesis.

(Flatten) We have to show that $\Gamma_S : \phi(\overline{e_k}) \Downarrow_{PE} \Theta : r$ and $(\Theta \uplus \Gamma_D) : r \Downarrow_1 \Delta : v$ imply $(\Gamma_S \uplus \Gamma_D) : \phi(\overline{e_k}) \Downarrow_1 \Delta : v$, where $\exists i \in \{1, \ldots, k\}$ such that $e_i \notin \mathcal{V}$ and $(\overline{y_l}, \overline{e'_l}, \overline{x_k}) = \text{splitArgs}(\overline{e_k})$. The induction hypothesis states that $\Gamma_S[\overline{y_l \mapsto e'_l}] : \phi(\overline{x_k}) \Downarrow_{PE} \Theta : r$ and $(\Theta \uplus \Gamma_D) : r \Downarrow_1 \Delta : v$ imply $(\Gamma_S[\overline{y_l \mapsto e'_l}] \uplus \Gamma_D) : \phi(\overline{x_k}) \Downarrow_1 \Delta : v$. Since $\overline{y_l}$ are fresh variables, it holds $(\Gamma_S \uplus \Gamma_D)[\overline{y_l \mapsto e'_l}] = \Gamma_S[\overline{y_l \mapsto e'_l}] \uplus \Gamma_D$ so that we can apply rule (Flatten) to construct the derivation

$$\dfrac{\begin{array}{rc} & (\Gamma_S[\overline{y_l \mapsto e'_l}] \uplus \Gamma_D) : \phi(\overline{x_k}) \Downarrow_1 \Delta : v \\ = & (\Gamma_S \uplus \Gamma_D)[\overline{y_l \mapsto e'_l}] : \phi(\overline{x_k}) \Downarrow_1 \Delta : v \end{array}}{(\Gamma_S \uplus \Gamma_D) : \phi(\overline{e_k}) \Downarrow_1 \Delta : v}$$

and the premise follows from the induction hypothesis.

(FunEval), (Let), (Or), (Free) The reasoning for these rules is analogous to the reasoning for rule (Flatten), and the claim follows from the induction hypothesis and application of the same rule.

(Select) We have to show that $\Gamma_S : \texttt{case}\ e\ \texttt{of}\ \{\ \overline{p_k \to e_k}\ \} \Downarrow_{PE} \Theta : r$ and $(\Theta \uplus \Gamma_D) : r \Downarrow_1 \Omega : v$ imply $(\Gamma_S \uplus \Gamma_D) : \texttt{case}\ e\ \texttt{of}\ \{\ \overline{p_k \to e_k}\ \} \Downarrow_1 \Omega : v$. The induction hypoth-

esis states that $\Gamma_S : e \Downarrow_{PE} \Theta' : c(\overline{x_n})$ and $(\Theta' \uplus \Gamma_D) : c(\overline{x_n}) \Downarrow_1 \Delta : c(\overline{x_n})$ imply $(\Gamma_S \uplus \Gamma_D) : e \Downarrow_1 \Delta : c(\overline{x_n})$, and that $\Theta' : \sigma(e_i) \Downarrow_{PE} \Theta : r$ and $(\Theta \uplus \Gamma_D) : r \Downarrow_1 \Omega : v$ imply $(\Theta' \uplus \Gamma_D) : \sigma(e_i) \Downarrow_1 \Omega : v$, where $c(\overline{x_n}) = \sigma(p_i)$ for $i \in \{1, \ldots, k\}$. Since $(\Theta' \uplus \Gamma_D) : c(\overline{x_n}) \Downarrow_1 \Delta : c(\overline{x_n})$ holds by rule (Value) for $\Delta = \Theta' \uplus \Gamma_D$, then $(\Gamma_S \uplus \Gamma_D) : e \Downarrow_1 (\Theta' \uplus \Gamma_D) : c(\overline{x_n})$ holds by the induction hypothesis. Since $(\Theta \uplus \Gamma_D) : r \Downarrow_1 \Omega : v$ holds by assumption, the induction hypothesis furthermore implies $(\Theta' \uplus \Gamma_D) : \sigma(e_i) \Downarrow_1 \Omega : v$ so that we can apply rule (Select) to construct the derivation

$$\frac{(\Gamma_S \uplus \Gamma_D) : e \Downarrow_1 (\Theta' \uplus \Gamma_D) : c(\overline{x_n}) \quad (\Theta' \uplus \Gamma_D) : \sigma(e_i) \Downarrow_1 \Omega : v}{(\Gamma_S \uplus \Gamma_D) : \mathtt{case}\ e\ \mathtt{of}\ \{\ \overline{p_k \to e_k}\ \} \Downarrow_1 \Omega : v}$$

(Guess) The reasoning for rule (Guess) is analogous to the reasoning for rule (Select), and the claim follows from the induction hypothesis.

(CaseVar) In this case we have to show that $\Gamma_S : \mathtt{case}\ e\ \mathtt{of}\ \{\ \overline{p_k \to e_k}\ \} \Downarrow_{PE} \Theta : \mathtt{case}\ x\ \mathtt{of}\ \{\ \overline{p_k \to \langle\!\langle e_k \rangle\!\rangle}\ \}$ and $(\Theta \uplus \Gamma_D) : \mathtt{case}\ x\ \mathtt{of}\ \{\ \overline{p_k \to e_k}\ \} \Downarrow_1 \Delta : v$ where $x \notin \mathcal{D}om(\Theta)$ imply $(\Gamma_S \uplus \Gamma_D) : \mathtt{case}\ e\ \mathtt{of}\ \{\ \overline{p_k \to e_k}\ \} \Downarrow_1 \Delta : v$, and the induction hypothesis states that $\Gamma_S : e \Downarrow_{PE} \Theta : x$ and $(\Theta \uplus \Gamma_D) : x \Downarrow_1 \Delta' : v'$ imply $(\Gamma_S \uplus \Gamma_D) : e \Downarrow_1 \Delta' : v'$. Since $(\Theta \uplus \Gamma_D) : \mathtt{case}\ x\ \mathtt{of}\ \{\ \overline{p_k \to e_k}\ \} \Downarrow_1 \Delta : v$ holds by assumption, there must exist a derivation

$$\frac{(\Theta \uplus \Gamma_D) : x \Downarrow_1 \Delta' : v' \quad \Delta'' : \theta(e_i) \Downarrow_1 \Delta : v}{(\Theta \uplus \Gamma_D) : \mathtt{case}\ x\ \mathtt{of}\ \{\ \overline{p_k \to e_k}\ \} \Downarrow_1 \Delta : v}$$

by either rule (Select) or rule (Guess). Hence, by application of the same rule we can construct the derivation

$$\frac{(\Gamma_S \uplus \Gamma_D) : e \Downarrow_1 \Delta' : v' \quad \Delta'' : \theta(e_i) \Downarrow_1 \Delta : v}{(\Gamma_S \uplus \Gamma_D) : \mathtt{case}\ e\ \mathtt{of}\ \{\ \overline{p_k \to e_k}\ \} \Downarrow_1 \Delta : v}$$

where the first premise follows from the induction hypothesis and the second premise from the derivation above.

(CaseDefer) The reasoning for rule (CaseDefer) is analogous to the reasoning for rule (CaseVar), and the claim follows from the induction hypothesis.

(CaseCase) We have to show that

$$\Gamma_S : \mathtt{case}\ e\ \mathtt{of}\ \{\ \overline{p_k \to e_k}\ \} \Downarrow_{PE} \Theta : \mathtt{case}\ e'\ \mathtt{of}\ \{\ \overline{p'_l \to \langle\!\langle \mathtt{case}\ e'_l\ \mathtt{of}\ \{\ \overline{p_k \to e_k}\ \} \rangle\!\rangle}\ \}$$

and

$$(\Theta \uplus \Gamma_D) : \mathtt{case}\ e'\ \mathtt{of}\ \{\ \overline{p'_l \to \mathtt{case}\ e'_l\ \mathtt{of}\ \{\ \overline{p_k \to e_k}\ \}}\ \} \Downarrow_1 \Delta : v$$

imply $(\Gamma_S \uplus \Gamma_D) : \mathtt{case}\ e\ \mathtt{of}\ \{\ \overline{p_k \to e_k}\ \} \Downarrow_1 \Delta : v$, where the induction hypothesis states that the statements $\Gamma_S : e \Downarrow_{PE} \Theta : \mathtt{case}\ e'\ \mathtt{of}\ \{\ \overline{p'_l \to \langle\!\langle e'_l \rangle\!\rangle}\ \}$ and $(\Theta \uplus \Gamma_D) : \mathtt{case}\ e'\ \mathtt{of}\ \{\ \overline{p'_l \to e'_l}\ \} \Downarrow_1 \Delta' : v'$ imply $(\Gamma_S \uplus \Gamma_D) : e \Downarrow_1 \Delta' : v'$. Note that for syntactic reasons, the variables introduced in $\overline{p'_l}$ cannot occur in $\overline{e_k}$. Thus, by

assumption there must exist a derivation

$$\dfrac{(\Theta \uplus \Gamma_D) : e' \Downarrow_1 \Delta_1 : v_1 \quad \dfrac{\Delta_2 : \theta(e'_j) \Downarrow_1 \Delta' : v' \quad \Delta_3 : \theta'(e_i) \Downarrow_1 \Delta : v}{\Delta_2 : \mathtt{case}\ \theta(e'_j)\ \mathtt{of}\ \{\ \overline{p_k \to e_k}\ \}) \Downarrow_1 \Delta : v}}{(\Theta \uplus \Gamma_D) : \mathtt{case}\ e'\ \mathtt{of}\ \{\ p'_l \to \mathtt{case}\ e'_l\ \mathtt{of}\ \{\ \overline{p_k \to e_k}\ \}\ \}\ \overline{\ }) \Downarrow_1 \Delta : v}$$

where $i \in \{1, \ldots, k\}$ and $j \in \{1, \ldots, l\}$ by application of rules (Select) or (Guess), and thus also a derivation

$$\dfrac{(\Theta \uplus \Gamma_D) : e' \Downarrow_1 \Delta_1 : v_1 \quad \Delta_2 : \theta(e'_j) \Downarrow_1 \Delta' : v'}{(\Theta \uplus \Gamma_D) : \mathtt{case}\ e'\ \mathtt{of}\ \{\ \overline{p'_l \to e'_l}\ \}\ \overline{\ }) \Downarrow_1 \Delta' : v'}$$

Thus, the induction hypothesis implies $(\Gamma_S \uplus \Gamma_D) : e \Downarrow_1 \Delta' : v'$ so that we can construct the derivation

$$\dfrac{(\Gamma_S \uplus \Gamma_D) : e \Downarrow_1 \Delta' : v' \quad \Delta_3 : \theta'(e_i) \Downarrow_1 \Delta : v}{(\Gamma_S \uplus \Gamma_D) : \mathtt{case}\ e\ \mathtt{of}\ \{\ \overline{p_k \to e_k}\ \}\ \overline{\ }) \Downarrow_1 \Delta : v}$$

by an application of either rule (Select) or (Guess), where the second premise holds by assumption as shown above. □

For the proof of completeness, we additionally need to show that partial evaluation does not increase the total number of steps necessary to evaluate an expression. Since the residualizing semantics contains some additional rules such as rule (FunDefer) or (CaseVar) that do not reduce the number of required evaluation steps thereafter, we only consider rules that are also present in the operational semantics. Furthermore, we exclude the rules (Value), (VarExp), (Flatten), (Let) and (Free) since they do not lead to a progress in evaluation and are mainly necessary for the implementation of sharing and local variable bindings.

Definition B.2 (Size of a Derivation)**.** *We define the size (number of evaluation steps) of a derivation D for a statement $\Gamma : e \Downarrow_1 \Delta : v$ or $\Gamma : e \Downarrow_{PE} \Delta : v$, denoted by $|D|$, as the number of applications of the rules (FunEval), (Select), (Guess), or (Or) in D.*

In consequence, the size of a derivation roughly corresponds to the number of narrowing steps in a narrowing derivation. Based on this notion, we require the number of steps applied in the operational semantics to equal the sum of steps applied for partial evaluation and evaluation of the residual configuration thereafter.

Lemma B.3 (Completeness of Residualizing Semantics)**.** *A derivation D for $(\Gamma_S \uplus \Gamma_D) : e \Downarrow_1 \Delta : v$ implies a derivation D' for $\Gamma_S : e \Downarrow_{PE} \Theta : r$ and a derivation D'' for $(\Theta \uplus \Gamma_D) : r \Downarrow_1 \Delta : v$ such that $|D| = |D'| + |D''|$.*

Proof. Note that $\mathcal{D}om(\Gamma_S) \cap \mathcal{D}om(\Gamma_D) = \varnothing$ and $\Gamma_S : e \Downarrow_{PE} \Theta : r$ imply $\mathcal{D}om(\Theta) \cap \mathcal{D}om(\Gamma_D) = \varnothing$ since every variable $x \in \mathcal{D}om(\Theta) \setminus \mathcal{D}om(\Gamma_S)$ must either originate from a local variable introduction or be introduced as a fresh variable, and thus $\Theta \uplus \Gamma_D$ denotes a valid heap. We show the claim by structural induction on the derivation for $(\Gamma_S \uplus \Gamma_D) : e \Downarrow_1 \Delta : v$.

(Value) For the base case of rule (Value), we have to show that a derivation D for $(\Gamma_S \uplus \Gamma_D) : v \Downarrow_1 (\Gamma_S \uplus \Gamma_D) : v$ implies for $v' \in \{v, \langle\!\langle v \rangle\!\rangle\}$ a derivation D' for $\Gamma_S : v' \Downarrow_{PE} \Theta : r$ and a derivation D'' for $(\Theta \uplus \Gamma_D) : r \Downarrow_1 (\Gamma_S \uplus \Gamma_D) : v$ such that $|D| = |D'| + |D''|$, where $v = \phi(\overline{x_k})$ with $\phi \in \mathcal{C}$ or $k < \mathsf{arity}(\phi)$ or $v \in \mathcal{V}$ with $(\Gamma_S \uplus \Gamma_D)(v) = \text{free}$. Since $(\Gamma_S \uplus \Gamma_D)(v) = \text{free}$ implies $\Gamma_S(v) = \text{free}$ or $v \notin \mathcal{D}om(\Gamma_S)$, then $\Gamma_S : v' \Downarrow_{PE} \Gamma_S : v'$ holds by rule (Value) with $|D'| = 0$, and $(\Gamma_S \uplus \Gamma_D) : v \Downarrow_1 (\Gamma_S \uplus \Gamma_D) : v$ holds by assumption with $D'' = D$ so that $|D| = |D'| + |D''|$.

We continue with the inductive cases and assume as the induction hypothesis that the claim holds for the premises of the assumed derivation. We first cover the case of e to be annotated in the residualizing semantics.

Annotated Expression We have to show that a derivation D for $(\Gamma_S \uplus \Gamma_D) : e \Downarrow_1 \Delta : v$ implies a derivation D' for $\Gamma_S : \langle\!\langle e \rangle\!\rangle \Downarrow_{PE} \Theta : r$ and a derivation D'' for $(\Theta \uplus \Gamma_D) : r \Downarrow_1 \Delta : v$ such that $|D| = |D'| + |D''|$. Since $\Gamma_S : \langle\!\langle e \rangle\!\rangle \Downarrow_{PE} \Gamma_S : \langle\!\langle e \rangle\!\rangle$ holds by rule (Value) with $|D'| = 0$ and $(\Gamma_S \uplus \Gamma_D) : e \Downarrow_1 \Delta : v$ by assumption with $D'' = D$, the claim follows.

For the remaining cases we can now assume that the expression to be evaluated is not annotated in the residualizing semantics.

(VarExp) We have to show that a derivation D for $\Gamma[x \mapsto e] : x \Downarrow_1 \Delta[x \mapsto v] : v$ with $\Gamma[x \mapsto e] = \Gamma_S \uplus \Gamma_D$ and $e \notin \{\text{free}, \blacksquare\}$ implies a derivation D' for $\Gamma_S : x \Downarrow_{PE} \Theta : r$ and a derivation D'' for $(\Theta \uplus \Gamma_D) : r \Downarrow_1 \Delta[x \mapsto v] : v$ such that $|D| = |D'| + |D''|$.

1. If $x \notin \mathcal{D}om(\Gamma_S)$, then $\Gamma_S : x \Downarrow_{PE} \Gamma_S : x$ holds by rule (Value) with $|D'| = 0$, and $(\Gamma_S \uplus \Gamma_D) : x \Downarrow_1 \Delta[x \mapsto v] : v$ holds by assumption with $D'' = D$ so that $|D| = |D'| + |D''|$.
2. If $x \in \mathcal{D}om(\Gamma_S)$, then $\Gamma_S = \Gamma'_S[x \mapsto e]$ and we have to show that a derivation D for $(\Gamma'_S \uplus \Gamma_D)[x \mapsto e] : x \Downarrow_1 \Delta[x \mapsto v] : v$ implies a derivation D' for $\Gamma'_S[x \mapsto e] : x \Downarrow_{PE} \Theta : r$ and a derivation D'' for $(\Theta \uplus \Gamma_D) : r \Downarrow_1 \Delta[x \mapsto v] : v$ such that $|D| = |D'| + |D''|$. The induction hypothesis states that a derivation D_1 for $(\Gamma'_S \uplus \Gamma_D)[x \mapsto \blacksquare] : e \Downarrow_1 \Delta : v$ implies a derivation D'_1 for $\Gamma'_S[x \mapsto \blacksquare] : e \Downarrow_{PE} \Theta' : r'$ and a derivation D''_1 for $(\Theta' \uplus \Gamma_D) : r' \Downarrow_1 \Delta : v$ such that $|D_1| = |D'_1| + |D''_1|$. We have $|D| = |D_1|$ and distinguish two cases for r'.
   (a) If $r' \in \mathcal{V}$ with $\Theta'(r') = \text{free}$ or $r' = \phi(\overline{x_k})$ with $\phi \in \mathcal{C}$ or $k < \mathsf{arity}(\phi)$, then we can apply rule (VarExp) to construct the derivation D' with $|D'| = |D'_1|$ as

$$\dfrac{\dfrac{\Gamma'_S[x \mapsto \blacksquare] : e \Downarrow_{PE} \Theta' : r'}{\Gamma'_S[x \mapsto e] : x \Downarrow_{PE} \Theta'[x \mapsto r'] : r'}}{= \qquad \Gamma_S : x \Downarrow_{PE} \Theta : r}$$

Furthermore, $(\Theta' \uplus \Gamma_D) : r' \Downarrow_1 \Delta : v$ then holds by rule (Value) with $\Delta : v = (\Theta' \uplus \Gamma_D) : r'$ and $|D_1''| = 0$. Thus, it remains to construct a derivation D'' for

$$\begin{array}{rc} & (\Theta \uplus \Gamma_D) : r \Downarrow_1 \Delta[x \mapsto v] : v \\ = & (\Theta'[x \mapsto r'] \uplus \Gamma_D) : r' \Downarrow_1 (\Theta' \uplus \Gamma_D)[x \mapsto r'] : r' \\ = & (\Theta' \uplus \Gamma_D)[x \mapsto r'] : r' \Downarrow_1 (\Theta' \uplus \Gamma_D)[x \mapsto r'] : r' \end{array}$$

which holds by rule (Value) with $|D''| = 0$, and thus $|D| = |D'| + |D''|$.

(b) If $r' \in \mathcal{V}$ with $r' \notin \mathcal{D}om(\Theta')$ or r' is a deferred or residual case expression, we can apply rule (VarDefer) to construct the derivation D' with $|D'| = |D_1'|$ as

$$\begin{array}{rc} & \dfrac{\Gamma_S'[x \mapsto \blacksquare] : e \Downarrow_{PE} \Theta' : r'}{\Gamma_S'[x \mapsto e] : x \Downarrow_{PE} \Theta'[x \mapsto r'] : \langle\!\langle x \rangle\!\rangle} \\ = & \Gamma_S : x \Downarrow_{PE} \Theta : r \end{array}$$

Thus, it remains to construct a derivation D'' for

$$\begin{array}{rc} & (\Theta \uplus \Gamma_D) : r \Downarrow_1 \Delta[x \mapsto v] : v \\ = & (\Theta'[x \mapsto r'] \uplus \Gamma_D) : x \Downarrow_1 \Delta[x \mapsto v] : v \end{array}$$

Since $\Gamma_S'[x \mapsto \blacksquare] : e \Downarrow_{PE} \Theta' : r'$ implies $\Theta'(x) = \blacksquare$ by Lemma 5.15, we can thus apply rule (VarExp) to construct the derivation D'' with $|D''| = |D_1''|$ as

$$\dfrac{\begin{array}{rc} & (\Theta' \uplus \Gamma_D) : r' \Downarrow_1 \Delta : v \\ = & (\Theta' \uplus \Gamma_D)[x \mapsto \blacksquare] : r' \Downarrow_1 \Delta : v \end{array}}{\begin{array}{rc} & (\Theta' \uplus \Gamma_D)[x \mapsto r'] : x \Downarrow_1 \Delta[x \mapsto v] : v \\ = & (\Theta'[x \mapsto r'] \uplus \Gamma_D) : x \Downarrow_1 \Delta[x \mapsto v] : v \end{array}}$$

such that $|D| = |D_1'| + |D_1''| = |D'| + |D''|$.

(Flatten) We have to show that a derivation D for $(\Gamma_S \uplus \Gamma_D) : \phi(\overline{e_k}) \Downarrow_1 \Delta : v$ where $\exists i \in \{1, \ldots, k\}$ such that $e_i \notin \mathcal{V}$ and $(\overline{y_l}, \overline{e_l'}, \overline{x_k}) = \mathsf{splitArgs}(\overline{e_k})$ implies a derivation D' for $\Gamma_S : \phi(\overline{e_k}) \Downarrow_{PE} \Theta : r$ and a derivation D'' for $(\Theta \uplus \Gamma_D) : r \Downarrow_1 \Delta : v$ such that $|D| = |D'| + |D''|$, and the induction hypothesis states that a derivation D_1 for $(\Gamma_S[\overline{y_l \mapsto e_l'}] \uplus \Gamma_D) : \phi(\overline{x_k}) \Downarrow_1 \Delta : v$ implies a derivation D_1' for $\Gamma_S[\overline{y_l \mapsto e_l'}] : \phi(\overline{x_k}) \Downarrow_{PE} \Theta' : r'$ and a derivation D_1'' for $(\Theta' \uplus \Gamma_D) : r' \Downarrow_1 \Delta : v$ such that $|D_1| = |D_1'| + |D_1''|$. We have $|D| = |D_1|$ and, since $\overline{y_l}$ are fresh variables, it holds $(\Gamma_S \uplus \Gamma_D)[\overline{y_l \mapsto e_l'}] = \Gamma_S[\overline{y_l \mapsto e_l'}] \uplus \Gamma_D$ so that we can apply the induction hypothesis. By application of rule (Flatten), we can then construct the derivation D' with $|D'| = |D_1'|$ as

$$\dfrac{\Gamma_S[\overline{y_l \mapsto e_l'}] : \phi(\overline{x_k}) \Downarrow_{PE} \Theta' : r'}{\Gamma_S : \phi(\overline{e_k}) \Downarrow_{PE} \Theta' : r'}$$

Thus, it remains to construct a derivation D'' for $(\Theta' \uplus \Gamma_D) : r' \Downarrow_1 \Delta : v$, which is covered by the induction hypothesis with $D'' = D_1''$ so that $|D| = |D'| + |D''|$.

(FunEval) We have to show that a derivation D for $(\Gamma_S \uplus \Gamma_D) : f(\overline{x_n}) \Downarrow_1 \Delta : v$ where

$f(\overline{x_n}) = e$ is a variable instance of a rule in P implies a derivation D' for $\Gamma_S : f(\overline{x_n}) \Downarrow_{PE} \Theta : r$ and a derivation D'' for $(\Theta \uplus \Gamma_D) : r \Downarrow_1 \Delta : v$ such that $|D| = |D'| + |D''|$, and the induction hypothesis states that a derivation D_1 for $(\Gamma_S \uplus \Gamma_D) : e \Downarrow_1 \Delta : v$ implies a derivation D_1' for $\Gamma_S : e \Downarrow_{PE} \Theta' : r'$ and a derivation D_1'' for $(\Theta' \uplus \Gamma_D) : r' \Downarrow_1 \Delta : v$ such that $|D_1| = |D_1'| + |D_1''|$, and it furthermore holds that $|D| = |D_1| + 1$. We distinguish two cases for the result of $\mathsf{proceed}(\ldots, \Gamma_S : f(\overline{x_n}))$.

1. If $\mathsf{proceed}(\ldots, \Gamma_S : f(\overline{x_n}))$ holds, we can apply rule (FunEval) to construct the derivation D' with $|D'| = |D_1'| + 1$ as
$$\frac{\Gamma_S : e \Downarrow_{PE} \Theta' : r'}{\Gamma_S : f(\overline{x_n}) \Downarrow_{PE} \Theta' : r'}$$
Thus, it remains to construct a derivation D'' for $(\Theta' \uplus \Gamma_D) : r' \Downarrow_1 \Delta : v$, which is covered by the induction hypothesis with $D'' = D_1''$ so that $|D| = |D'| + |D''|$.
2. If $\mathsf{proceed}(\ldots, \Gamma_S : f(\overline{x_n}))$ does not hold, we can apply rule (FunDefer) to construct the derivation D' with $|D'| = 0$ as
$$\Gamma_S : f(\overline{x_n}) \Downarrow_{PE} \Gamma_S : \langle\!\langle f(\overline{x_n}) \rangle\!\rangle$$
It remains to construct a derivation D'' for $(\Gamma_S \uplus \Gamma_D) : f(\overline{x_n}) \Downarrow_1 \Delta : v$, which holds by assumption with $D'' = D$ so that $|D| = |D'| + |D''|$.

(Let), (Or), (Free) The reasoning for these rules is analogous to the reasoning for rule (Flatten), and the claim follows from the induction hypothesis and by application of the same rule.

(Select) In this case we have to show that a derivation D for the statement $(\Gamma_S \uplus \Gamma_D) : \mathsf{case}\ e\ \mathsf{of}\ \{\ \overline{p_k \to e_k}\ \} \Downarrow_1 \Omega : v$ implies a derivation D' for the statement $\Gamma_S : \mathsf{case}\ e\ \mathsf{of}\ \{\ \overline{p_k \to e_k}\ \} \Downarrow_{PE} \Theta : r$ and a derivation D'' for $(\Theta \uplus \Gamma_D) : r \Downarrow_1 \Omega : v$ such that $|D| = |D'| + |D''|$. The induction hypothesis states that a derivation D_1 for $(\Gamma_S \uplus \Gamma_D) : e \Downarrow_1 \Delta : c(\overline{x_n})$ implies a derivation D_1' for $\Gamma_S : e \Downarrow_{PE} \Theta' : r'$ and a derivation D_1'' for $(\Theta' \uplus \Gamma_D) : r' \Downarrow_1 \Delta : c(\overline{x_n})$ such that $|D_1| = |D_1'| + |D_1''|$, and that for $\Delta = \Delta_S \uplus \Delta_D$ a derivation D_2 for $\Delta : \sigma(e_i) \Downarrow_1 \Omega : v$ implies a derivation D_2' for $\Delta_S : \sigma(e_i) \Downarrow_{PE} \Theta'' : r''$ and a derivation D_2'' for $(\Theta'' \uplus \Delta_D) : r'' \Downarrow_1 \Omega : v$ such that $|D_2| = |D_2'| + |D_2''|$, where $c(\overline{x_n}) = \sigma(p_i)$ for $i \in \{1, \ldots, k\}$. Furthermore, $|D| = 1 + |D_1| + |D_2|$. Then r' must either be a constructor application $c(\overline{x_n})$, an unbound variable, or a deferred or residual case expression, and we distinguish these cases in the following.

1. If $r' = c(\overline{x_n})$, then $(\Theta' \uplus \Gamma_D) : r' \Downarrow_1 \Delta : c(\overline{x_n})$ must hold by rule (Value) with $\Delta = \Theta' \uplus \Gamma_D$ and $|D_1''| = 0$. We can then apply rule (Select) to construct the derivation D' with $|D'| = 1 + |D_1'| + |D_2'|$ as
$$\frac{\Gamma_S : e \Downarrow_{PE} \Theta' : c(\overline{x_n}) \quad \Theta' : \sigma(e_i) \Downarrow_{PE} \Theta'' : r''}{\Gamma_S : \mathsf{case}\ e\ \mathsf{of}\ \{\ \overline{p_k \to e_k}\ \} \Downarrow_{PE} \Theta'' : r''}$$

where both premises follow from the induction hypothesis. Thus, it remains to construct a derivation D'' for $(\Theta'' \uplus \Gamma_D) : r'' \Downarrow_1 \Omega : v$, which holds by the induction hypothesis with $D'' = D_2''$ so that $|D| = 1 + |D_1| + |D_2| = 1 + |D_1'| + |D_1''| + |D_2'| + |D_2''| = |D'| + |D''|$.

2. If $r' = x \in \mathcal{V}$ with $x \notin \mathcal{D}om(\Gamma_S)$, we can apply rule (CaseVar) to construct the derivation D' as

$$\frac{\Gamma_S : e \Downarrow_{PE} \Theta' : x}{\Gamma_S : \texttt{case}\ e\ \texttt{of}\ \{\ \overline{p_k \to e_k}\ \} \Downarrow_{PE} \Theta' : \texttt{case}\ x\ \texttt{of}\ \{\ \overline{p_k \to \langle\!\langle e_k \rangle\!\rangle}\ \}}$$

with $|D'| = |D_1'|$. Furthermore, we can apply rule (Select) to construct the derivation D'' as

$$\frac{(\Theta' \uplus \Gamma_D) : x \Downarrow_1 \Delta : c(\overline{x_n}) \quad \Delta : \sigma(e_i) \Downarrow_1 \Omega : v}{(\Theta' \uplus \Gamma_D) : \texttt{case}\ x\ \texttt{of}\ \{\ \overline{p_k \to e_k}\ \} \Downarrow_1 \Omega : v}$$

where the first premise follows from the induction hypothesis and the second premise from the assumption. Furthermore, $|D''| = 1 + |D_1''| + |D_2|$ so that $|D| = |D'| + |D''|$.

3. If $r' = \langle\!\langle e' \rangle\!\rangle$, the claim follows with the same reasoning as the case above, except that rule (CaseDefer) instead of rule (CaseVar) is applied in the derivation D'.

4. If $r' = \texttt{case}\ e'\ \texttt{of}\ \{\ \overline{p_l' \to \langle\!\langle e_l' \rangle\!\rangle}\ \}$, we can apply rule (CaseCase) to construct the derivation D' with $|D'| = |D_1'|$ as

$$\frac{\Gamma_S : e \Downarrow_{PE} \Theta' : \texttt{case}\ e'\ \texttt{of}\ \{\ \overline{p_l' \to \langle\!\langle e_l' \rangle\!\rangle}\ \}}{\Gamma_S : \texttt{case}\ e\ \texttt{of}\ \{\ \overline{p_k \to e_k}\ \} \Downarrow_{PE} \Theta' : \texttt{case}\ e'\ \texttt{of}\ \{\ \overline{p_l' \to \langle\!\langle \texttt{case}\ e_l'\ \texttt{of}\ \{\ \overline{p_k \to e_k}\ \} \rangle\!\rangle}\ \}}$$

Note that for syntactic reasons, the variables introduced in $\overline{p_l'}$ cannot occur in $\overline{e_k}$. It now remains to construct a derivation D'' for

$$(\Theta' \uplus \Gamma_D) : \texttt{case}\ e'\ \texttt{of}\ \{\ \overline{p_l' \to \texttt{case}\ e_l'\ \texttt{of}\ \{\ \overline{p_k \to e_k}\ \}}\ \} \Downarrow_1 \Omega : v$$

By the induction hypothesis, there exists a derivation D_1'' of the form

$$\frac{(\Theta' \uplus \Gamma_D) : e' \Downarrow_1 \Delta' : v' \quad \Delta'' : \theta(e_j') \Downarrow_1 \Delta : c(\overline{x_n})}{(\Theta' \uplus \Gamma_D) : \texttt{case}\ e'\ \texttt{of}\ \{\ \overline{p_l' \to e_l'}\ \} \Downarrow_1 \Delta : c(\overline{x_n})}$$

with $j \in \{1, \ldots, l\}$ using either rule (Select) or rule (Guess). Thus, we can construct the derivation D'' as

$$\frac{(\Theta' \uplus \Gamma_D) : e' \Downarrow_1 \Delta' : v' \quad \dfrac{\Delta'' : \theta(e_j') \Downarrow_1 \Delta : c(\overline{x_n}) \quad \Delta : \sigma(e_i) \Downarrow_1 \Omega : v}{\Delta'' : \texttt{case}\ \theta(e_j')\ \texttt{of}\ \{\ \overline{p_k \to e_k}\ \}) \Downarrow_1 \Omega : v}}{(\Theta' \uplus \Gamma_D) : \texttt{case}\ e'\ \texttt{of}\ \{\ \overline{p_l' \to \texttt{case}\ e_l'\ \texttt{of}\ \{\ \overline{p_k \to e_k}\ \}}\ \} \Downarrow_1 \Omega : v}$$

with $|D''| = 1 + |D_1''| + |D_2|$ so that $|D| = |D'| + |D''|$.

(Guess) This case follows with the same reasoning as for the previous case. □

We can now combine both results about the soundness and completeness of the residualizing semantics to state its correctness.

Theorem 7.9 (Correctness of Residualizing Semantics)**.** $(\Gamma_S \uplus \Gamma_D) : e \Downarrow_1 \Delta : v$ *if and only if* $\Gamma_S : e \Downarrow_{PE} \Theta : r$ *and* $(\Theta \uplus \Gamma_D) : r \Downarrow_1 \Delta : v$.

Proof. Direct consequence of Lemma B.1 and Lemma B.3. □

Finally, we show that for every successful derivation in $\Downarrow_{PE}$ without an application of rule (FunEval), the complexities of the in-configurations strictly decrease. For this purpose we define the computation of the depth of expressions to not consider annotations, i. e., it holds $\mathsf{depth}(\langle\!\langle e \rangle\!\rangle) = \mathsf{depth}(e)$.

Lemma 7.10 (Strictly Decreasing Complexity of In-Configurations in $\Downarrow_{PE}$ without (FunEval))**.** *The following conditions hold for every derivation D with respect to* $\Downarrow_{PE}$ *that does not employ rule (FunEval):*

1. *If D is a derivation of the form*

$$\frac{C_1 \Downarrow_{PE} C_1' \quad \ldots \quad C_n \Downarrow_{PE} C_n'}{C \Downarrow_{PE} C'}$$

   where $n > 0$*, it holds that* $\mathcal{M}_{C_i} <_{\mathsf{mul}} \mathcal{M}_C$ *for all* $i \in \{1, \ldots, n\}$*, i. e., the complexities of the in-configurations of the premises are strictly smaller.*

2. *If D is a derivation for* $\Gamma : e \Downarrow_{PE} \Delta : v$*, it holds that* $\mathcal{M}_{\Delta:v} \leqslant_{\mathsf{mul}} \mathcal{M}_{\Gamma:e}$*, i. e., the complexity does not increase, or v is a deferred or residual* `case` *expression.*

Proof. By structural induction on the assumed derivation, where we start with the base cases of rule (Value) and rule (FunDefer).

(Value) For rule (Value), the claim holds since $\mathcal{M}_{\Gamma:v} \leqslant_{\mathsf{mul}} \mathcal{M}_{\Gamma:v}$.

(FunDefer) For rule (FunDefer), the claim holds since $\mathcal{M}_{\Gamma:f(\overline{x_n})} \leqslant_{\mathsf{mul}} \mathcal{M}_{\Gamma:\langle\!\langle f(\overline{x_n}) \rangle\!\rangle}$.

We continue with the inductive cases and assume as the induction hypothesis that the claim holds for the premises of the assumed derivation.

(VarExp) We have to show that $\Gamma[x \mapsto e] : x \Downarrow_{PE} \Delta[x \mapsto v] : v$ where $e \notin \{\mathrm{free}, \blacksquare\}$ and v is a value implies $\mathcal{M}_{\Gamma[x\mapsto\blacksquare]:e} <_{\mathsf{mul}} \mathcal{M}_{\Gamma[x\mapsto e]:x}$ and $\mathcal{M}_{\Delta[x\mapsto v]:v} \leqslant_{\mathsf{mul}} \mathcal{M}_{\Gamma[x\mapsto e]:x}$. Since $\mathcal{M}_{\Gamma[x\mapsto\blacksquare]:e} = \mathcal{M}_{\Gamma:e} <_{\mathsf{mul}} \mathcal{M}_{\Gamma[x\mapsto e]:x}$, the first condition is satisfied. Furthermore, the induction hypothesis implies $\mathcal{M}_{\Delta:v} \leqslant_{\mathsf{mul}} \mathcal{M}_{\Gamma[x\mapsto\blacksquare]:e}$, so that $\mathcal{M}_{\Delta:v} \leqslant_{\mathsf{mul}} \mathcal{M}_{\Gamma:e}$ and thus $\mathcal{M}_{\Delta[x\mapsto v]:x} \leqslant_{\mathsf{mul}} \mathcal{M}_{\Gamma[x\mapsto e]:x}$. Since v is a value with $\mathsf{depth}(v) = \mathsf{depth}(x)$, then $\mathcal{M}_{\Delta[x\mapsto v]:v} \leqslant_{\mathsf{mul}} \mathcal{M}_{\Gamma[x\mapsto e]:x}$ also holds.

(VarDefer) We have to show that $\Gamma[x \mapsto e] : x \Downarrow_{PE} \Delta[x \mapsto v] : \langle\!\langle x \rangle\!\rangle$ where $e \notin \{\mathrm{free}, \blacksquare\}$ and $v \in \mathcal{V}$ with $v \notin \mathcal{D}om(\Delta)$ or v is a deferred or residual `case` expression implies $\mathcal{M}_{\Gamma[x\mapsto\blacksquare]:e} <_{\mathsf{mul}} \mathcal{M}_{\Gamma[x\mapsto e]:x}$, which directly holds. Furthermore, the second condition is trivially satisfied since $\langle\!\langle x \rangle\!\rangle$ is a deferred expression.

(Flatten) We have to show that $\Gamma : \phi(\overline{e_k}) \Downarrow_{PE} \Delta : v$ where $\exists i \in \{1,\ldots,k\}$ such that $e_i \notin \mathcal{V}$ and $(\overline{y_l}, \overline{e'_l}, \overline{x_k}) = \mathsf{splitArgs}(\overline{e_k})$ implies $\mathcal{M}_{\Gamma[\overline{y_l \mapsto e'_l}]:\phi(\overline{x_k})} <_{\mathsf{mul}} \mathcal{M}_{\Gamma:\phi(\overline{e_k})}$ and $\mathcal{M}_{\Delta:v} \leqslant_{\mathsf{mul}} \mathcal{M}_{\Gamma:\phi(\overline{e_k})}$ or v is a deferred or residual `case` expression. Since $\overline{e'_l}$ are non-variable expressions contained in $\overline{e_k}$, we have $\mathsf{depth}(e'_j) < \mathsf{depth}(\phi(\overline{e_k}))$ for all $j \in \{1,\ldots,l\}$. Furthermore, we have $\mathsf{depth}(\phi(\overline{x_k})) < \mathsf{depth}(\phi(\overline{e_k}))$ and thus $\mathcal{M}_{\Gamma[\overline{y_l \mapsto e'_l}]:\phi(\overline{x_k})} <_{\mathsf{mul}} \mathcal{M}_{\Gamma:\phi(\overline{e_k})}$. Finally, the induction hypothesis states that $\mathcal{M}_{\Delta:v} \leqslant_{\mathsf{mul}} \mathcal{M}_{\Gamma[\overline{y_l \mapsto e'_l}]:\phi(\overline{x_k})}$ or v is a deferred or residual `case` expression, which implies the second condition.

(Let), *(Or)*, *(Free)* These cases follow with the same reasoning as for rule (Flatten), since the complexity of the in-configuration of the premise is strictly smaller by the definition of depth and the second condition follows from the induction hypothesis.

(Select) We have to show that $\Gamma : \mathsf{case}\ e\ \mathsf{of}\ \{\ \overline{p_k \to e_k}\ \} \Downarrow_{PE} \Theta : v$ implies $\mathcal{M}_{\Gamma:e} <_{\mathsf{mul}} \mathcal{M}_{\Gamma:\mathsf{case}\ e\ \mathsf{of}\ \{\ \overline{p_k \to e_k}\ \}}$ and $\mathcal{M}_{\Delta:\sigma(e_i)} <_{\mathsf{mul}} \mathcal{M}_{\Gamma:\mathsf{case}\ e\ \mathsf{of}\ \{\ \overline{p_k \to e_k}\ \}}$ and that $\mathcal{M}_{\Theta:v} \leqslant_{\mathsf{mul}} \mathcal{M}_{\Gamma:\mathsf{case}\ e\ \mathsf{of}\ \{\ \overline{p_k \to e_k}\ \}}$ or v is a deferred or residual `case` expression. The induction hypothesis states that $\Gamma : e \Downarrow_{PE} \Delta : c(\overline{x_n})$ implies $\mathcal{M}_{\Delta:c(\overline{x_n})} \leqslant_{\mathsf{mul}} \mathcal{M}_{\Gamma:e}$, and that $\Delta : \sigma(e_i) \Downarrow_{PE} \Theta : v$ implies $\mathcal{M}_{\Theta:v} \leqslant_{\mathsf{mul}} \mathcal{M}_{\Delta:\sigma(e_i)}$ or v is a deferred or residual `case` expression, where $c(\overline{x_n}) = \sigma(p_i)$ for $i \in \{1,\ldots,k\}$. Firstly, $\mathcal{M}_{\Gamma:e} <_{\mathsf{mul}} \mathcal{M}_{\Gamma:\mathsf{case}\ e\ \mathsf{of}\ \{\ \overline{p_k \to e_k}\ \}}$ follows from the definition of depth. Furthermore, $\mathcal{M}_{\Delta:c(\overline{x_n})} \leqslant_{\mathsf{mul}} \mathcal{M}_{\Gamma:e}$ holds by the induction hypothesis, and then $\mathcal{M}_{\Delta:\sigma(e_i)} <_{\mathsf{mul}} \mathcal{M}_{\Gamma:\mathsf{case}\ e\ \mathsf{of}\ \{\ \overline{p_k \to e_k}\ \}}$ also holds since $\mathsf{depth}(\sigma(e_i)) < \mathsf{depth}(\mathsf{case}\ e\ \mathsf{of}\ \{\ \overline{p_k \to e_k}\ \})$ and $\mathsf{depth}(e) < \mathsf{depth}(\mathsf{case}\ e\ \mathsf{of}\ \{\ \overline{p_k \to e_k}\ \})$ by the definition of depth. Finally, the second condition follows from the induction hypothesis.

(Guess) We have to show that $\Gamma : \mathsf{case}\ e\ \mathsf{of}\ \{\ \overline{p_k \to e_k}\ \} \Downarrow_{PE} \Theta : v$ implies $\mathcal{M}_{\Gamma:e} <_{\mathsf{mul}} \mathcal{M}_{\Gamma:\mathsf{case}\ e\ \mathsf{of}\ \{\ \overline{p_k \to e_k}\ \}}$ and $\mathcal{M}_{\Delta[x \mapsto c(\overline{x_n}), \overline{x_n \mapsto \mathsf{free}}]:e_i} <_{\mathsf{mul}} \mathcal{M}_{\Gamma:\mathsf{case}\ e\ \mathsf{of}\ \{\ \overline{p_k \to e_k}\ \}}$ and that $\mathcal{M}_{\Theta:v} \leqslant_{\mathsf{mul}} \mathcal{M}_{\Gamma:\mathsf{case}\ e\ \mathsf{of}\ \{\ \overline{p_k \to e_k}\ \}}$ or v is a deferred or residual `case` expression, where $\Delta(x) = \mathsf{free}$ and $c(\overline{x_n}) = p_i$ for $i \in \{1,\ldots,k\}$. In this case the induction hypothesis states that $\Gamma : e \Downarrow_{PE} \Delta : x$ implies $\mathcal{M}_{\Delta:x} \leqslant_{\mathsf{mul}} \mathcal{M}_{\Gamma:e}$, and that $\Delta[x \mapsto c(\overline{x_n}), \overline{x_n \mapsto \mathsf{free}}] : e_i \Downarrow_{PE} \Theta : v$ implies $\mathcal{M}_{\Theta:v} \leqslant_{\mathsf{mul}} \mathcal{M}_{\Delta[x \mapsto c(\overline{x_n}), \overline{x_n \mapsto \mathsf{free}}]:e_i}$ or v is a deferred or residual `case` expression. Thus, $\mathcal{M}_{\Delta:x} \leqslant_{\mathsf{mul}} \mathcal{M}_{\Gamma:e} <_{\mathsf{mul}} \mathcal{M}_{\Gamma:\mathsf{case}\ e\ \mathsf{of}\ \{\ \overline{p_k \to e_k}\ \}}$ by the definition of depth and the induction hypothesis, and then $\mathcal{M}_{\Delta[x \mapsto c(\overline{x_n}), \overline{x_n \mapsto \mathsf{free}}]:e_i} <_{\mathsf{mul}} \mathcal{M}_{\Gamma:\mathsf{case}\ e\ \mathsf{of}\ \{\ \overline{p_k \to e_k}\ \}}$ since $\mathsf{depth}(x) = \mathsf{depth}(c(\overline{x_n}))$ and $\mathsf{depth}(e_i) < \mathsf{depth}(\mathsf{case}\ e\ \mathsf{of}\ \{\ \overline{p_k \to e_k}\ \})$. Finally, the second condition follows from the induction hypothesis.

(CaseVar), *(CaseDefer)*, *(CaseCase)* For these cases it is obvious that the condition $\mathcal{M}_{\Gamma:e} <_{\mathsf{mul}} \mathcal{M}_{\Gamma:\mathsf{case}\ e\ \mathsf{of}\ \{\ \overline{p_k \to e_k}\ \}}$ is satisfied, and since all rules lead to a deferred or residual `case` expression as the value, the second condition also holds. □

B.2 Partial Evaluation

In this section we prove the correctness of partial evaluation as stated in Theorem 7.29. For this purpose, we first need some auxiliary results regarding the addition and removal of unreferenced bindings to a heap (Section B.2.1), the correctness of the dereferencing operation (Section B.2.2), the correctness of pre-partial evaluations (Section B.2.3), the correspondence between closedness and renaming (Section B.2.4), and the correctness of extensions of a configuration (Section B.2.5). Based on these results, we can finally provide the correctness proof of partial evaluation in Section B.2.6.

B.2.1 Additional Bindings

The first lemma states that for a successful derivation, we may add bindings for fresh variables to the heap of the in-configuration of the derived statement without affecting the value or other bindings in the heap of the out-configuration. Moreover, the added bindings remain unchanged.

Lemma B.4 (Addition of Bindings)**.** *Let Γ' be a heap and D a derivation for $\Gamma : e \Downarrow_1 \Delta : v$ such that every variable bound in Γ' or in a configuration in D is unique (introduced at most once). Then the derivation D for $\Gamma : e \Downarrow_1 \Delta : v$ and $\mathcal{D}om(\Gamma') \cap \mathcal{V}ar(\Gamma : e) = \varnothing$ imply $\mathcal{D}om(\Gamma') \cap \mathcal{V}ar(\Delta : v) = \varnothing$ and a derivation D' for $(\Gamma \uplus \Gamma') : e \Downarrow_1 (\Delta \uplus \Gamma') : v$ such that $|D| = |D'|$.*

Proof. Note that $\mathcal{D}om(\Gamma') \cap \mathcal{V}ar(\Gamma : e) = \varnothing$ implies $\mathcal{D}om(\Gamma') \cap \mathcal{D}om(\Gamma) = \varnothing$, and $\mathcal{D}om(\Gamma') \cap \mathcal{V}ar(\Delta : v) = \varnothing$ implies $\mathcal{D}om(\Gamma') \cap \mathcal{D}om(\Delta) = \varnothing$, so that $\Gamma \uplus \Gamma'$ and $\Delta \uplus \Gamma'$ denote valid heaps. We proceed by structural induction on the derivation of the premise.

(Value) For the base case of rule (Value) we have to show that a derivation D for $\Gamma : v \Downarrow_1 \Gamma : v$ and $\mathcal{D}om(\Gamma') \cap \mathcal{V}ar(\Gamma : v) = \varnothing$ imply $\mathcal{D}om(\Gamma') \cap \mathcal{V}ar(\Gamma : v) = \varnothing$ and a derivation D' for $(\Gamma \uplus \Gamma') : v \Downarrow_1 (\Gamma \uplus \Gamma') : v$ such that $|D| = 0 = |D'|$, where $v = \phi(\overline{x_k})$ with $\phi \in \mathcal{C}$ or $k < \mathsf{arity}(\phi)$, or $v \in \mathcal{V}$ with $\Gamma(v) = \text{free}$. Then $(\Gamma \uplus \Gamma') : v \Downarrow_1 (\Gamma \uplus \Gamma') : v$ holds by rule (Value), and the claim follows.

We continue with the inductive cases and assume as the induction hypothesis that the claim holds for the premises of a derivation.

(VarExp) We have to show that a derivation D for $\Gamma[x \mapsto e] : x \Downarrow_1 \Delta[x \mapsto v] : v$ and $\mathcal{D}om(\Gamma') \cap \mathcal{V}ar(\Gamma[x \mapsto e] : x) = \varnothing$ imply $\mathcal{D}om(\Gamma') \cap \mathcal{V}ar(\Delta[x \mapsto v] : v) = \varnothing$ and a derivation D' for $(\Gamma[x \mapsto e] \uplus \Gamma') : x \Downarrow_1 (\Delta[x \mapsto v] \uplus \Gamma') : v$ such that $|D| = |D'|$, where $e \notin \{\text{free}, \blacksquare\}$. The induction hypothesis states that a derivation D_1 for $\Gamma[x \mapsto \blacksquare] : e \Downarrow_1 \Delta : v$ and $\mathcal{D}om(\Gamma') \cap \mathcal{V}ar(\Gamma[x \mapsto \blacksquare] : e) = \varnothing$ imply $\mathcal{D}om(\Gamma') \cap \mathcal{V}ar(\Delta : v) = \varnothing$ and a derivation D_1' for $(\Gamma[x \mapsto \blacksquare] \uplus \Gamma') : e \Downarrow_1 (\Delta \uplus \Gamma') : v$ such

that $|D_1| = |D_1'|$. Since $\mathcal{D}om(\Gamma') \cap \mathcal{V}ar(\Gamma[x \mapsto e] : x) = \varnothing$ implies $\mathcal{D}om(\Gamma') \cap \mathcal{V}ar(\Gamma[x \mapsto \blacksquare] : e) = \varnothing$, we can construct the derivation D' with $|D'| = |D|$ as

$$\dfrac{\begin{array}{rl} & (\Gamma[x \mapsto \blacksquare] \uplus \Gamma') : e \Downarrow_1 (\Delta \uplus \Gamma') : v \\ = & (\Gamma \uplus \Gamma')[x \mapsto \blacksquare] : e \Downarrow_1 (\Delta \uplus \Gamma') : v \end{array}}{\begin{array}{rl} & (\Gamma \uplus \Gamma')[x \mapsto e] : x \Downarrow_1 (\Delta \uplus \Gamma')[x \mapsto v] : v \\ = & (\Gamma[x \mapsto e] \uplus \Gamma') : x \Downarrow_1 (\Delta[x \mapsto v] \uplus \Gamma') : v \end{array}}$$

where the premise follows from the induction hypothesis. Furthermore, $\mathcal{D}om(\Gamma') \cap \mathcal{V}ar(\Delta[x \mapsto v] : v) = \varnothing$ follows from $\mathcal{D}om(\Gamma') \cap \mathcal{V}ar(\Delta : v) = \varnothing$ and $x \notin \mathcal{D}om(\Gamma')$, which hold by assumption.

(Flatten) We have to show that a derivation D for $\Gamma : \phi(\overline{e_k}) \Downarrow_1 \Delta : v$ where $\exists i \in \{1, \ldots, k\}$ such that $e_i \notin \mathcal{V}$ and $(\overline{y_l}, \overline{e_l'}, \overline{x_k}) = \mathsf{splitArgs}(\overline{e_k})$ and $\mathcal{D}om(\Gamma') \cap \mathcal{V}ar(\Gamma : \phi(\overline{e_k})) = \varnothing$ imply $\mathcal{D}om(\Gamma') \cap \mathcal{V}ar(\Delta : v) = \varnothing$ and a derivation D' for $(\Gamma \uplus \Gamma') : \phi(\overline{e_k}) \Downarrow_1 (\Delta \uplus \Gamma') : v$ such that $|D| = |D'|$. In this case the induction hypothesis states that a derivation D_1 for $\Gamma[\overline{y_l \mapsto e_l'}] : \phi(\overline{x_k}) \Downarrow_1 \Delta : v$ and $\mathcal{D}om(\Gamma') \cap \mathcal{V}ar(\Gamma[\overline{y_l \mapsto e_l'}] : \phi(\overline{x_k})) = \varnothing$ imply $\mathcal{D}om(\Gamma') \cap \mathcal{V}ar(\Delta : v) = \varnothing$ and a derivation D_1' for $(\Gamma[\overline{y_l \mapsto e_l'}] \uplus \Gamma') : \phi(\overline{x_k}) \Downarrow_1 (\Delta \uplus \Gamma') : v$ such that $|D_1| = |D_1'|$. Since $\mathcal{D}om(\Gamma') \cap \mathcal{V}ar(\Gamma : \phi(\overline{e_k})) = \varnothing$ and $\overline{y_l}$ fresh imply $\mathcal{D}om(\Gamma') \cap \mathcal{V}ar(\Gamma[\overline{y_l \mapsto e_l'}] : \phi(\overline{x_k})) = \varnothing$, we can construct the derivation D' with $|D'| = |D|$ as

$$\dfrac{\begin{array}{rl} & (\Gamma[\overline{y_l \mapsto e_l'}] \uplus \Gamma') : \phi(\overline{x_k}) \Downarrow_1 (\Delta \uplus \Gamma') : v \\ = & (\Gamma \uplus \Gamma')[\overline{y_l \mapsto e_l'}] : \phi(\overline{x_k}) \Downarrow_1 (\Delta \uplus \Gamma') : v \end{array}}{(\Gamma \uplus \Gamma') : \phi(\overline{e_k}) \Downarrow_1 (\Delta \uplus \Gamma') : v}$$

where the premise and $\mathcal{D}om(\Gamma') \cap \mathcal{V}ar(\Delta : v) = \varnothing$ follow from the induction hypothesis.

(FunEval) We have to show that a derivation D for $\Gamma : f(\overline{x_n}) \Downarrow_1 \Delta : v$ and $\mathcal{D}om(\Gamma') \cap \mathcal{V}ar(\Gamma : f(\overline{x_n})) = \varnothing$ where $f(\overline{x_n}) = e$ is a variable instance of a rule in P imply $\mathcal{D}om(\Gamma') \cap \mathcal{V}ar(\Delta : v) = \varnothing$ and a derivation D' for $(\Gamma \uplus \Gamma') : f(\overline{x_n}) \Downarrow_1 (\Delta \uplus \Gamma') : v$ such that $|D| = |D'|$, and the induction hypothesis states that a derivation D_1 for $\Gamma : e \Downarrow_1 \Delta : v$ and $\mathcal{D}om(\Gamma') \cap \mathcal{V}ar(\Gamma : e) = \varnothing$ imply $\mathcal{D}om(\Gamma') \cap \mathcal{V}ar(\Delta : v) = \varnothing$ and a derivation D_1' for $(\Gamma \uplus \Gamma') : e \Downarrow_1 (\Delta \uplus \Gamma') : v$ such that $|D_1| = |D_1'|$. Since $\mathcal{UV}(e) \subseteq \{\overline{x_n}\}$, $\mathcal{D}om(\Gamma') \cap \mathcal{BV}(e) = \varnothing$ by the variable convention, and $\mathcal{D}om(\Gamma') \cap \mathcal{V}ar(\Gamma : f(\overline{x_n})) = \varnothing$, this implies $\mathcal{D}om(\Gamma') \cap \mathcal{V}ar(\Gamma : e) = \varnothing$ so that we can construct the derivation D' with $|D'| = |D|$ as

$$\dfrac{(\Gamma \uplus \Gamma') : e \Downarrow_1 (\Delta \uplus \Gamma') : v}{(\Gamma \uplus \Gamma') : f(\overline{x_n}) \Downarrow_1 (\Delta \uplus \Gamma') : v}$$

where the premise and $\mathcal{D}om(\Gamma') \cap \mathcal{V}ar(\Delta : v) = \varnothing$ follow from the induction hypothesis.

(Let), *(Free)* These cases follow with the same reasoning as for rule (Flatten).

(Or) This case follows with the same reasoning as for rule (FunEval).

(Select) We have to show that a derivation D for $\Gamma : \texttt{case}\ e\ \texttt{of}\ \{\ \overline{p_k \to e_k}\ \} \Downarrow_1 \Theta : v$ and $\mathcal{D}om(\Gamma') \cap \mathcal{V}ar(\Gamma : \texttt{case}\ e\ \texttt{of}\ \{\ \overline{p_k \to e_k}\ \}) = \varnothing$ imply $\mathcal{D}om(\Gamma') \cap \mathcal{V}ar(\Theta : v) = \varnothing$ and a derivation D' for $(\Gamma \uplus \Gamma') : \texttt{case}\ e\ \texttt{of}\ \{\ \overline{p_k \to e_k}\ \} \Downarrow_1 (\Theta \uplus \Gamma') : v$ such that $|D| = |D'|$, where the induction hypothesis states that a derivation D_1 for $\Gamma : e \Downarrow_1 \Delta : c(\overline{x_n})$ and $\mathcal{D}om(\Gamma') \cap \mathcal{V}ar(\Gamma : e) = \varnothing$ imply $\mathcal{D}om(\Gamma') \cap \mathcal{V}ar(\Delta : c(\overline{x_n})) = \varnothing$ and a derivation D_1' for $(\Gamma \uplus \Gamma') : e \Downarrow_1 (\Delta \uplus \Gamma') : c(\overline{x_n})$ such that $|D_1| = |D_1'|$, and that a derivation D_2 for $\Delta : \sigma(e_i) \Downarrow_1 \Theta : v$ and $\mathcal{D}om(\Gamma') \cap \mathcal{V}ar(\Delta : \sigma(e_i)) = \varnothing$ imply $\mathcal{D}om(\Gamma') \cap \mathcal{V}ar(\Theta : v) = \varnothing$ and a derivation D_2' for $(\Delta \uplus \Gamma') : \sigma(e_i) \Downarrow_1 (\Theta \uplus \Gamma') : v$ such that $|D_2| = |D_2'|$, where $c(\overline{x_n}) = \sigma(p_i)$ for $i \in \{1, \ldots, k\}$. Since $\mathcal{D}om(\Gamma') \cap \mathcal{V}ar(\Gamma : \texttt{case}\ e\ \texttt{of}\ \{\ \overline{p_k \to e_k}\ \}) = \varnothing$ implies $\mathcal{D}om(\Gamma') \cap \mathcal{V}ar(\Gamma : e) = \varnothing$, whereas $\mathcal{D}om(\Gamma') \cap \mathcal{V}ar(\Gamma : \texttt{case}\ e\ \texttt{of}\ \{\ \overline{p_k \to e_k}\ \}) = \varnothing$ and $\mathcal{D}om(\Gamma') \cap \mathcal{V}ar(\Delta : c(\overline{x_n})) = \varnothing$ imply $\mathcal{D}om(\Gamma') \cap \mathcal{V}ar(\Delta : \sigma(e_i)) = \varnothing$, we can construct the derivation D' with $|D'| = |D|$ as

$$\frac{(\Gamma \uplus \Gamma') : e \Downarrow_1 (\Delta \uplus \Gamma') : c(\overline{x_n}) \quad (\Delta \uplus \Gamma') : \sigma(e_i) \Downarrow_1 (\Theta \uplus \Gamma') : v}{(\Gamma \uplus \Gamma') : \texttt{case}\ e\ \texttt{of}\ \{\ \overline{p_k \to e_k}\ \} \Downarrow_1 (\Theta \uplus \Gamma') : v}$$

where the premises and $\mathcal{D}om(\Gamma') \cap \mathcal{V}ar(\Theta : v) = \varnothing$ follow from the induction hypothesis.

(Guess) This case follows with the same reasoning as for rule (Select). □

The following lemma states the opposite direction, i. e., bindings not transitively reachable from the expression of a configuration to be evaluated can be safely removed.

Lemma B.5 (Removal of Bindings)**.** *A derivation D for $(\Gamma \uplus \Gamma') : e \Downarrow_1 \Delta' : v$ and $\mathcal{D}om(\Gamma') \cap \mathcal{V}ar(\Gamma : e) = \varnothing$ imply $\Delta' = \Delta \uplus \Gamma'$ with $\mathcal{D}om(\Gamma') \cap \mathcal{V}ar(\Delta : v) = \varnothing$ and a derivation D' for $\Gamma : e \Downarrow_1 \Delta : v$ such that $|D| = |D'|$.*

Proof. By structural induction on the derivation of the premise.

(Value) For the base case of rule (Value) we have to show that a derivation D for $(\Gamma \uplus \Gamma') : v \Downarrow_1 (\Gamma \uplus \Gamma') : v$ and $\mathcal{D}om(\Gamma') \cap \mathcal{V}ar(\Gamma : v) = \varnothing$ imply $\Gamma \uplus \Gamma' = \Gamma \uplus \Gamma'$ with $\mathcal{D}om(\Gamma') \cap \mathcal{V}ar(\Gamma : v) = \varnothing$ and a derivation D' for $\Gamma : v \Downarrow_1 \Gamma : v$ with $|D| = |D'|$, where $v = \phi(\overline{x_k})$ with $\phi \in \mathcal{C}$ or $k < \mathsf{arity}(\phi)$, or $v \in \mathcal{V}$ with $\Gamma(v) = \text{free}$. Hence, $\Gamma : v \Downarrow_1 \Gamma : v$ holds by rule (Value) and the claim follows.

We continue with the inductive cases and assume as the induction hypothesis that the claim holds for the premises of a derivation.

(VarExp) We have to show that a derivation D for $\Gamma[x \mapsto e] \uplus \Gamma' : x \Downarrow_1 \Delta'[x \mapsto v] : v$ where $e \notin \{\text{free}, \blacksquare\}$ and $\mathcal{D}om(\Gamma') \cap \mathcal{V}ar(\Gamma[x \mapsto e] : x) = \varnothing$ imply $\Delta'[x \mapsto v] = \Delta[x \mapsto v] \uplus \Gamma'$ with $\mathcal{D}om(\Gamma') \cap \mathcal{V}ar(\Delta[x \mapsto v] : v) = \varnothing$ and a derivation D' for

$\Gamma[x \mapsto e] : x \Downarrow_1 \Delta[x \mapsto v] : v$ with $|D| = |D'|$. The induction hypothesis states that a derivation D_1 for $(\Gamma[x \mapsto \blacksquare] \uplus \Gamma') : e \Downarrow_1 \Delta' : v$ and $\mathcal{D}om(\Gamma') \cap \mathcal{V}ar(\Gamma[x \mapsto \blacksquare] : e) = \varnothing$ imply $\Delta' = \Delta \uplus \Gamma'$ with $\mathcal{D}om(\Gamma') \cap \mathcal{V}ar(\Delta : v) = \varnothing$ and a derivation D_1' for $\Gamma[x \mapsto \blacksquare] : e \Downarrow_1 \Delta : v$ with $|D_1| = |D_1'|$. Since $\mathcal{D}om(\Gamma') \cap \mathcal{V}ar(\Gamma[x \mapsto e] : x) = \varnothing$ implies $\mathcal{D}om(\Gamma') \cap \mathcal{V}ar(\Gamma[x \mapsto \blacksquare] : e) = \varnothing$, we can construct the derivation D' with $|D| = |D'|$ as

$$\frac{\Gamma[x \mapsto \blacksquare] : e \Downarrow_1 \Delta : v}{\Gamma[x \mapsto e] : x \Downarrow_1 \Delta[x \mapsto v] : v}$$

where the premise follows from the induction hypothesis. Furthermore, $\Delta'[x \mapsto v] = \Delta[x \mapsto v] \uplus \Gamma'$ follows from $\Delta' = \Delta \uplus \Gamma'$, whereas $\mathcal{D}om(\Gamma') \cap \mathcal{V}ar(\Delta[x \mapsto v] : v) = \varnothing$ follows from $\mathcal{D}om(\Gamma') \cap \mathcal{V}ar(\Delta : v) = \varnothing$ and $x \notin \mathcal{D}om(\Gamma')$, which holds by assumption.

(Flatten) We have to show that a derivation D for $(\Gamma \uplus \Gamma') : \phi(\overline{e_k}) \Downarrow_1 \Delta' : v$ where $\exists i \in \{1, \ldots, k\}$ such that $e_i \notin \mathcal{V}$ and $(\overline{y_l}, \overline{e_l'}, \overline{x_k}) = \mathsf{splitArgs}(\overline{e_k})$ and $\mathcal{D}om(\Gamma') \cap \mathcal{V}ar(\Gamma : \phi(\overline{e_k})) = \varnothing$ imply $\Delta' = \Delta \uplus \Gamma'$ with $\mathcal{D}om(\Gamma') \cap \mathcal{V}ar(\Delta : v) = \varnothing$ and a derivation D' for $\Gamma : \phi(\overline{e_k}) \Downarrow_1 \Delta : v$ with $|D| = |D'|$, and the induction hypothesis states that a derivation D_1 for $(\Gamma[\overline{y_l \mapsto e_l'}] \uplus \Gamma') : \phi(\overline{x_k}) \Downarrow_1 \Delta' : v$ and $\mathcal{D}om(\Gamma') \cap \mathcal{V}ar(\Gamma[\overline{y_l \mapsto e_l'}] : \phi(\overline{x_k})) = \varnothing$ imply $\Delta' = \Delta \uplus \Gamma'$ with $\mathcal{D}om(\Gamma') \cap \mathcal{V}ar(\Delta : v) = \varnothing$ and a derivation D_1' for $\Gamma[\overline{y_l \mapsto e_l'}] : \phi(\overline{x_k}) \Downarrow_1 \Delta : v$ with $|D_1| = |D_1'|$. Since $\mathcal{D}om(\Gamma') \cap \mathcal{V}ar(\Gamma : \phi(\overline{e_k})) = \varnothing$ and $\overline{y_l}$ fresh imply $\mathcal{D}om(\Gamma') \cap \mathcal{V}ar(\Gamma[\overline{y_l \mapsto e_l'}] : \phi(\overline{x_k})) = \varnothing$, we can construct the derivation D' with $|D| = |D'|$ as

$$\frac{\Gamma[\overline{y_l \mapsto e_l'}] : \phi(\overline{x_k}) \Downarrow_1 \Delta : v}{\Gamma : \phi(\overline{e_k}) \Downarrow_1 \Delta : v}$$

where the premise and $\Delta' = \Delta \uplus \Gamma'$ with $\mathcal{D}om(\Gamma') \cap \mathcal{V}ar(\Delta : v) = \varnothing$ follow from the induction hypothesis.

(FunEval) We have to show that a derivation D for $(\Gamma \uplus \Gamma') : f(\overline{x_n}) \Downarrow_1 \Delta' : v$ where $f(\overline{x_n}) = e$ is a variable instance of a rule in P and $\mathcal{D}om(\Gamma') \cap \mathcal{V}ar(\Gamma : f(\overline{x_n})) = \varnothing$ imply $\Delta' = \Delta \uplus \Gamma'$ with $\mathcal{D}om(\Gamma') \cap \mathcal{V}ar(\Delta : v) = \varnothing$ and a derivation D for $\Gamma : f(\overline{x_n}) \Downarrow_1 \Delta : v$ with $|D| = |D'|$, and the induction hypothesis states that a derivation D_1 for $(\Gamma \uplus \Gamma') : e \Downarrow_1 \Delta' : v$ and $\mathcal{D}om(\Gamma') \cap \mathcal{V}ar(\Gamma : e) = \varnothing$ imply $\Delta' = \Delta \uplus \Gamma'$ with $\mathcal{D}om(\Gamma') \cap \mathcal{V}ar(\Delta : v) = \varnothing$ and a derivation D_1' for $\Gamma : e \Downarrow_1 \Delta : v$ with $|D_1| = |D_1'|$. Since $\mathcal{UV}(e) \subseteq \{\overline{x_n}\}$, $\mathcal{BV}(e) \cap \mathcal{D}om(\Gamma') = \varnothing$ by the variable convention, and $\mathcal{D}om(\Gamma') \cap \mathcal{V}ar(\Gamma : f(\overline{x_n})) = \varnothing$, this implies $\mathcal{D}om(\Gamma') \cap \mathcal{V}ar(\Gamma : e) = \varnothing$ so that we can construct the derivation D' with $|D| = |D'|$ as

$$\frac{\Gamma : e \Downarrow_1 \Delta : v}{\Gamma : f(\overline{x_n}) \Downarrow_1 \Delta : v}$$

where the premise and $\Delta' = \Delta \uplus \Gamma'$ with $\mathcal{D}om(\Gamma') \cap \mathcal{V}ar(\Delta : v) = \varnothing$ follow from the induction hypothesis.

(Let), *(Free)* These cases follow with the same reasoning as for rule (Flatten).

(Or) This case follows with the same reasoning as for rule (FunEval).

(Select) We have to show that a derivation D for $(\Gamma \uplus \Gamma') : \texttt{case}\ e\ \texttt{of}\ \{\ \overline{p_k \to e_k}\ \} \Downarrow_1 \Theta' : v$ and $\mathcal{D}om(\Gamma') \cap \mathcal{V}ar(\Gamma : \texttt{case}\ e\ \texttt{of}\ \{\ \overline{p_k \to e_k}\ \}) = \varnothing$ imply $\Theta' = \Theta \uplus \Gamma'$ with $\mathcal{D}om(\Gamma') \cap \mathcal{V}ar(\Theta : v) = \varnothing$ and a derivation D' with $|D| = |D'|$ for the statement $\Gamma : \texttt{case}\ e\ \texttt{of}\ \{\ \overline{p_k \to e_k}\ \} \Downarrow_1 \Theta : v$, and the induction hypothesis states that a derivation D_1 for $(\Gamma \uplus \Gamma') : e \Downarrow_1 \Delta' : c(\overline{x_n})$ and $\mathcal{D}om(\Gamma') \cap \mathcal{V}ar(\Gamma : e) = \varnothing$ imply $\Delta' = \Delta \uplus \Gamma'$ with $\mathcal{D}om(\Gamma') \cap \mathcal{V}ar(\Delta : c(\overline{x_n})) = \varnothing$ and a derivation D_1' for $\Gamma : e \Downarrow_1 \Delta : c(\overline{x_n})$ with $|D_1| = |D_1'|$, and that a derivation D_2 for $(\Delta \uplus \Gamma') : \sigma(e_i) \Downarrow_1 \Theta' : v$ and $\mathcal{D}om(\Gamma') \cap \mathcal{V}ar(\Delta : \sigma(e_i)) = \varnothing$ imply $\Theta' = \Theta \uplus \Gamma'$ with $\mathcal{D}om(\Gamma') \cap \mathcal{V}ar(\Theta : v) = \varnothing$ and a derivation D_2' for $\Gamma : \sigma(e_i) \Downarrow_1 \Theta : v$ with $|D_2| = |D_2'|$, where $c(\overline{x_n}) = \sigma(p_i)$ for $i \in \{1, \ldots, k\}$. Since $\mathcal{D}om(\Gamma') \cap \mathcal{V}ar(\Gamma : \texttt{case}\ e\ \texttt{of}\ \{\ \overline{p_k \to e_k}\ \}) = \varnothing$ implies $\mathcal{D}om(\Gamma') \cap \mathcal{V}ar(\Gamma : e) = \varnothing$, whereas $\mathcal{D}om(\Gamma') \cap \mathcal{V}ar(\Gamma : \texttt{case}\ e\ \texttt{of}\ \{\ \overline{p_k \to e_k}\ \}) = \varnothing$ and $\mathcal{D}om(\Gamma') \cap \mathcal{V}ar(\Delta : c(\overline{x_n})) = \varnothing$ imply $\mathcal{D}om(\Gamma') \cap \mathcal{V}ar(\Delta : \sigma(e_i)) = \varnothing$, we can construct the derivation D' with $|D| = |D'|$ as

$$\frac{\Gamma : e \Downarrow_1 \Delta : c(\overline{x_n}) \quad \Delta : \sigma(e_i) \Downarrow_1 \Theta : v}{\Gamma : \texttt{case}\ e\ \texttt{of}\ \{\ \overline{p_k \to e_k}\ \} \Downarrow_1 \Theta : v}$$

where the premises and $\Theta' = \Theta \uplus \Gamma'$ with $\mathcal{D}om(\Gamma') \cap \mathcal{V}ar(\Theta : v) = \varnothing$ follow from the induction hypothesis.

(Guess) This case follows with the same reasoning as for rule (Select). □

Finally, we show that additional bindings do not affect the result of the recursive heap lookup operation.

Lemma B.6 (Heap Lookup with Additional Bindings)**.** *Let* $(\Gamma \uplus \Gamma')$ *be a heap and* x *a variable such that* $\mathcal{D}om(\Gamma') \cap \mathcal{V}ar(\Gamma : x) = \varnothing$*. Then* $(\Gamma \uplus \Gamma')^*(x) = \Gamma^*(x)$*.*

Proof. Note that $\mathcal{D}om(\Gamma') \cap \mathcal{V}ar(\Gamma : x) = \varnothing$ implies $x \notin \mathcal{D}om(\Gamma')$, so that the claim intuitively holds since the recursive heap lookup only consider bindings reachable from x. Formally, we show the claim by well-founded induction on the complexity of the heap Γ. The base case of $\mathcal{M}_\Gamma = \varnothing$ implies $\Gamma = []$, and thus $(\Gamma \uplus \Gamma')^*(x) = x = \Gamma^*(x)$. We continue with the inductive cases and assume as the induction hypothesis that the claim holds for all heaps Γ'' with $\mathcal{M}_{\Gamma''} <_{\mathsf{mul}} \mathcal{M}_\Gamma$.

- ▷ If $x \notin \mathcal{D}om(\Gamma)$ or $\Gamma(x) \in \{\text{free}, \blacksquare, x\}$, then $(\Gamma \uplus \Gamma')^*(x) = x = \Gamma^*(x)$.
- ▷ If $\Gamma = \Gamma''[x \mapsto y]$ with $x \neq y$, then $(\Gamma''[x \mapsto y] \uplus \Gamma')^*(x) = (\Gamma'' \uplus \Gamma')^*(y)$ and $\Gamma''[x \mapsto y]^*(x) = \Gamma''^*(y)$, which are equal by the induction hypothesis.
- ▷ If $\Gamma = \Gamma''[x \mapsto \phi(\overline{e_k})]$ with $\phi \in \mathcal{C}$ or $k < \mathsf{arity}(\phi)$, then for the left-hand side we have $(\Gamma''[x \mapsto \phi(\overline{e_k})] \uplus \Gamma')^*(x) = \phi(\overline{(\Gamma''[\overline{y_l \mapsto e_l'}] \uplus \Gamma')^*(x_k)})$ where $(\overline{y_l}, \overline{e_l'}, \overline{x_k}) = \mathsf{splitArgs}(\overline{e_k})$, and $\Gamma''[x \mapsto \phi(\overline{e_k})]^*(x) = \phi(\Gamma''[\overline{y_l \mapsto e_l'}]^*(x_k))$ for the right-hand side, which are equal by the induction hypothesis.
- ▷ For all other cases, we have $(\Gamma \uplus \Gamma')^*(x) = x = \Gamma^*(x)$. □

B.2.2 Dereferencing

The process of dereferencing does not affect the semantics of a configuration with respect to the operational semantics, which is stated in the following two lemmata.

Lemma B.7 (Soundness of Dereferencing)**.** *Let* Γ *and* Γ' *be two heaps with* $\blacksquare \notin \mathcal{R}an(\Gamma')$ *and* $\mathcal{D}om(\Gamma') \cap \mathcal{V}ar(\Gamma) = \varnothing$*. If* $\Gamma : \mathsf{drf}(\Gamma', e) \Downarrow_1 \Delta : v$*, then* $(\Gamma \uplus \Gamma') : e \Downarrow_1 \Delta : v$*.*

Proof. Note that $\mathcal{D}om(\Gamma') \cap \mathcal{V}ar(\Gamma) = \varnothing$ implies $\mathcal{D}om(\Gamma') \cap \mathcal{D}om(\Gamma) = \varnothing$ so that $\Gamma \uplus \Gamma'$ is a valid heap. We consider $\Gamma' = [\overline{x_k \mapsto \mathsf{free}}, \overline{y_l \mapsto e_l}]$ with $k, l \geqslant 0$ and proceed by well-founded induction on the depth of e. Note that annotations are generally disregarded in $\Downarrow_1$ and thus not explicitly mentioned in the proof.

Base Case If $\mathsf{depth}(e) = 1$, then e is not a residual `case` expression and $\mathsf{drf}(\Gamma', e) =$ `let` $\overline{x_k}$ `free in let` $\{\overline{y_l = e_l}\}$ `in` e. Then the premise must hold by the derivation

$$\dfrac{\dfrac{(\Gamma \uplus \Gamma') : e \Downarrow_1 \Delta : v}{= \quad \Gamma[\overline{x_k \mapsto \mathsf{free}}, \overline{y_l \mapsto e_l}] : e \Downarrow_1 \Delta : v}}{\dfrac{\Gamma[\overline{x_k \mapsto \mathsf{free}}] : \texttt{let } \{\overline{y_l = e_l}\} \texttt{ in } e \Downarrow_1 \Delta : v}{\Gamma : \texttt{let } \overline{x_k} \texttt{ free in let } \{\overline{y_l = e_l}\} \texttt{ in } e \Downarrow_1 \Delta : v}}$$

and the claim follows from the premise.

We continue with the inductive cases and assume as the induction hypothesis that the claim holds for all expressions e' such that $\mathsf{depth}(e') < \mathsf{depth}(e)$.

Residual `case` *Expression* If $e =$ `case` x `of` $\{ \overline{p_k \to e_k} \}$ is a residual `case` expression with $x \notin \mathcal{D}om(\Gamma')$, then $\mathsf{drf}(\Gamma', \texttt{case } x \texttt{ of } \{ \overline{p_k \to e_k} \}) =$ `case` x `of` $\{ \overline{p_k \to \mathsf{drf}(\Gamma', e_k)} \}$. Thus, we have to show that $\Gamma :$ `case` x `of` $\{ \overline{p_k \to \mathsf{drf}(\Gamma', e_k)} \} \Downarrow_1 \Theta : v$ implies $(\Gamma \uplus \Gamma') :$ `case` x `of` $\{ \overline{p_k \to e_k} \} \Downarrow_1 \Theta : v$, and the premise must hold by either rule (Select) or (Guess) which we consider separately.

(Select) In this case, the premise must hold by a derivation

$$\dfrac{\Gamma : x \Downarrow_1 \Delta : c(\overline{x_n}) \quad \Delta : \sigma(\mathsf{drf}(\Gamma', e_i)) \Downarrow_1 \Theta : v}{\Gamma : \texttt{case } x \texttt{ of } \{ \overline{p_k \to \mathsf{drf}(\Gamma', e_k)} \} \Downarrow_1 \Theta : v}$$

where $c(\overline{x_n}) = \sigma(p_i)$ for $i \in \{1, \ldots, k\}$. Then $\Gamma : x \Downarrow_1 \Delta : c(\overline{x_n})$ with $\mathcal{D}om(\Gamma') \cap \mathcal{V}ar(\Gamma) = \varnothing$ and $x \notin \mathcal{D}om(\Gamma')$ imply $(\Gamma \uplus \Gamma') : x \Downarrow_1 (\Delta \uplus \Gamma') : c(\overline{x_n})$ and $\mathcal{D}om(\Gamma') \cap \mathcal{V}ar(\Delta) = \varnothing$ by Lemma B.4. Since Γ' cannot reference variables introduced in p_i by the variable convention, we have $\sigma(\mathsf{drf}(\Gamma', e_i)) = \mathsf{drf}(\Gamma', \sigma(e_i))$ so that $\Delta : \mathsf{drf}(\Gamma', \sigma(e_i)) \Downarrow_1 \Theta : v$ implies $(\Delta \uplus \Gamma') : \sigma(e_i) \Downarrow_1 \Theta : v$ by the induction hypothesis. We can then construct the derivation

$$\dfrac{(\Gamma \uplus \Gamma') : x \Downarrow_1 (\Delta \uplus \Gamma') : c(\overline{x_n}) \quad (\Delta \uplus \Gamma') : \sigma(e_i) \Downarrow_1 \Theta : v}{(\Gamma \uplus \Gamma') : \texttt{case } x \texttt{ of } \{ \overline{p_k \to e_k} \} \Downarrow_1 \Theta : v}$$

(Guess) This case is analogous to the previous case with the only difference that the heap Δ is updated and no variable substitution σ is applied.

Remaining Cases For all other cases, the claim follows analogously to the base case. □

Lemma B.8 (Completeness of Dereferencing)**.** *Let* Γ *and* Γ' *be two heaps with* $\blacksquare \notin \mathcal{R}an(\Gamma')$ *and* $\mathcal{D}om(\Gamma') \cap \mathcal{V}ar(\Gamma) = \varnothing$*. Then a derivation* D *for* $(\Gamma \uplus \Gamma') : e \Downarrow_1 \Delta : v$ *implies a derivation* D' *for* $\Gamma : \mathsf{drf}(\Gamma', e) \Downarrow_1 \Delta : v$ *such that* $|D| = |D'|$*.*

Proof. We consider $\Gamma' = [\overline{x_k \mapsto \mathsf{free}}, \overline{y_l \mapsto e_l}]$ with $k, l \geqslant 0$ and proceed by well-founded induction on the depth of e. Note that annotations are generally disregarded in $\Downarrow_1$ and thus not explicitly mentioned in the proof.

Base Case If $\mathsf{depth}(e) = 1$, then e is not a residual `case` expression and by the definition of drf we have $\mathsf{drf}(\Gamma', e) = \texttt{let}\ \overline{x_k}\ \texttt{free in let}\ \{\overline{y_l = e_l}\}\ \texttt{in}\ e$. We can thus construct the derivation D' as

$$\dfrac{\dfrac{\begin{array}{c}(\Gamma \uplus \Gamma') : e \Downarrow_1 \Delta : v \\ = \quad \Gamma[\overline{x_k \mapsto \mathsf{free}}, \overline{y_l \mapsto e_l}] : e \Downarrow_1 \Delta : v\end{array}}{\Gamma[\overline{x_k \mapsto \mathsf{free}}] : \texttt{let}\ \{\overline{y_l = e_l}\}\ \texttt{in}\ e \Downarrow_1 \Delta : v}}{\Gamma : \texttt{let}\ \overline{x_k}\ \texttt{free in let}\ \{\overline{y_l = e_l}\}\ \texttt{in}\ e \Downarrow_1 \Delta : v}$$

and the claim follows from the premise and the fact that neither rule (Free) nor rule (Let) count as an evaluation step.

We continue with the inductive cases and assume as the induction hypothesis that the claim holds for all expressions e' such that $\mathsf{depth}(e') < \mathsf{depth}(e)$.

Residual `case` *Expression* If $e = \texttt{case}\ x\ \texttt{of}\ \{\ \overline{p_k \to e_k}\ \}$ is a residual `case` expression with $x \notin \mathcal{D}om(\Gamma')$, then $\mathsf{drf}(\Gamma', \texttt{case}\ x\ \texttt{of}\ \{\ \overline{p_k \to e_k}\ \}) = \texttt{case}\ x\ \texttt{of}\ \{\ \overline{p_k \to \mathsf{drf}(\Gamma', e_k)}\ \}$. Thus, we have to show that $(\Gamma \uplus \Gamma') : \texttt{case}\ x\ \texttt{of}\ \{\ \overline{p_k \to e_k}\ \} \Downarrow_1 \Theta : v$ implies $\Gamma : \texttt{case}\ x\ \texttt{of}\ \{\ \overline{p_k \to \mathsf{drf}(\Gamma', e_k)}\ \} \Downarrow_1 \Theta : v$. Then the premise must hold either by rule (Select) or (Guess), and we distinguish both cases.

(Select) In this case the derivation D is of the form

$$\dfrac{(\Gamma \uplus \Gamma') : x \Downarrow_1 \Delta' : c(\overline{x_n}) \quad \Delta' : \sigma(e_i) \Downarrow_1 \Theta : v}{(\Gamma \uplus \Gamma') : \texttt{case}\ x\ \texttt{of}\ \{\ \overline{p_k \to e_k}\ \} \Downarrow_1 \Theta : v}$$

where $c(\overline{x_n}) = \sigma(p_i)$ for $i \in \{1, \ldots, k\}$. Since $x \notin \mathcal{D}om(\Gamma')$ and $\mathcal{D}om(\Gamma') \cap \mathcal{V}ar(\Gamma) = \varnothing$ hold by assumption, the derivation D_1 for $(\Gamma \uplus \Gamma') : x \Downarrow_1 \Delta' : c(\overline{x_n})$ implies $\Delta' = \Delta \uplus \Gamma'$ with $\mathcal{D}om(\Gamma') \cap \mathcal{V}ar(\Delta) = \varnothing$ and a derivation D_1' for $\Gamma : x \Downarrow_1 \Delta : c(\overline{x_n})$ with $|D_1'| = |D_1|$ by Lemma B.5. Since Γ' cannot reference variables introduced in p_i due to the variable convention, we have $\sigma(\mathsf{drf}(\Gamma', e_i)) = \mathsf{drf}(\Gamma', \sigma(e_i))$, so that the derivation D_2 for $(\Delta \uplus \Gamma') : \sigma(e_i) \Downarrow_1 \Theta : v$ with $|D_2'| = |D_2|$ implies a derivation D_2' for $\Delta : \mathsf{drf}(\Gamma', \sigma(e_i)) \Downarrow_1 \Theta : v$ by the

induction hypothesis. We can then construct the derivation D' with $|D'| = |D|$ as

$$\frac{\Gamma : x \Downarrow_1 \Delta : c(\overline{x_n}) \quad \Delta : \sigma(\mathsf{drf}(\Gamma', e_i)) \Downarrow_1 \Theta : v}{\Gamma : \mathtt{case}\ x\ \mathtt{of}\ \{\ \overline{p_k \to \mathsf{drf}(\Gamma', e_k)}\ \} \Downarrow_1 \Theta : v}$$

(Guess) This case is analogous to the previous case with the only difference that the heap Δ is updated and no variable substitution σ is applied.

Remaining Cases For all other cases, the claim follows analogously to the base case. □

B.2.3 Pre-Partial Evaluations

We continue to show the soundness and completeness of pre-partial evaluations, based on the results for the residualizing semantics, the dereferencing operation, and the combination of expressions.

Lemma B.9 (Correctness of discomb)**.** *Let $\overline{e_n}$ be a finite sequence of expressions. Then $\Gamma : \mathsf{discomb}(\overline{e_n}) \Downarrow_1 \Delta : v$ if and only if $\Gamma : e_i \Downarrow_1 \Delta : v$ for some $i \in \{1, \ldots, n\}$.*

Proof. For $n = 0$, there neither exists $1 \leqslant i \leqslant 0$ nor does $\Gamma : \mathtt{failed} \Downarrow_1 \Delta : v$ hold. For $n > 0$, the statement $\Gamma : e_1\ ?\ \ldots\ ?\ e_n \Downarrow_1 \Delta : v$ holds if and only if $\Gamma : e_i \Downarrow_1 \Delta : v$ holds for $i \in \{1, \ldots, n\}$ by (repeated) application of rule (Or). □

Lemma B.10 (Soundness of Pre-Partial Evaluation)**.** *Let $e \to e'$ be a pre-partial evaluation of e in P. If $\Gamma : e' \Downarrow_1 \Delta : v$, then $\Gamma : e \Downarrow_1 \Delta : v$.*

Proof. We have $e' = \mathsf{discomb}(\{\mathsf{drf}(\Theta, r) \mid [] : e \Downarrow_{PE} \Theta : r\})$ by the definition of pre-partial evaluations. By Lemma B.9, $\Gamma : e' \Downarrow_1 \Delta : v$ implies a configuration $\Theta : r$ such that $\Gamma : \mathsf{drf}(\Theta, r) \Downarrow_1 \Delta : v$ and $[] : e \Downarrow_{PE} \Theta : r$ hold. Note that $\mathcal{V}ar(\Gamma) \cap \mathcal{BV}(e) = \varnothing$ due to the variable convention, so that we can also assume $\mathcal{D}om(\Theta) \cap \mathcal{V}ar(\Gamma) = \varnothing$. By Lemma B.7, the statement $\Gamma : \mathsf{drf}(\Theta, r) \Downarrow_1 \Delta : v$ then implies $(\Theta \uplus \Gamma) : r \Downarrow_1 \Delta : v$, and by Lemma B.1 $[] : e \Downarrow_{PE} \Theta : r$ and $(\Theta \uplus \Gamma) : r \Downarrow_1 \Delta : v$ imply $([] \uplus \Gamma) : e \Downarrow_1 \Delta : v$ and thus $\Gamma : e \Downarrow_1 \Delta : v$. □

Lemma B.11 (Completeness of Pre-Partial Evaluation)**.** *Let $e \to e'$ be a pre-partial evaluation of e in P. If D is a derivation for $\Gamma : e \Downarrow_1 \Delta : v$, then $e' = e'_1\ ?\ \ldots\ ?\ e'_n$ and there exists $i \in \{1, \ldots, n\}$ and a derivation D' for $\Gamma : e'_i \Downarrow_1 \Delta : v$ such that $|D'| < |D|$.*

Proof. The derivation D for $\Gamma : e \Downarrow_1 \Delta : v$ implies a derivation D_{PE} for $[] : e \Downarrow_{PE} \Theta : r$ and a derivation D'' for $(\Theta \uplus \Gamma) : r \Downarrow_1 \Delta : v$ such that $|D| = |D_{PE}| + |D''|$ by Lemma B.3. Note that $\mathcal{V}ar(\Gamma) \cap \mathcal{BV}(e) = \varnothing$ due to the variable convention, so that we can assume $\mathcal{D}om(\Theta) \cap \mathcal{V}ar(\Gamma) = \varnothing$. Since $e \to e'$ is a pre-partial evaluation, e must be partially evaluable, which implies $|D_{PE}| > 0$ and thus $|D''| < |D|$. Furthermore, the derivation D'' for $(\Theta \uplus \Gamma) : r \Downarrow_1 \Delta : v$ implies a derivation D' for $\Gamma : \mathsf{drf}(\Theta, r) \Downarrow_1 \Delta : v$

with $|D'| = |D''|$ by Lemma B.8 and thus $|D'| < |D|$. Since $e \to \mathsf{drf}(\Theta, r)$ is a resultant, then $e' = e'_1 \ ? \ \ldots \ ? \ e'_n$ with $n \geqslant 1$ and there exists $i \in \{1, \ldots, n\}$ such that $e'_i = \mathsf{drf}(\Theta, r)$ by the definition of pre-partial evaluations. □

Finally, we like to state that for a pre-partial evaluation the unbound variables can be changed to other unbound variables without affecting the characterization as a pre-partial evaluation.

Lemma B.12 (Change of Unbound Variables in a Pre-Partial Evaluation)**.** *Let σ be a variable substitution. If $e \to e'$ is a pre-partial evaluation in P, then $\sigma(e) \to \sigma(e')$ is a pre-partial evaluation in P.*

Proof. It holds that $[] : e \Downarrow_{PE} \Theta : r$ implies $[] : \sigma(e) \Downarrow_{PE} \sigma(\Theta) : \sigma(r)$ with the same derivation size since the residualizing semantics proceeds uniformly for all unbound variables. This furthermore implies that $\sigma(e)$ is partially evaluable if e is partially evaluable according to Definition 7.11, and that $\sigma(e) \to \mathsf{drf}(\sigma(\Theta), \sigma(r))$ is a resultant if $e \to \mathsf{drf}(\Theta, r)$ is a resultant. Furthermore, we have $\sigma(\mathsf{drf}(\Theta, r)) = \mathsf{drf}(\sigma(\Theta), \sigma(r))$ as well as $\sigma(\mathsf{discomb}(\overline{e_n})) = \mathsf{discomb}(\overline{\sigma(e_n)})$ by the definition of drf and discomb, and the claim follows. □

B.2.4 Closedness and Renaming

As an auxiliary lemma for the transitivity of closedness, we show that an instance $e = \sigma(e')$ is closed if the expression e' and the range of the substitution σ are closed.

Lemma B.13 (Closedness of Instance)**.** *If e is an expression and σ a substitution such that e and $\mathcal{R}an(\sigma)$ are S-closed, then $\sigma(e)$ is S-closed.*

Proof. We prove the claim by well-founded induction on the depth of e and begin with the base cases where $\mathsf{depth}(e) = 1$.

- ▷ If $e = x \in \mathcal{V}$, then we distinguish two cases. If $x \in \mathcal{D}om(\sigma)$, then $\mathsf{closed}(S, \sigma(x))$ follows from $\mathsf{closed}(S, \mathcal{R}an(\sigma))$, and if $x \notin \mathcal{D}om(\sigma)$, then $\sigma(x) = x$ is S-closed by the definition of closedness.
- ▷ If $e = c(\overline{x_k})$ with $c \in \mathcal{C}$, then the closedness of $\sigma(e) = c(\overline{\sigma(x_k)})$ follows from the closedness of $\mathcal{R}an(\sigma)$ and the closedness of variables.
- ▷ If $e = f(\overline{x_k})$ with $f \in \mathcal{F}^{(n)}$ and $k \leqslant n$, then $\mathsf{closed}(S, e)$ implies $f(\overline{y_n}) \in S$, so that S-closedness of $\sigma(e) = f(\overline{\sigma(e_k)})$ holds since variables and $\mathcal{R}an(\sigma)$ are S-closed.

We continue with the inductive cases of $\mathsf{depth}(e) > 1$ and assume as the induction hypothesis that the claim holds for all expressions e' with $\mathsf{depth}(e') < \mathsf{depth}(e)$.

- ▷ If $e = f(\overline{e_k})$ with $f \in \mathcal{F}^{(n)}$, $k < n$, and $\exists i \in \{1, \ldots, k\}$ such that $e_i \notin \mathcal{V}$, then $\mathsf{closed}(S, e)$ implies that $f(\overline{x_n}) \in S$ and $\overline{e_k}$ is S-closed. Thus, $\overline{\sigma(e_k)}$ is S-closed by the induction hypothesis, so that $\sigma(e) = f(\overline{\sigma(e_k)})$ is also S-closed.

- If $\mathsf{closed}(S, e)$ holds because e is an instance of $s \in S$, then $e = \theta(s)$ and $\mathcal{R}an(\theta)$ is S-closed. Since $s \notin \mathcal{V}$ by the definition of closedness, all expressions in $\mathcal{R}an(\theta)$ are subexpressions with a depth smaller than $\mathsf{depth}(e)$. We can then apply the induction hypothesis to conclude that $\mathsf{closed}(S, \mathcal{R}an(\theta))$ implies $\mathsf{closed}(S, \sigma(\mathcal{R}an(\theta)))$. Then $\mathsf{closed}(S, \mathcal{R}an(\sigma \circ \theta))$ also holds, and therefore $\mathsf{closed}(S, \sigma(e)) = \mathsf{closed}(S, \sigma(\theta(s)))$ holds by the induction hypothesis.

- For all other cases, the closedness of e follows from the closedness of its subexpressions, so that the claims follows from the induction hypothesis. □

Using this auxiliary lemma, we can continue to show the transitivity of closedness.

Lemma 7.19 (Transitivity of Closedness)**.** *If an expression e is S_1-closed and S_1 is S_2-closed, then e is S_2-closed.*

Proof. We prove the claim by well-founded induction on the depth of e and begin with the base cases where $\mathsf{depth}(e) = 1$.

- If $e \in \mathcal{V}$ or $e = c(\overline{x_k})$ with $c \in \mathcal{C}$, then the claim follows from the definition of closedness.

- If $e = f(\overline{x_k})$ with $f \in \mathcal{F}^{(n)}$ and $k \leqslant n$, then $\mathsf{closed}(S, e)$ implies $f(\overline{y_n}) \in S_1$. Then $f(\overline{y_n}) \in S_2$ since S_1 is S_2-closed, and thus e is S_2-closed as well.

We continue with the inductive cases of $\mathsf{depth}(e) > 1$ and assume as the induction hypothesis that the claim holds for all expressions e' with $\mathsf{depth}(e') < \mathsf{depth}(e)$.

- If $e = f(\overline{e_k})$ with $f \in \mathcal{F}^{(n)}$, $k < n$, and $\exists i \in \{1, \ldots, k\}$ such that $e_i \notin \mathcal{V}$, then S_1-closedness of e implies $f(\overline{x_n}) \in S_1$ and S_1-closedness of $\overline{e_k}$. Thus, $\overline{e_k}$ is S_2-closed by the induction hypothesis and $f(\overline{x_n})$ is S_2-closed by assumption, so that e is S_2-closed as well.

- If $\mathsf{closed}(S, e)$ holds because $e = \sigma(s_1)$ is an instance of $s_1 \in S_1$ and $\mathcal{R}an(\sigma)$ is S_1-closed, then s_1 is S_2-closed by assumption, and since $s_1 \notin \mathcal{V}$, all expressions in $\mathcal{R}an(\sigma)$ must be subexpressions of e with a smaller depth. By the induction hypothesis, $\mathsf{closed}(S_1, \mathcal{R}an(\sigma))$ then implies $\mathsf{closed}(S_2, \mathcal{R}an(\sigma))$. Furthermore, $\mathsf{closed}(S_2, s_1)$ and $\mathsf{closed}(S_2, \mathcal{R}an(\sigma))$ imply $\mathsf{closed}(S_2, \sigma(s_1)) = \mathsf{closed}(S_2, e)$ by Lemma B.13.

- For all other cases, the closedness of e follows from the closedness of its subexpressions, so that the claim follows from the induction hypothesis. □

Finally, we show the correspondence of closedness of an expression and its renaming, i. e., an expression e is closed with respect to a set of expressions E if and only if its renaming $\mathsf{ren}_\rho(e)$ is closed with respect to the renamed set $\rho(E)$.

Lemma 7.23 (Correspondence of Closedness under Renaming)**.** *Let E be a finite set of linear and partially evaluable expressions, ρ an independent renaming for E, and $E' = \rho(E)$. Then an expression e is E-closed if and only if $e' = \mathsf{ren}_\rho(e)$ is E'-closed.*

Proof. We first show that $\mathsf{closed}(E, e)$ implies $\mathsf{closed}(E', e')$ by well-founded induction on the depth of e and begin with the base cases where $\mathsf{depth}(e) = 1$.

- ▷ If $e \in \mathcal{V}$ or $e = c(\overline{x_k})$ with $c \in \mathcal{C}$, then $e' = e$ and the claim follows from the definition of closedness.
- ▷ If $e = f(\overline{x_k})$ with $f \in \mathcal{F}^{(n)}$ and $k \leqslant n$, then $f(\overline{y_n}) \in E$ and $e' = f'(\overline{x_k})$ where $f'(\overline{y_n}) = \rho(f(\overline{y_n}))$. Since $f'(\overline{y_n}) \in E'$, then $e' = f'(\overline{x_k})$ is E'-closed.

We continue with the inductive cases of $\mathsf{depth}(e) > 1$ and assume as the induction hypothesis that the claim holds for all expressions e'' such that $\mathsf{depth}(e'') < \mathsf{depth}(e)$.

- ▷ If $e = f(\overline{e_k})$ with $f \in \mathcal{F}^{(n)}$, $k < n$, and $\exists i \in \{1, \ldots, k\}$ such that $e_i \notin \mathcal{V}$, then $e' = f'(\overline{\mathsf{ren}_\rho(e_k)})$ where $f'(\overline{x_n}) = \rho(f(\overline{x_n}))$ and $f(\overline{x_n}) \in E$. Furthermore, E-closedness of e implies E-closedness of $\overline{e_k}$, so that $\overline{\mathsf{ren}_\rho(e_k)}$ is E'-closed by the induction hypothesis. Since $f'(\overline{x_n}) \in E'$, then $e' = f'(\overline{\mathsf{ren}_\rho(e_k)})$ is also E'-closed.
- ▷ If $e = \sigma(e'')$ is an instance of $e'' \in E$ with $\sigma = \{\overline{x_n \mapsto e_n}\}$ and $\mathsf{closed}(E, \mathcal{R}an(\sigma))$, then $e' = \sigma'(\rho(e'')) = f_{e''}(\overline{\mathsf{ren}_\rho(e_n)})$ for $\rho(e'') = f_{e''}(\overline{x_n})$ and $\sigma' = \{\overline{x_n \mapsto \mathsf{ren}_\rho(e_n)}\}$. Since $e'' \notin \mathcal{V}$, the expressions $\overline{e_n}$ are smaller than e, and since they are E-closed by assumption, then $\mathcal{R}an(\sigma')$ is E'-closed by the induction hypothesis. Furthermore, $f_{e''}(\overline{x_n}) \in E'$, so that $e' = f_{e''}(\overline{\mathsf{ren}_\rho(e_n)})$ is also E'-closed.
- ▷ For all other cases it holds that $e' = \mathsf{ren}'_\rho(e)$, so that the claim follows from the induction hypothesis.

We then show that $\mathsf{closed}(E', e')$ implies $\mathsf{closed}(E, e)$ by well-founded induction on the depth of e' and begin with the base cases where $\mathsf{depth}(e') = 1$.

- ▷ If $e' \in \mathcal{V}$ or $e' = c(\overline{x_k})$ with $c \in \mathcal{C}$, then $e = e'$ and the claim follows from the definition of closedness.
- ▷ If $e' = f'(\overline{x_k})$ with $f' \in \mathcal{F}^{(n)}$ and $k \leqslant n$, then $f'(\overline{y_n}) \in E'$ and $e = f(\overline{x_k})$ where $f'(\overline{y_n}) = \rho(f(\overline{y_n}))$. Then $f(\overline{y_n}) \in E$, so that $e = f(\overline{x_k})$ is E-closed.

We continue with the inductive cases of $\mathsf{depth}(e') > 1$ and assume as the induction hypothesis that the claim holds for all expressions e'' such that $\mathsf{depth}(e'') < \mathsf{depth}(e')$.

- ▷ If $e' = f'(\overline{e'_k})$ with $f' \in \mathcal{F}^{(n)}$, $k < n$, and $\exists i \in \{1, \ldots, k\}$ such that $e'_i \notin \mathcal{V}$, then $\overline{e'_k}$ is E'-closed and $e = f(\overline{e_k})$ with $f \in \mathcal{F}^{(n)}$ and $e'_i = \mathsf{ren}_\rho(e_i)$ for all $i \in \{1, \ldots, k\}$ by the definition of renaming, where $f'(\overline{x_n}) = \rho(f(\overline{x_n}))$ and $f(\overline{x_n}) \in E$. Furthermore, $\overline{e_k}$ is E-closed by the induction hypothesis, so that e is also E-closed.
- ▷ If $e' = f'(\overline{e'_n})$ with $f' \in \mathcal{F}^{(n)}$, then $e = \sigma(e'')$ must be an instance of $e'' \in E$ with $\rho(e'') = f'(\overline{x_n})$ and $\mathsf{closed}(E, \mathcal{R}an(\sigma))$, so that e must be E-closed by Lemma B.13.

▷ For all other cases, it holds that $e' = \mathsf{ren}_\rho(e) = \mathsf{ren}'_\rho(e)$, so that the claim follows from the induction hypothesis. □

Furthermore, we show that the process of renaming does not affect the abstract value semantics besides the renaming of partial function applications. To show this result we need two auxiliary lemmata, where the first one states that the renaming of an expression is a variable if and only if the expression is a variable.

Lemma B.14 (Variable Preservation of Renaming)**.** $\mathsf{ren}_\rho(e) \in \mathcal{V}$ *if and only if* $e \in \mathcal{V}$.

Proof. By the definition of renaming. □

The second auxiliary lemma then states that the renaming does not affect the recursive heap lookup operation.

Lemma B.15 (Heap Lookup under Renaming)**.** *Let E be a finite set of linear and partially evaluable expressions, ρ an independent renaming for E, and Γ a heap such that $\mathsf{ren}_\rho(\Gamma)$ is E'-closed, where $E' = \rho(E)$. Then $\mathsf{ren}_\rho(\Gamma^*(x)) = \mathsf{ren}_\rho(\Gamma)^*(x)$ for every variable x.*

Proof. By well-founded induction on the complexity of the heap Γ. The base case of $\mathcal{M}_\Gamma = \varnothing$ implies $\Gamma = [\,]$, for which we have $\mathsf{ren}_\rho([\,]^*(x)) = \mathsf{ren}_\rho(x) = x$ and $\mathsf{ren}_\rho([\,])^*(x) = [\,]^*(x) = x$. We continue with the inductive cases where we assume that the claim holds for all heaps Γ' with $\mathcal{M}_{\Gamma'} <_{\mathsf{mul}} \mathcal{M}_\Gamma$.

▷ If $x \notin \mathcal{D}om(\Gamma)$ or $\Gamma(x) \in \{\mathsf{free}, \blacksquare, x\}$, then $\mathsf{ren}_\rho(\Gamma^*(x)) = \mathsf{ren}_\rho(x) = x = \mathsf{ren}_\rho(\Gamma)^*(x)$.

▷ If $\Gamma = \Gamma'[x \mapsto y]$ with $x \neq y$, then $\mathsf{ren}_\rho(\Gamma'[x \mapsto y]^*(x)) = \mathsf{ren}_\rho(\Gamma'^*(y))$ and $\mathsf{ren}_\rho(\Gamma'[x \mapsto y])^*(x) = \mathsf{ren}_\rho(\Gamma')[x \mapsto y]^*(x) = \mathsf{ren}_\rho(\Gamma')^*(y)$, which are equal by the induction hypothesis.

▷ If $\Gamma = \Gamma'[x \mapsto \phi(\overline{e_k})]$ with $\phi \in \mathcal{C}$ or $k < \mathsf{arity}(\phi)$, then

$$\begin{aligned} & \mathsf{ren}_\rho(\Gamma'[x \mapsto \phi(\overline{e_k})]^*(x)) \\ = \; & \mathsf{ren}_\rho(\phi(\Gamma'[\overline{y_l \mapsto e'_l}]^*(x_k))) \\ = \; & \phi'(\mathsf{ren}_\rho(\Gamma'[\overline{y_l \mapsto e'_l}]^*(x_k))) \end{aligned}$$

where $(\overline{y_l}, \overline{e'_l}, \overline{x_k}) = \mathsf{splitArgs}(\overline{e_k})$ and $\mathsf{ren}_\rho(\phi(\overline{x_k})) = \phi'(\overline{x_k})$, and

$$\begin{aligned} & \mathsf{ren}_\rho(\Gamma'[x \mapsto \phi(\overline{e_k})])^*(x) \\ = \; & \mathsf{ren}_\rho(\Gamma')[x \mapsto \phi'(\overline{\mathsf{ren}_\rho(e_k)})]^*(x) \\ = \; & \phi'(\overline{\mathsf{ren}_\rho(\Gamma')[\overline{y_l \mapsto \mathsf{ren}_\rho(e'_l)}]^*(x_k)}) \\ = \; & \phi'(\mathsf{ren}_\rho(\Gamma')[\overline{y_l \mapsto e'_l}])^*(x_k)) \end{aligned}$$

according to Lemma B.14, and both results are equal by the induction hypothesis.

▷ For all other cases, we have $\Gamma = \Gamma'[x \mapsto e]$ where e is either a function application, a `let` expression, a non-deterministic choice, or a `case` expression.

- ▷ If $e = \sigma(e')$ is an instance of $e' \in E$ with $\sigma = \{\overline{x_n \mapsto e_n}\}$, then we have $\mathsf{ren}_\rho(\Gamma'[x \mapsto e]^*(x)) = \mathsf{ren}_\rho(x) = x$ by the definition of heap lookup, as well as $\mathsf{ren}_\rho(\Gamma'[x \mapsto e])^*(x) = \mathsf{ren}_\rho(\Gamma')[x \mapsto f'(\overline{\mathsf{ren}_\rho(e_n)})]^*(x) = x$ for $f'(\overline{x_n}) = \rho(e')$.
- ▷ If e is not an instance of any $e' \in E$, then $\mathsf{ren}_\rho(\Gamma'[x \mapsto e]^*(x)) = \mathsf{ren}_\rho(x) = x$ for the left-hand side, and $\mathsf{ren}_\rho(\Gamma'[x \mapsto e])^*(x) = \mathsf{ren}_\rho(\Gamma')[x \mapsto \mathsf{ren}'_\rho(e)]^*(x) = x$ for the right-hand side. □

This lemma can easily be extended to show that the recursive renaming can be applied before or after computation of the abstract value semantics of a configuration.

Lemma 7.32 (Abstract Value Semantics under Renaming)**.** *Let E be a finite set of linear and partially evaluable expressions, ρ an independent renaming for E, and $\Gamma : v$ a configuration such that v is a value and $\mathsf{ren}_\rho(\Gamma : v)$ is E'-closed, where $E' = \rho(E)$. Then $\mathsf{ren}_\rho(\llbracket \Gamma : v \rrbracket) = \llbracket \mathsf{ren}_\rho(\Gamma : v) \rrbracket$.*

Proof. We distinguish two cases for v.

- ▷ If $v \in \mathcal{V}$, then $\mathsf{ren}_\rho(\llbracket \Gamma : v \rrbracket) = \mathsf{ren}_\rho(v) = v$ and $\llbracket \mathsf{ren}_\rho(\Gamma : v) \rrbracket = \llbracket \mathsf{ren}_\rho(\Gamma) : v \rrbracket = v$.
- ▷ If $v = \phi(\overline{x_k})$ with $\phi \in \mathcal{C}$ or $k < \mathsf{arity}(\phi)$, then $\mathsf{ren}_\rho(\llbracket \Gamma : \phi(\overline{x_k}) \rrbracket) = \mathsf{ren}_\rho(\phi(\overline{\Gamma^*(x_k)})) = \phi'(\overline{\mathsf{ren}_\rho(\Gamma^*(x_k))})$ with $\mathsf{ren}_\rho(\phi(\overline{x_k})) = \phi'(\overline{x_k})$ for the left-hand side, as well as $\llbracket \mathsf{ren}_\rho(\Gamma : \phi(\overline{x_k})) \rrbracket = \llbracket \mathsf{ren}_\rho(\Gamma) : \phi'(\overline{x_k}) \rrbracket = \phi'(\overline{\mathsf{ren}_\rho(\Gamma)^*(x_k)})$ for the right-hand side, and their equivalence follows from Lemma B.15. □

B.2.5 Extensions of Configurations

Since the process of renaming may introduce extensions of a configuration, we need to show that two in-configurations that are equivalent up to multiple extension can be evaluated to out-configurations that are equivalent up to multiple extension. We prove the soundness and completeness results individually and combine them afterwards.

Soundness

To show the soundness of equivalence up to multiple extension, we start with the following lemma stating the soundness of a single extension.

Lemma B.16. *Let $\Gamma' : e'$ and $\Gamma : e$ be two configurations such that $\Gamma' : e' >_x \Gamma : e$. If $\Gamma' : e' \Downarrow_1 \Delta' : v$, then there exists a heap Δ such that $\Gamma : e \Downarrow_1 \Delta : v$ and $\Delta' : v \geqslant_x \Delta : v$.*

Proof. By structural induction on the derivation of the premise. Note that we will simply speak about the occurrence of a variable in an expression e, heap Γ, or configuration $\Gamma : e$ if we refer to its occurrence in $\mathcal{V}ar_M(e)$, $\mathcal{V}ar_M(\Gamma)$, or $\mathcal{V}ar_M(\Gamma : e)$.

(Value) For the base case of rule (Value) we have to show that the statement $\Gamma[x \mapsto e''] : v \Downarrow_1 \Gamma[x \mapsto e''] : v$ and $\Gamma[x \mapsto e''] : v >_x \sigma(\Gamma) : \sigma(v)$ for $\sigma = \{x \mapsto e''\}$ imply $\sigma(\Gamma) : \sigma(v) \Downarrow_1 \Delta : v$ and $\Gamma[x \mapsto e''] : v \geqslant_x \Delta : v$, where $v = \phi(\overline{x_k})$ with $\phi \in \mathcal{C}$ or $k < \mathsf{arity}(\phi)$, or $v \in \mathcal{V}$ with $\Gamma[x \mapsto e''](v) = \text{free}$.

1. If $v \in \mathcal{V}$ or $v = \phi(\overline{x_k})$ with $x \notin \overline{x_k}$, then $\sigma(v) = v$ and $\sigma(\Gamma) : v \Downarrow_1 \sigma(\Gamma) : v$ holds by rule (Value). Furthermore, $\Gamma[x \mapsto e''] : v \geqslant_x \sigma(\Gamma) : v$ follows from $\Gamma[x \mapsto e''] : v >_x \sigma(\Gamma) : \sigma(v)$ and $\sigma(v) = v$.
2. If $v = \phi(\overline{x_k})$ and $x = x_i$ for any $i \in \{1, \ldots, k\}$, then x does not occur in Γ, so that $\sigma(\Gamma) = \Gamma$ and $\sigma(v) = \phi(x_1, \ldots, x_{i-1}, e'', x_{i+1}, \ldots, x_k)$. We can thus construct the following derivation using rules (Flatten) and (Value)
$$\dfrac{\dfrac{}{\Gamma[x \mapsto e''] : \phi(\overline{x_k}) \Downarrow_1 \Gamma[x \mapsto e''] : \phi(\overline{x_k})}}{\begin{array}{rc} & \Gamma : \phi(x_1, \ldots, x_{i-1}, e'', x_{i+1}, \ldots, x_k) \Downarrow_1 \Gamma[x \mapsto e''] : \phi(\overline{x_k}) \\ = & \sigma(\Gamma) : \sigma(\phi(\overline{x_k})) \Downarrow_1 \Gamma[x \mapsto e''] : \phi(\overline{x_k}) \end{array}}$$
and $\Gamma[x \mapsto e''] : \phi(\overline{x_k}) \geqslant_x \Gamma[x \mapsto e''] : \phi(\overline{x_k})$ directly holds.

We continue with the inductive cases and assume as the induction hypothesis that the claim holds for the premises of the assumed derivation. We first consider the general case of $\Gamma[x \mapsto e''] : e \Downarrow_1 \Delta' : v$ such that x does not occur in $\Gamma : e$.

(General Case) If $\Gamma[x \mapsto e''] : e \Downarrow_1 \Delta' : v$ and x does not occur in $\Gamma : e$, we have to show that $\Gamma[x \mapsto e''] : e \Downarrow_1 \Delta' : v$ implies $\Gamma : e \Downarrow_1 \Delta : v$ such that $\Delta' : v \geqslant_x \Delta : v$. In this case $\Gamma[x \mapsto e''] : e \Downarrow_1 \Delta' : v$ and $x \notin \mathcal{V}ar(\Gamma : e)$ implies $\Delta' = \Delta[x \mapsto e'']$ with $x \notin \mathcal{V}ar(\Delta : v)$ and $\Gamma : e \Downarrow_1 \Delta : v$ by Lemma B.5, and $\Delta[x \mapsto e''] : v \geqslant_x \Delta : v$ follows from $x \notin \mathcal{V}ar(\Delta : v)$.

For the remaining cases we can then assume that x occurs once in $\Gamma : e$.

(VarExp) We assume a derivation for $\Gamma[x \mapsto e''][y \mapsto e] : y \Downarrow_1 \Delta'[y \mapsto v] : v$ where $e \notin \{\text{free}, \blacksquare\}$ and we distinguish two cases for y.

1. If $y = x$, then $e = e''$ and we have to show that $\Gamma[x \mapsto e''] : x \Downarrow_1 \Delta'[x \mapsto v] : v$ and $\Gamma[x \mapsto e''] : x >_x \sigma(\Gamma) : \sigma(x)$ for $\sigma = \{x \mapsto e''\}$ imply $\sigma(\Gamma) : \sigma(x) \Downarrow_1 \Delta : v$ and $\Delta'[x \mapsto v] : v \geqslant_x \Delta : v$. Since x occurs once in $\Gamma : x$ it cannot occur in Γ, and thus $\sigma(\Gamma) = \Gamma$ and $\sigma(x) = e''$. Furthermore, the assumed derivation holds by the premise $\Gamma[x \mapsto \blacksquare] : e'' \Downarrow_1 \Delta' : v$, where $\Delta' = \Delta[x \mapsto \blacksquare]$ according to Lemma 5.15. Since x neither occurs in Γ nor in e'', this implies $\Gamma : e'' \Downarrow_1 \Delta : v$ and $x \notin \mathcal{V}ar(\Delta : v)$ by Lemma B.5, so that $\sigma(\Gamma) : \sigma(x) \Downarrow_1 \Delta : v$ as well as $\Delta[x \mapsto v] : v \geqslant_x \Delta : v$ hold.
2. If $y \neq x$, then we have to show that $\Gamma[x \mapsto e''][y \mapsto e] : y \Downarrow_1 \Delta'[y \mapsto v] : v$ and $\Gamma[x \mapsto e''][y \mapsto e] : y >_x \sigma(\Gamma[y \mapsto e]) : y$ for $\sigma = \{x \mapsto e''\}$ imply $\sigma(\Gamma[y \mapsto e]) : y \Downarrow_1 \Delta : v$ and $\Delta[y \mapsto v] : v \geqslant_x \Delta : v$. The induction hypothesis states that

$\Gamma[x \mapsto e''][y \mapsto \blacksquare] : e \Downarrow_1 \Delta' : v$ and $\Gamma[x \mapsto e''][y \mapsto \blacksquare] : e >_x \sigma(\Gamma)[y \mapsto \blacksquare] : \sigma(e)$ imply $\sigma(\Gamma)[y \mapsto \blacksquare] : \sigma(e) \Downarrow_1 \Delta'' : v$ and $\Delta' : v \geqslant_x \Delta'' : v$. Since $\Gamma[x \mapsto e''][y \mapsto e] : y >_x \sigma(\Gamma[y \mapsto e]) : y$ and $y \neq x$ imply $\Gamma[x \mapsto e''][y \mapsto \blacksquare] : e >_x \sigma(\Gamma)[y \mapsto \blacksquare] : \sigma(e)$, we can apply the induction hypothesis to construct the derivation

$$\frac{\sigma(\Gamma)[y \mapsto \blacksquare] : \sigma(e) \Downarrow_1 \Delta'' : v}{\sigma(\Gamma)[y \mapsto \sigma(e)] : y \Downarrow_1 \Delta''[y \mapsto v] : v}$$

where $\Delta'[y \mapsto v] : v \geqslant_x \Delta''[y \mapsto v] : v$ follows from $\Delta' : v \geqslant_x \Delta'' : v$.

(Flatten) We have to show that $\Gamma[x \mapsto e''] : \phi(\overline{e_k}) \Downarrow_1 \Delta' : v$ and $\Gamma[x \mapsto e''] : \phi(\overline{e_k}) >_x \sigma(\Gamma) : \sigma(\phi(\overline{e_k}))$ for $\sigma = \{x \mapsto e''\}$ imply $\sigma(\Gamma) : \sigma(\phi(\overline{e_k})) \Downarrow_1 \Delta : v$ and $\Delta' : v \geqslant_x \Delta : v$, where $\exists i \in \{1, \ldots, k\}$ such that $e_i \notin \mathcal{V}$ and $(\overline{y_l}, \overline{e_l'}, \overline{x_k}) = \mathsf{splitArgs}(\overline{e_k})$. We distinguish two cases based on the occurrence of x in $\Gamma : \phi(\overline{e_k})$.

1. If $x = e_i$ for any $i \in \{1, \ldots, k\}$, then $\sigma(\phi(\overline{e_k})) = \phi(e_1, \ldots, e_{i-1}, e'', e_{i+1}, \ldots, e_k)$ and $\sigma(\Gamma) = \Gamma$. By application of rule (Flatten), we can then construct the derivation
$$\begin{array}{c} \Gamma[x \mapsto e'', \overline{y_l \mapsto e_l'}] : \phi(\overline{x_k}) \Downarrow_1 \Delta' : v \\ \hline \Gamma : \phi(e_1, \ldots, e_{i-1}, e'', e_{i+1}, \ldots, e_k) \Downarrow_1 \Delta' : v \\ = \quad \sigma(\Gamma) : \sigma(\phi(\overline{e_k})) \Downarrow_1 \Delta' : v \end{array}$$
where the premise is implied by the assumed statement $\Gamma[x \mapsto e''] : \phi(\overline{e_k}) \Downarrow_1 \Delta' : v$ and we trivially have $\Delta' : v \geqslant_x \Delta' : v$.

2. For all other cases, we have that $\Gamma[x \mapsto e''] : \phi(\overline{e_k}) >_x \sigma(\Gamma) : \sigma(\phi(\overline{e_k}))$ implies $\Gamma[x \mapsto e''][\overline{y_l \mapsto e_l'}] : \phi(\overline{x_k}) >_x \sigma(\Gamma[\overline{y_l \mapsto e_l'}]) : \sigma(\phi(\overline{x_k}))$. Thus, we can apply the induction hypothesis stating that $\Gamma[x \mapsto e''][\overline{y_l \mapsto e_l'}] : \phi(\overline{x_k}) \Downarrow_1 \Delta' : v$ and $\Gamma[x \mapsto e''][\overline{y_l \mapsto e_l'}] : \phi(\overline{x_k}) >_x \sigma(\Gamma[\overline{y_l \mapsto e_l'}]) : \sigma(\phi(\overline{x_k}))$ imply $\sigma(\Gamma[\overline{y_l \mapsto e_l'}]) : \sigma(\phi(\overline{x_k})) \Downarrow_1 \Delta : v$ and $\Delta' : v \geqslant_x \Delta : v$. By rule (Flatten), we can then construct the derivation
$$\begin{array}{c} \sigma(\Gamma[\overline{y_l \mapsto e_l'}]) : \sigma(\phi(\overline{x_k})) \Downarrow_1 \Delta : v \\ = \quad \sigma(\Gamma)[\overline{y_l \mapsto \sigma(e_l')}] : \phi(\overline{x_k}) \Downarrow_1 \Delta : v \\ \hline \sigma(\Gamma) : \phi(\overline{\sigma(e_k)}) \Downarrow_1 \Delta : v \\ = \quad \sigma(\Gamma) : \sigma(\phi(\overline{e_k})) \Downarrow_1 \Delta : v \end{array}$$
and $\Delta' : v \geqslant_x \Delta : v$ follows from the induction hypothesis.

(FunEval) We have to show that $\Gamma[x \mapsto e''] : f(\overline{x_n}) \Downarrow_1 \Delta' : v$ and $\Gamma[x \mapsto e''] : f(\overline{x_n}) >_x \sigma(\Gamma) : \sigma(f(\overline{x_n}))$ for $\sigma = \{x \mapsto e''\}$ imply $\sigma(\Gamma) : \sigma(f(\overline{x_n})) \Downarrow_1 \Delta : v$ and $\Delta' : v \geqslant_x \Delta : v$, where $f(\overline{x_n}) = e$ is a variable instance of a rule in P. We distinguish two cases based on the occurrence of x in $\Gamma : f(\overline{x_n})$.

1. If $x = x_i$ for any $i \in \{1, \ldots, n\}$, then $\sigma(f(\overline{x_n})) = f(x_1, \ldots, x_{i-1}, e'', x_{i+1}, \ldots, x_n)$ and $\sigma(\Gamma) = \Gamma$. By application of rule (Flatten), we can then construct the

derivation
$$\frac{\Gamma[x \mapsto e''] : f(\overline{x_n}) \Downarrow_1 \Delta' : v}{\begin{array}{rc}\Gamma : f(x_1, \ldots, x_{i-1}, e'', x_{i+1}, \ldots, x_n) & \Downarrow_1 \Delta' : v \\ = \quad \sigma(\Gamma) : \sigma(f(\overline{x_n})) & \Downarrow_1 \Delta' : v\end{array}}$$
where the premise holds by assumption and $\Delta' : v \geqslant_x \Delta' : v$ by equality.

2. Otherwise, x occurs in Γ and not in $\overline{x_n}$. Then x cannot occur in e, so that $\sigma(e) = e$ and $\Gamma[x \mapsto e''] : e >_x \sigma(\Gamma) : \sigma(e)$. In this case, the induction hypothesis states that $\Gamma[x \mapsto e''] : e \Downarrow_1 \Delta' : v$ and $\Gamma[x \mapsto e''] : e >_x \sigma(\Gamma) : \sigma(e)$ imply $\sigma(\Gamma) : \sigma(e) \Downarrow_1 \Delta : v$ and $\Delta' : v \geqslant_x \Delta : v$. We can then construct the derivation
$$\frac{\begin{array}{rc}\sigma(\Gamma) : \sigma(e) & \Downarrow_1 \Delta : v \\ = \quad \sigma(\Gamma) : e & \Downarrow_1 \Delta : v\end{array}}{\begin{array}{rc}\sigma(\Gamma) : f(\overline{x_n}) & \Downarrow_1 \Delta : v \\ = \quad \sigma(\Gamma) : \sigma(f(\overline{x_n})) & \Downarrow_1 \Delta : v\end{array}}$$
and $\Delta' : v \geqslant_x \Delta : v$ follows from the induction hypothesis.

(Let) We have to show that $\Gamma[x \mapsto e''] : \texttt{let } \{\overline{x_k = e_k}\} \texttt{ in } e \Downarrow_1 \Delta' : v$ and $\Gamma[x \mapsto e''] : \texttt{let } \{\overline{x_k = e_k}\} \texttt{ in } e >_x \sigma(\Gamma) : \sigma(\texttt{let } \{\overline{x_k = e_k}\} \texttt{ in } e)$ for $\sigma = \{x \mapsto e''\}$ imply $\sigma(\Gamma) : \sigma(\texttt{let } \{\overline{x_k = e_k}\} \texttt{ in } e) \Downarrow_1 \Delta : v$ and $\Delta' : v \geqslant_x \Delta : v$. Then it holds $\Gamma[x \mapsto e''][\overline{x_k \mapsto e_k}] : e >_x \sigma(\Gamma[\overline{x_k \mapsto e_k}]) : \sigma(e)$, and the induction hypothesis states that $\Gamma[x \mapsto e''][\overline{x_k \mapsto e_k}] : e \Downarrow_1 \Delta' : v$ and $\Gamma[x \mapsto e''][\overline{x_k \mapsto e_k}] : e >_x \sigma(\Gamma[\overline{x_k \mapsto e_k}]) : \sigma(e)$ imply $\sigma(\Gamma[\overline{x_k \mapsto e_k}]) : \sigma(e) \Downarrow_1 \Delta : v$ and $\Delta' : v \geqslant_x \Delta : v$. We can then construct the derivation
$$\frac{\begin{array}{rc}\sigma(\Gamma[\overline{x_k \mapsto e_k}]) : \sigma(e) & \Downarrow_1 \Delta : v \\ = \quad \sigma(\Gamma)[\overline{x_k \mapsto \sigma(e_k)}]) : \sigma(e) & \Downarrow_1 \Delta : v\end{array}}{\begin{array}{rc}\sigma(\Gamma) : \texttt{let } \{\overline{x_k = \sigma(e_k)}\} \texttt{ in } \sigma(e) & \Downarrow_1 \Delta : v \\ = \quad \sigma(\Gamma) : \sigma(\texttt{let } \{\overline{x_k = e_k}\} \texttt{ in } e) & \Downarrow_1 \Delta : v\end{array}}$$
and $\Delta' : v \geqslant_x \Delta : v$ follows from the induction hypothesis.

(Or), (Free) These cases follow with the same reasoning as for rule (Let).

(Select) In this case we have to show that $\Gamma[x \mapsto e''] : \texttt{case } e \texttt{ of } \{ \overline{p_k \to e_k} \} \Downarrow_1 \Theta' : v$ and $\Gamma[x \mapsto e''] : \texttt{case } e \texttt{ of } \{ \overline{p_k \to e_k} \} >_x \sigma(\Gamma) : \sigma(\texttt{case } e \texttt{ of } \{ \overline{p_k \to e_k} \})$ for $\sigma = \{x \mapsto e''\}$ imply $\sigma(\Gamma) : \sigma(\texttt{case } e \texttt{ of } \{ \overline{p_k \to e_k} \}) \Downarrow_1 \Theta : v$ and $\Theta' : v \geqslant_x \Theta : v$. Furthermore, we have $\Gamma[x \mapsto e''] : e \Downarrow_1 \Delta' : c(\overline{x_n})$ and $\Delta' : \theta(e_i) \Downarrow_1 \Theta' : v$, where $c(\overline{x_n}) = \theta(p_i)$ for $i \in \{1, \ldots, k\}$. We distinguish two cases based on the occurrence of x in $\Gamma : \texttt{case } e \texttt{ of } \{ \overline{p_k \to e_k} \}$.

1. If x occurs in Γ or e, we have $\Gamma[x \mapsto e''] : e >_x \sigma(\Gamma) : \sigma(e)$ and can apply the induction hypothesis stating that $\Gamma[x \mapsto e''] : e \Downarrow_1 \Delta' : c(\overline{x_n})$ and $\Gamma[x \mapsto e''] : e >_x \sigma(\Gamma) : \sigma(e)$ imply $\sigma(\Gamma) : \sigma(e) \Downarrow_1 \Delta : c(\overline{x_n})$ and $\Delta' : c(\overline{x_n}) \geqslant_x \Delta : c(\overline{x_n})$.

(a) If $\Delta = \Delta'$, we can construct the derivation

$$\frac{\sigma(\Gamma) : \sigma(e) \Downarrow_1 \Delta' : c(\overline{x_n}) \quad \Delta' : \theta(e_i) \Downarrow_1 \Theta' : v}{\begin{array}{rl} & \sigma(\Gamma) : \texttt{case}\ \sigma(e)\ \texttt{of}\ \{\ \overline{p_k \to e_k}\ \} \Downarrow_1 \Theta' : v \\ = & \sigma(\Gamma) : \sigma(\texttt{case}\ e\ \texttt{of}\ \{\ \overline{p_k \to e_k}\ \}) \Downarrow_1 \Theta' : v \end{array}}$$

where $\Theta' : v \geqslant_x \Theta' : v$ directly holds.

(b) If $\Delta' : c(\overline{x_n}) >_x \Delta : c(\overline{x_n})$, then $\Delta' : \theta(e_i) >_x \Delta : \theta(e_i)$ and we can apply the induction hypothesis stating that $\Delta' : \theta(e_i) \Downarrow_1 \Theta' : v$ and $\Delta' : \theta(e_i) >_x \Delta : \theta(e_i)$ imply $\Delta : \theta(e_i) \Downarrow_1 \Theta : v$ and $\Theta' : v \geqslant_x \Theta : v$. Thus, we can construct the derivation

$$\frac{\sigma(\Gamma) : \sigma(e) \Downarrow_1 \Delta : c(\overline{x_n}) \quad \Delta : \theta(e_i) \Downarrow_1 \Theta : v}{\begin{array}{rl} & \sigma(\Gamma) : \texttt{case}\ \sigma(e)\ \texttt{of}\ \{\ \overline{p_k \to e_k}\ \} \Downarrow_1 \Theta : v \\ = & \sigma(\Gamma) : \sigma(\texttt{case}\ e\ \texttt{of}\ \{\ \overline{p_k \to e_k}\ \}) \Downarrow_1 \Theta : v \end{array}}$$

and $\Theta' : v \geqslant_x \Theta : v$ follows from the induction hypothesis.

2. If x does not occur in Γ or e, then $\Gamma[x \mapsto e''] : e \Downarrow_1 \Delta' : c(\overline{x_n})$ implies $\Delta' = \Delta[x \mapsto e'']$ and $\Gamma : e \Downarrow_1 \Delta : c(\overline{x_n})$ with $x \notin \mathcal{V}ar(\Delta : c(\overline{x_n}))$ by Lemma B.5. Then we have $\Delta' : \theta(e_i) >_x \Delta : \sigma(\theta(e_i))$ and can apply the induction hypothesis stating that $\Delta' : \theta(e_i) \Downarrow_1 \Theta' : v$ and $\Delta' : \theta(e_i) >_x \Delta : \sigma(\theta(e_i))$ imply $\Delta : \sigma(\theta(e_i)) \Downarrow_1 \Theta : v$ and $\Theta' : v \geqslant_x \Theta : v$. Furthermore, we have $\mathcal{D}om(\sigma) \cap \mathcal{D}om(\theta) = \varnothing$, $\mathcal{D}om(\sigma) \cap \mathcal{R}an(\theta) = \varnothing$, and $\mathcal{D}om(\theta) \cap \mathcal{R}an(\sigma) = \varnothing$, so that $\sigma(\theta(e_i)) = \theta(\sigma(e_i))$. Hence, we can construct the derivation

$$\frac{\Gamma : e \Downarrow_1 \Delta : c(\overline{x_n}) \quad \Delta : \theta(\sigma(e_i)) \Downarrow_1 \Theta : v}{\begin{array}{rl} & \Gamma : \texttt{case}\ e\ \texttt{of}\ \{\ \overline{p_k \to \sigma(e_k)}\ \}) \Downarrow_1 \Theta : v \\ = & \sigma(\Gamma) : \sigma(\texttt{case}\ e\ \texttt{of}\ \{\ \overline{p_k \to e_k}\ \}) \Downarrow_1 \Theta : v \end{array}}$$

where $\Theta' : v \geqslant_x \Theta : v$ follows from the induction hypothesis.

(Guess) This case follows with the same reasoning as for rule (Select) with the simplification that no variable substitution θ has to be considered. □

We can easily extend the soundness of a single extension to soundness of equality up to single extension.

Lemma B.17. *Let $\Gamma' : e'$ and $\Gamma : e$ be two configurations such that $\Gamma' : e' \geqslant_x \Gamma : e$. If $\Gamma' : e' \Downarrow_1 \Delta' : v$, then there exists a heap Δ such that $\Gamma : e \Downarrow_1 \Delta : v$ and $\Delta' : v \geqslant_x \Delta : v$.*

Proof. If $\Gamma' : e' = \Gamma : e$, then the claim holds for $\Delta = \Delta'$, and for $\Gamma' : e' >_x \Gamma : e$ the claim follows from Lemma B.16. □

We can further generalize this lemma to the soundness of equality up to multiple extension.

Lemma B.18 (Soundness of Equality Up To Multiple Extension)**.** *Let $\Gamma' : e'$ and $\Gamma : e$ be two configurations such that $\Gamma' : e' \geqslant_{\overline{x_n}} \Gamma : e$. If $\Gamma' : e' \Downarrow_1 \Delta' : v$, then there exists a heap Δ such that $\Gamma : e \Downarrow_1 \Delta : v$ and $\Delta' : v \geqslant_{\overline{x_n}} \Delta : v$.*

Proof. By natural induction on the length n of the extension. If $n = 0$, then $\Gamma' : e' = \Gamma : e$ and the claim holds for $\Delta' = \Delta$. We continue with the inductive case of $n > 0$ and assume as the induction hypothesis that the claim holds for n. If $\Gamma' : e' \geqslant_{x \cdot \overline{x_n}} \Gamma : e$, then there must exist a configuration $\Gamma'' : e''$ such that $\Gamma' : e' \geqslant_x \Gamma'' : e''$ and $\Gamma'' : e'' \geqslant_{\overline{x_n}} \Gamma : e$. Thus, $\Gamma' : e' \Downarrow_1 \Delta' : v$ and $\Gamma' : e' \geqslant_x \Gamma'' : e''$ imply $\Gamma'' : e'' \Downarrow_1 \Delta'' : v$ and $\Delta' : v \geqslant_x \Delta'' : v$ by Lemma B.17. Furthermore, $\Gamma'' : e'' \Downarrow_1 \Delta'' : v$ and $\Gamma'' : e'' \geqslant_{\overline{x_n}} \Gamma : e$ imply $\Gamma : e \Downarrow_1 \Delta : v$ and $\Delta'' : v \geqslant_{\overline{x_n}} \Delta : v$ by the induction hypothesis. Finally, $\Delta' : v \geqslant_{x \cdot \overline{x_n}} \Delta : v$ follows from $\Delta' : v \geqslant_x \Delta'' : v$ and $\Delta'' : v \geqslant_{\overline{x_n}} \Delta : v$ by Lemma 7.27. □

Completeness

Regarding the completeness of extensions, we furthermore show that extensions do not affect the size of a derivation. We begin with the following lemma stating the completeness of a single extension.

Lemma B.19. *Let $\Gamma' : e'$ and $\Gamma : e$ be two configurations such that $\Gamma' : e' >_x \Gamma : e$. If D is a derivation for $\Gamma : e \Downarrow_1 \Delta : v$, then there exist a heap Δ' and a derivation D' for $\Gamma' : e' \Downarrow_1 \Delta' : v$ such that $\Delta' : v \geqslant_x \Delta : v$ and $|D'| = |D|$.*

Proof. By structural induction on the derivation of the premise.

(Value) For the base case of (Value) we have to show that a derivation D for $\Gamma : v \Downarrow_1 \Gamma : v$ and $\Gamma'[x \mapsto e''] : e' >_x \Gamma : v$ imply a derivation D' for $\Gamma'[x \mapsto e''] : e' \Downarrow_1 \Delta' : v$ with $\Delta' : v \geqslant_x \Gamma : v$ and $|D'| = 0$, where $v = \phi(\overline{x_k})$ with $\phi \in \mathcal{C}$ or $k < \mathsf{arity}(\phi)$, or $v \in \mathcal{V}$ with $\Gamma(v) = \text{free}$.

1. If $e' = x$, then $e'' = v$, $\Gamma' = \Gamma$, and x does not occur in $\Gamma : v$. By application of rule (Flatten), we can then construct the derivation D' with $|D'| = 0$ as
$$\frac{\Gamma[x \mapsto \blacksquare] : v \Downarrow_1 \Gamma[x \mapsto \blacksquare] : v}{\Gamma[x \mapsto v] : x \Downarrow_1 \Gamma[x \mapsto v] : v}$$
and $\Gamma[x \mapsto v] : v \geqslant_x \Gamma : v$ follows from $x \notin \mathcal{V}ar(\Gamma : v)$.
2. If $e' \neq x$, then $e' = v$ since $v = \sigma(e')$ for $\sigma = \{x \mapsto e''\}$ and e'' is a non-variable expression. Thus, we can construct the derivation D' with $|D'| = 0$ by rule (Value) as
$$\Gamma'[x \mapsto e''] : v \Downarrow_1 \Gamma'[x \mapsto e''] : v$$
and $\Gamma'[x \mapsto e''] : v \geqslant_x \Gamma : v$ follows from $\Gamma'[x \mapsto e''] : v >_x \Gamma : v$.

We continue with the inductive cases and assume as the induction hypothesis that the claim holds for the premises of the assumed derivation. We first consider the general case of $e' = x$.

General Case If $e' = x$, then $\Gamma' = \Gamma[x \mapsto e]$, x does not occur in $\Gamma : e$, and we have to show that a derivation D for $\Gamma : e \Downarrow_1 \Delta : v$ and $\Gamma[x \mapsto e] : x >_x \Gamma : e$ imply a

derivation D' for $\Gamma[x \mapsto e] : x \Downarrow_1 \Delta' : v$ with $\Delta' : v \geqslant_x \Delta : v$ and $|D'| = |D|$. In this case, $\Gamma : e \Downarrow_1 \Delta : v$ implies a derivation D'' for $\Gamma[x \mapsto \blacksquare] : e \Downarrow_1 \Delta[x \mapsto \blacksquare] : v$ such that $x \notin \mathcal{V}ar(\Delta : v)$ and $|D''| = |D|$ by Lemma B.4. We can thus construct the derivation D' with $|D'| = |D|$ by application of rule (VarExp) as

$$\frac{\Gamma[x \mapsto \blacksquare] : e \Downarrow_1 \Delta[x \mapsto \blacksquare] : v}{\Gamma[x \mapsto e] : x \Downarrow_1 \Delta[x \mapsto v] : v}$$

and $\Delta[x \mapsto v] : v \geqslant_x \Delta : v$ follows from $x \notin \mathcal{V}ar(\Delta : v)$.

For the remaining cases we proceed under the assumption that $e' \neq x$.

(VarExp) We have to show that a derivation D for $\Gamma[y \mapsto e] : y \Downarrow_1 \Delta[y \mapsto v] : v$ and $\Gamma'[x \mapsto e''] : y >_x \Gamma[y \mapsto e] : y$ where $e \notin \{\text{free}, \blacksquare\}$ imply a derivation D' for $\Gamma'[x \mapsto e''] : y \Downarrow_1 \Delta' : v$ with $\Delta' : v \geqslant_x \Delta[y \mapsto v] : v$ and $|D'| = |D|$. Since $\Gamma'[x \mapsto e''] : y >_x \Gamma[y \mapsto e] : y$, we have $\Gamma' = \Gamma''[y \mapsto e''']$, and $\Gamma''[x \mapsto e''][y \mapsto e'''] : y >_x \Gamma[y \mapsto e] : y$ implies $\Gamma''[x \mapsto e''][y \mapsto \blacksquare] : e''' >_x \Gamma[y \mapsto \blacksquare] : e$. The induction hypothesis then states that a derivation D_1 for $\Gamma[y \mapsto \blacksquare] : e \Downarrow_1 \Delta : v$ and $\Gamma''[x \mapsto e''][y \mapsto \blacksquare] : e''' >_x \Gamma[y \mapsto \blacksquare] : e$ imply a derivation D_1' for $\Gamma''[x \mapsto e''][y \mapsto \blacksquare] : e''' \Downarrow_1 \Delta'' : v$ with $\Delta'' : v \geqslant_x \Delta : v$ and $|D_1'| = |D_1|$. We can then construct the derivation D' with $|D'| = |D|$ as

$$\frac{\Gamma''[x \mapsto e''][y \mapsto \blacksquare] : e''' \Downarrow_1 \Delta'' : v}{\Gamma''[x \mapsto e''][y \mapsto e'''] : y \Downarrow_1 \Delta''[y \mapsto v] : v}$$

where $\Delta''[y \mapsto v] : v \geqslant_x \Delta[y \mapsto v] : v$ follows from $\Delta'' : v \geqslant_x \Delta : v$.

(Flatten) We have to show that a derivation D for $\Gamma : \phi(\overline{e_k}) \Downarrow_1 \Delta : v$ where $\exists i \in \{1, \ldots, k\}$ such that $e_i \notin \mathcal{V}$ and $(\overline{y_l}, \overline{e_l'}, \overline{x_k}) = \mathsf{splitArgs}(\overline{e_k})$ and $\Gamma'[x \mapsto e''] : \phi(\overline{e_k''}) >_x \Gamma : \phi(\overline{e_k})$ imply a derivation D' for $\Gamma'[x \mapsto e''] : \phi(\overline{e_k''}) \Downarrow_1 \Delta' : v$ with $\Delta' : v \geqslant_x \Delta : v$ and $|D'| = |D|$. We distinguish two cases for the occurrence of x in $\Gamma' : \phi(\overline{e_k''})$.

1. If $x = e_i''$ for any $i \in \{1, \ldots, k\}$, then $\Gamma' = \Gamma$. Since e'' is not a variable, the derivation D must be of the form

   $$\frac{\Gamma[\overline{y_l \mapsto e_l'}] : \phi(\overline{x_k}) \Downarrow_1 \Delta : v}{\Gamma : \phi(\overline{e_k}) \Downarrow_1 \Delta : v}$$

   where $x \mapsto e'' \in \overline{y_l \mapsto e_l'}$. If $l = 1$, then e'' is the only non-variable expression in $\overline{e_k}$, so that $\Gamma[x \mapsto e''] : \phi(\overline{e_k''}) \Downarrow_1 \Delta : v = \Gamma[\overline{y_l \mapsto e_l'}] : \phi(\overline{x_k}) \Downarrow_1 \Delta : v$, which holds with $|D'| = |D|$. Otherwise, we can construct the derivation D' with $|D'| = |D|$ as

   $$\frac{\Gamma[\overline{y_l \mapsto e_l'}] : \phi(\overline{x_k}) \Downarrow_1 \Delta : v}{\Gamma[x \mapsto e''] : \phi(\overline{e_k''}) \Downarrow_1 \Delta : v}$$

   and $\Delta : v \geqslant_x \Delta : v$ directly holds for both cases.

2. In all other cases, it holds $\Gamma'[x \mapsto e''][\overline{y_l \mapsto e_l'''}] : \phi(\overline{x_k}) >_x \Gamma[\overline{y_l \mapsto e_l'}] : \phi(\overline{x_k})$ for $(\overline{y_l}, \overline{e_l'''}, \overline{x_k}) = \mathsf{splitArgs}(\overline{e_k''})$. The induction hypothesis states that the derivation

D_1 for $\Gamma[\overline{y_l \mapsto e'_l}] : \phi(\overline{x_k}) \Downarrow_1 \Delta : v$ and $\Gamma'[x \mapsto e''][\overline{y_l \mapsto e'''_l}] : \phi(\overline{x_k}) >_x \Gamma[\overline{y_l \mapsto e'_l}] : \phi(\overline{x_k})$ imply a derivation D'_1 for $\Gamma'[x \mapsto e''][\overline{y_l \mapsto e'''_l}] : \phi(\overline{x_k}) \Downarrow_1 \Delta' : v$ with $\Delta' : v \geqslant_x \Delta : v$ and $|D'_1| = |D_1|$. We can thus construct the derivation D' with $|D'| = |D|$ as

$$\frac{\Gamma'[x \mapsto e'][\overline{y_l \mapsto e'''_l}] : \phi(\overline{x_k}) \Downarrow_1 \Delta' : v}{\Gamma'[x \mapsto e'] : \phi(\overline{e''_k}) \Downarrow_1 \Delta' : v}$$

where $\Delta' : v \geqslant_x \Delta : v$ follows from the induction hypothesis.

(FunEval) We have to show that a derivation D for $\Gamma : f(\overline{x_n}) \Downarrow_1 \Delta : v$ and $\Gamma'[x \mapsto e''] : f(\overline{x_n}) >_x \Gamma : f(\overline{x_n})$ imply a derivation D' for $\Gamma'[x \mapsto e''] : f(\overline{x_n}) \Downarrow_1 \Delta' : v$ with $\Delta' : v \geqslant_x \Delta : v$ and $|D'| = |D|$, where $f(\overline{x_n}) = e$ is a variable instance of a rule in P. The induction hypothesis states that a derivation D_1 for $\Gamma : e \Downarrow_1 \Delta : v$ and $\Gamma'[x \mapsto e''] : e >_x \Gamma : e$ imply a derivation D'_1 for $\Gamma'[x \mapsto e''] : e \Downarrow_1 \Delta' : v$ with $\Delta' : v \geqslant_x \Delta : v$ and $|D'_1| = |D_1|$. Since x cannot occur in $f(\overline{x_n})$ by assumption, it cannot occur in e, and thus $\Gamma'[x \mapsto e'] : e >_x \Gamma : e$ so that we can construct the derivation D' with $|D'| = |D|$ as

$$\frac{\Gamma'[x \mapsto e'] : e \Downarrow_1 \Delta' : v}{\Gamma'[x \mapsto e'] : f(\overline{x_n}) \Downarrow_1 \Delta' : v}$$

where $\Delta' : v \geqslant_x \Delta : v$ follows from the induction hypothesis.

(Let) We have to show that a derivation D for $\Gamma : \texttt{let } \{\overline{x_k = e_k}\} \texttt{ in } e \Downarrow_1 \Delta : v$ and $\Gamma'[x \mapsto e''] : \texttt{let } \{\overline{x_k = e'_k}\} \texttt{ in } e' >_x \Gamma : \texttt{let } \{\overline{x_k = e_k}\} \texttt{ in } e$ imply a derivation D' for $\Gamma'[x \mapsto e''] : \texttt{let } \{\overline{x_k = e'_k}\} \texttt{ in } e' \Downarrow_1 \Delta' : v$ with $\Delta' : v \geqslant_x \Delta : v$ and $|D'| = |D|$. The induction hypothesis states that a derivation D_1 for $\Gamma[\overline{x_k \mapsto e_k}] : e \Downarrow_1 \Delta : v$ and $\Gamma'[x \mapsto e''][\overline{x_k \mapsto e'_k}] : e' >_x \Gamma[\overline{x_k \mapsto e_k}] : e$ imply a derivation D'_1 for $\Gamma'[x \mapsto e''][\overline{x_k \mapsto e'_k}] : e' \Downarrow_1 \Delta' : v$ with $\Delta' : v \geqslant_x \Delta : v$ and $|D'_1| = |D_1|$. Since $\Gamma'[x \mapsto e''] : \texttt{let } \{\overline{x_k = e'_k}\} \texttt{ in } e' >_x \Gamma : \texttt{let } \{\overline{x_k = e_k}\} \texttt{ in } e$ implies $\Gamma'[x \mapsto e''][\overline{x_k \mapsto e'_k}] : e' >_x \Gamma[\overline{x_k \mapsto e_k}] : e$, we can construct the derivation D' with $|D'| = |D|$ as

$$\frac{\Gamma'[x \mapsto e''][\overline{x_k \mapsto e'_k}] : e' \Downarrow_1 \Delta' : v}{\Gamma'[x \mapsto e''] : \texttt{let } \{\overline{x_k = e'_k}\} \texttt{ in } e' \Downarrow_1 \Delta' : v}$$

where $\Delta' : v \geqslant_x \Delta : v$ follows from the induction hypothesis.

(Or), (Free) These cases follow with the same reasoning as for rule (Let).

(Select) We have to show that a derivation D for $\Gamma : \texttt{case } e \texttt{ of } \{ \overline{p_k \to e_k} \} \Downarrow_1 \Theta : v$ and $\Gamma'[x \mapsto e''] : \texttt{case } e' \texttt{ of } \{ \overline{p_k \to e'_k} \} >_x \Gamma : \texttt{case } e \texttt{ of } \{ \overline{p_k \to e_k} \}$ imply a derivation D' for $\Gamma'[x \mapsto e''] : \texttt{case } e' \texttt{ of } \{ \overline{p_k \to e'_k} \} \Downarrow_1 \Theta' : v$ with $\Theta' : v \geqslant_x \Theta : v$ and $|D'| = |D|$. Furthermore, we have a derivation D_1 for $\Gamma : e \Downarrow_1 \Delta : c(\overline{x_n})$ and a derivation D_2 for $\Delta : \theta(e_i) \Downarrow_1 \Theta : v$ where $c(\overline{x_n}) = \theta(p_i)$ for $i \in \{1, \ldots, k\}$. We distinguish two cases based on the occurrence of x in $\Gamma' : \texttt{case } e' \texttt{ of } \{ \overline{p_k \to e'_k} \}$.

1. If x occurs in $\Gamma' : e'$, then $\overline{e'_k} = \overline{e_k}$ and $\Gamma'[x \mapsto e''] : e' >_x \Gamma : e$, and we can apply the induction hypothesis stating that a derivation D_1 for $\Gamma : e \Downarrow_1 \Delta : c(\overline{x_n})$ and $\Gamma'[x \mapsto e''] : e' >_x \Gamma : e$ imply a derivation D'_1 for $\Gamma'[x \mapsto e''] : e' \Downarrow_1 \Delta' : c(\overline{x_n})$ with $\Delta' : c(\overline{x_n}) \geqslant_x \Delta : c(\overline{x_n})$ and $|D'_1| = |D_1|$.
   (a) If $\Delta' = \Delta$, we can construct the derivation D' with $|D'| = |D|$ as
   $$\frac{\Gamma'[x \mapsto e''] : e' \Downarrow_1 \Delta : c(\overline{x_n}) \quad \Delta : \theta(e_i) \Downarrow_1 \Theta : v}{\Gamma'[x \mapsto e''] : \texttt{case}\ e'\ \texttt{of}\ \{\ \overline{p_k \to e_k}\ \} \Downarrow_1 \Theta : v}$$
   where $\Theta : v \geqslant_x \Theta : v$ directly holds.
   (b) If $\Delta' : c(\overline{x_n}) >_x \Delta : c(\overline{x_n})$, then $\Delta' : \theta(e_i) >_x \Delta : \theta(e_i)$ and we can apply the induction hypothesis stating that the derivation D_2 for $\Delta : \theta(e_i) \Downarrow_1 \Theta : v$ and $\Delta' : \theta(e_i) >_x \Delta : \theta(e_i)$ imply a derivation D'_2 for $\Delta' : \theta(e_i) \Downarrow_1 \Theta' : v$ with $\Theta' : v \geqslant_x \Theta : v$ and $|D'_2| = |D_2|$. Thus, we can construct the derivation D' with $|D'| = |D|$ as
   $$\frac{\Gamma'[x \mapsto e''] : e' \Downarrow_1 \Delta' : c(\overline{x_n}) \quad \Delta' : \theta(e_i) \Downarrow_1 \Theta' : v}{\Gamma'[x \mapsto e''] : \texttt{case}\ e'\ \texttt{of}\ \{\ \overline{p_k \to e_k}\ \} \Downarrow_1 \Theta' : v}$$
   where $\Theta' : v \geqslant_x \Theta : v$ follows from the induction hypothesis.
2. If x does not occur in $\Gamma' : e'$, then $\Gamma' : e' = \Gamma : e$ and the derivation D_1 for $\Gamma : e \Downarrow_1 \Delta : c(\overline{x_n})$ implies a derivation D'_1 for $\Gamma[x \mapsto e''] : e \Downarrow_1 \Delta[x \mapsto e''] : c(\overline{x_n})$ such that $x \notin \mathcal{V}ar(\Delta : c(\overline{x_n}))$ and $|D'_1| = |D_1|$ by Lemma B.4. Then $\Delta[x \mapsto e''] : e'_i >_x \Delta : e_i$, and for $\sigma = \{x \mapsto e''\}$ we have $\mathcal{D}om(\sigma) \cap \mathcal{D}om(\theta) = \varnothing$, $\mathcal{D}om(\sigma) \cap \mathcal{R}an(\theta) = \varnothing$, and $\mathcal{D}om(\theta) \cap \mathcal{R}an(\sigma) = \varnothing$, so that $\sigma(\theta(e'_i)) = \theta(\sigma(e'_i))$. Thus, we have $\Delta[x \mapsto e''] : \theta(e'_i) >_x \Delta : \theta(e_i)$ and can apply the induction hypothesis stating that a derivation D_2 for $\Delta : \theta(e_i) \Downarrow_1 \Theta : v$ and $\Delta[x \mapsto e''] : \theta(e'_i) >_x \Delta : \theta(e_i)$ imply a derivation D'_2 for $\Delta[x \mapsto e''] : \theta(e'_i) \Downarrow_1 \Theta' : v$ with $\Theta' : v \geqslant_x \Theta : v$ and $|D'_2| = |D_2|$. We can then construct the derivation D' with $|D'| = |D|$ as
   $$\frac{\Gamma[x \mapsto e''] : e \Downarrow_1 \Delta[x \mapsto e''] : c(\overline{x_n}) \quad \Delta[x \mapsto e''] : \theta(e'_i) \Downarrow_1 \Theta' : v}{\Gamma[x \mapsto e''] : \texttt{case}\ e\ \texttt{of}\ \{\ \overline{p_k \to e'_k}\ \} \Downarrow_1 \Theta' : v}$$
   where $\Theta' : v \geqslant_x \Theta : v$ follows from the induction hypothesis.

(Guess) The same reasoning as for rule (Select) applies for this rule with the simplification that no variable substitution θ has to be considered. □

We can easily extend the completeness of a single extension to completeness of equality up to single extension.

Lemma B.20. *Let $\Gamma' : e'$ and $\Gamma : e$ be two configurations such that $\Gamma' : e' \geqslant_x \Gamma : e$. If D is a derivation for $\Gamma : e \Downarrow_1 \Delta : v$, then there exist a heap Δ' and a derivation D' for $\Gamma' : e' \Downarrow_1 \Delta' : v$ such that $\Delta' : v \geqslant_x \Delta : v$ and $|D| = |D'|$.*

Proof. If $\Gamma' : e' = \Gamma : e$, then the claim holds for $\Delta' = \Delta$ and $D' = D$, and for $\Gamma' : e' >_x \Gamma : e$ the claim follows from Lemma B.19. □

Finally, we can generalize this lemma to the completeness of equality up to multiple extension.

Lemma B.21 (Completeness of Equality Up To Multiple Extension)**.** *Let* $\Gamma' : e'$ *and* $\Gamma : e$ *be two configurations such that* $\Gamma' : e' \geqslant_{\overline{x_n}} \Gamma : e$*. If* D *is a derivation for* $\Gamma : e \Downarrow_1 \Delta : v$*, then there exists a heap* Δ' *and a derivation* D' *for* $\Gamma' : e' \Downarrow_1 \Delta' : v$ *such that* $\Delta' : v \geqslant_{\overline{x_n}} \Delta : v$ *and* $|D| = |D'|$*.*

Proof. By natural induction on the length n of the extension. If $n = 0$, then $\Gamma' : e' = \Gamma : e$ and the claim holds for $\Delta' = \Delta$ and $D' = D$. We continue with the inductive case of $n > 0$ and assume as the induction hypothesis that the claim holds for n. If $\Gamma' : e' \geqslant_{x \cdot \overline{x_n}} \Gamma : e$, then there must exist a configuration $\Gamma'' : e''$ such that $\Gamma' : e' \geqslant_x \Gamma'' : e''$ and $\Gamma'' : e'' \geqslant_{\overline{x_n}} \Gamma : e$. Thus, the derivation D for $\Gamma : e \Downarrow_1 \Delta : v$ and $\Gamma'' : e'' \geqslant_{\overline{x_n}} \Gamma : e$ imply a derivation D'' for $\Gamma'' : e'' \Downarrow_1 \Delta'' : v$ with $\Delta'' : v \geqslant_{\overline{x_n}} \Delta : v$ and $|D''| = |D|$ by the induction hypothesis. Furthermore, the derivation D'' for $\Gamma'' : e'' \Downarrow_1 \Delta'' : v$ and $\Gamma' : e' \geqslant_x \Gamma'' : e''$ imply a derivation D' for $\Gamma' : e' \Downarrow_1 \Delta' : v$ with $\Delta' : v \geqslant_x \Delta'' : v$ and $|D'| = |D''|$ by Lemma B.20. Finally, $\Delta' : v \geqslant_{x \cdot \overline{x_n}} \Delta : v$ follows from $\Delta' : v \geqslant_x \Delta'' : v$ and $\Delta'' : v \geqslant_{\overline{x_n}} \Delta : v$ by Lemma 7.27, and we have $|D'| = |D''| = |D|$. □

Correctness and Abstract Value Semantics

The correctness of equality up to multiple extension directly follows from the soundness and completeness results.

Theorem 7.28 (Correctness of Equality Up To Multiple Extension)**.** *Let* $\Gamma' : e'$ *and* $\Gamma : e$ *be two configurations such that* $\Gamma' : e' \geqslant_{\overline{x_n}} \Gamma : e$*. Then* $\Gamma' : e' \Downarrow_1 \Delta' : v$ *if and only if* $\Gamma : e \Downarrow_1 \Delta : v$ *such that* $\Delta' : v \geqslant_{\overline{x_n}} \Delta : v$*.*

Proof. Direct consequence of Lemma B.18 and Lemma B.21. □

Furthermore, two configurations that are equivalent up to multiple extension share the same abstract value semantics. In order to prove this claim, we need some auxiliary lemmata. The first one considers the heap lookup operation under single extension and the assumption that the variable of the extension is involved in a variable chain and not bound to a value.

Lemma B.22. *Let* $\Gamma[z \mapsto x][x \mapsto e']$ *be a heap such that* $z \neq x$*,* x *does not occur in* $\Gamma : e'$*,* $e' \notin \mathcal{V}$*, and* e' *is no partial or constructor application. Then for every variable* $y \neq x$*, it holds that* $\Gamma[z \mapsto x][x \mapsto e']^*(y)$ *is a variant of* $\Gamma[z \mapsto e']^*(y)$ *with a renaming of* x *to* z*.*

Proof. For the general case of $y = z$, we have $\Gamma[z \mapsto x, x \mapsto e']^*(z) = \Gamma[x \mapsto e']^*(x) = x$ and $[z \mapsto e']^*(z) = z$, which are variants for the assumed renaming.

In the following, we assume $y \neq z$ and proceed by well-founded induction on the complexity of the heap Γ. The base case of $\mathcal{M}_\Gamma = \varnothing$ implies $\Gamma = []$, and since we assume $y \neq z$, then $[z \mapsto x, x \mapsto e']^*(y) = y = [z \mapsto e']^*(y)$. For the inductive cases

we assume as the induction hypothesis that the claim holds for all heaps Γ' with $\mathcal{M}_{\Gamma'} <_{\mathsf{mul}} \mathcal{M}_\Gamma$.

- ▷ If $y \notin \mathcal{D}om(\Gamma)$ or $\Gamma(y) \in \{\blacksquare, \mathsf{free}, y\}$, then $\Gamma[z \mapsto x, x \mapsto e']^*(y) = y = \Gamma[z \mapsto x]^*(y)$.
- ▷ If $\Gamma = \Gamma'[y \mapsto y']$ with $y' \neq y$, then $\Gamma[z \mapsto x, x \mapsto e']^*(y) = \Gamma'[z \mapsto x, x \mapsto e']^*(y')$ and $\Gamma[z \mapsto e']^*(y) = \Gamma'[z \mapsto e']^*(y')$, which are variants by the induction hypothesis.
- ▷ If $\Gamma = \Gamma'[y \mapsto \phi(\overline{e_k})]$ with $\phi \in \mathcal{C}$ or $k < \mathsf{arity}(\phi)$, then $\Gamma[z \mapsto x, x \mapsto e']^*(y) = \phi(\overline{\Gamma'[\overline{y_l \mapsto e'_l}, z \mapsto x, x \mapsto e']^*(x_k)})$ for $(\overline{y_l}, \overline{e'_l}, \overline{x_k}) = \mathsf{splitArgs}(\overline{e_k})$, and $\Gamma[z \mapsto e']^*(y) = \phi(\overline{\Gamma'[\overline{y_l \mapsto e'_l}, z \mapsto e']^*(x_k)})$, which are variants by the induction hypothesis.
- ▷ For all other cases, we have $\Gamma[z \mapsto x, x \mapsto e']^*(y) = y = \Gamma[z \mapsto e']^*(y)$. □

The next lemma considers the remaining cases under a single extension, i. e., the extension variable is not involved in a variable chain or bound to a value.

Lemma B.23. *Let $\Gamma[x \mapsto e']$ be a heap such that x occurs exactly once in Γ and not in e', $\sigma = \{x \mapsto e'\}$, and either $x \notin \mathcal{R}an(\Gamma)$ or $e' = \phi(\overline{e_k})$ with $\phi \in \mathcal{C}$ or $k < \mathsf{arity}(\phi)$. Then for every variable $y \neq x$, it holds $\Gamma[x \mapsto e']^*(y) = \sigma(\Gamma)^*(y)$.*

Proof. By well-founded induction on the complexity of the heap Γ. The base case of $\mathcal{M}_\Gamma = \varnothing$ implies $\Gamma = []$, and $[x \mapsto e']^*(y) = y = []^*(y) = \sigma(\Gamma)^*(y)$. For the inductive cases we assume as the induction hypothesis that the claim holds for all heaps Γ' with $\mathcal{M}_{\Gamma'} <_{\mathsf{mul}} \mathcal{M}_\Gamma$.

- ▷ If $y \notin \mathcal{D}om(\Gamma)$ or $\Gamma(y) \in \{\blacksquare, \mathsf{free}, y\}$, then $\Gamma[x \mapsto e']^*(y) = y = \sigma(\Gamma)^*(y)$.
- ▷ If $\Gamma = \Gamma'[y \mapsto x]$, then $e' = \phi(\overline{e_k})$ by assumption, so that $\Gamma[x \mapsto e']^*(y) = \Gamma'[x \mapsto \phi(\overline{e_k})]^*(x) = \phi(\overline{\Gamma'[\overline{y_l \mapsto e'_l}]^*(x_k)})$ for $(\overline{y_l}, \overline{e'_l}, \overline{x_k}) = \mathsf{splitArgs}(\overline{e_k})$, as well as $\sigma(\Gamma)^*(y) = \sigma(\Gamma')[y \mapsto e']^*(y) = \Gamma'[y \mapsto \phi(\overline{e_k})]^*(y) = \phi(\overline{\Gamma'[\overline{y_l \mapsto e'_l}]^*(x_k)})$.
- ▷ If $\Gamma = \Gamma'[y \mapsto y']$ with $y' \notin \{x, y\}$, then $\Gamma[x \mapsto e']^*(y) = \Gamma'[x \mapsto e']^*(y')$ and $\sigma(\Gamma)^*(y) = \sigma(\Gamma'[y \mapsto y'])^*(y) = \sigma(\Gamma')^*(y')$, which are equal by the induction hypothesis.
- ▷ If $\Gamma = \Gamma'[y \mapsto \phi(\overline{e_k})]$ with $\phi \in \mathcal{C}$ or $k < \mathsf{arity}(\phi)$, and $(\overline{y_l}, \overline{e'_l}, \overline{x_k}) = \mathsf{splitArgs}(\overline{e_k})$, then we distinguish two cases.
  - ▷ If $x = e_i$ for any $i \in \{1, \ldots, k\}$, then $\Gamma[x \mapsto e']^*(y) = \overline{\phi(\overline{\Gamma'[\overline{y_l \mapsto e'_l}, x \mapsto e']^*(x_k)})}$, as well as $\sigma(\Gamma)^*(y) = \Gamma'[y \mapsto \phi(\overline{\sigma(e_k)})]^*(y) = \phi(\overline{\Gamma'[\overline{y_l \mapsto e'_l}, x \mapsto e']^*(x_k)})$, since x cannot occur in Γ' or $\overline{e'_l}$ by assumption.

▷ Otherwise, we have $\Gamma[x \mapsto e']^*(y) = \phi(\overline{\Gamma'[\overline{y_l \mapsto e'_l}, x \mapsto e']^*(x_k)})$ and $\sigma(\Gamma)^*(y) = \sigma(\Gamma')[y \mapsto \phi(\overline{\sigma(e_k)})]^*(y) = \phi(\overline{\sigma(\Gamma'[\overline{y_l \mapsto e'_l}])^*(x_k)})$, which are equal by the induction hypothesis.

▷ For all other cases, we have $\Gamma[x \mapsto e']^*(y) = y = \sigma(\Gamma)^*(y)$. □

We can then combine both auxiliary lemmata to show that two configurations that are equal up to single extension share the same abstract value semantics.

Lemma B.24. *Let $\Gamma' : v$ and $\Gamma : v$ be two configurations such that $\Gamma' : v \geqslant_x \Gamma : v$ and v is a value. Then $[\![\Gamma' : v]\!] = [\![\Gamma : v]\!]$.*

Proof. For $\Gamma' : v = \Gamma : v$ or $v \in \mathcal{V}$ the claim follows immediately. If $\Gamma' : v >_x \Gamma : v$ and $v \notin \mathcal{V}$, then $\Gamma' = \Gamma''[x \mapsto e']$ and x does not occur in v or e'. If x does also not occur in Γ'', then the claim follows from Lemma B.6, otherwise it follows either from Lemma B.23 or from Lemma B.22 and the fact that the abstract value semantics considers equivalence classes of variants. □

Lemma 7.31 (Abstract Value Semantics under Equality up to Multiple Extension)**.** *Let $\Gamma' : v$ and $\Gamma : v$ be two configurations such that $\Gamma' : v \geqslant_{\overline{x_n}} \Gamma : v$ and v is a value. Then $[\![\Gamma' : v]\!] = [\![\Gamma : v]\!]$.*

Proof. By natural induction on the length n of the extension. If $n = 0$, then $\Gamma' = \Gamma$ and the claim follows immediately. We continue with the inductive case of $n > 0$ and assume as the induction hypothesis that the claim holds for n. If $\Gamma' : v \geqslant_{x \cdot \overline{x_n}} \Gamma : v$, then there must exist a heap Γ'' such that $\Gamma' : v \geqslant_x \Gamma'' : v$ and $\Gamma'' : v \geqslant_{\overline{x_n}} \Gamma : v$. We then have $[\![\Gamma' : v]\!] = [\![\Gamma'' : v]\!]$ by Lemma B.24 and $[\![\Gamma'' : v]\!] = [\![\Gamma : v]\!]$ by the induction hypothesis so that the claim follows. □

B.2.6 Correctness of Partial Evaluation

Based on the results stated above we continue to show the correctness of partial evaluation. We consider the soundness and completeness of partial evaluation individually before we combine both results to show its correctness.

Theorem B.25 (Soundness of Partial Evaluation)**.** *Let P be a program, E a finite set of linear and partially evaluable expressions, ρ an independent renaming for E, and P' a partial evaluation of E in P under ρ such that P' is E'-closed, where $E' = \rho(E)$. If $\Gamma' : e' = \mathsf{ren}_\rho(\Gamma : e)$ is an E'-closed configuration and $\Gamma' : e' \Downarrow_1 \Delta' : v'$ holds in P', then there exist two heaps Δ and Δ'' such that $\Gamma : e \Downarrow_1 \Delta : v$ holds in P with $\Delta'' : v \geqslant_{\mathsf{ext}} \Delta : v$ and $\Delta' : v' = \mathsf{ren}_\rho(\Delta'' : v)$.*

Proof. We prove the claim by structural induction on the derivation of the premise. Since all expressions in E' are linear left-hand sides and P' as well as $\Gamma' : e'$ are E'-closed, every configuration in the derivation for $\Gamma' : e' \Downarrow_1 \Delta' : v'$ is also E'-closed.

(Value) For the base case of rule (Value) we have to show that $\Gamma' : v' \Downarrow_1 \Gamma' : v'$ and $\Gamma' : v' = \mathsf{ren}_\rho(\Gamma : e)$ imply $\Gamma : e \Downarrow_1 \Delta : v$ with $\Delta'' : v \geqslant_{\mathsf{ext}} \Delta : v$ and $\Delta' : v' = \mathsf{ren}_\rho(\Delta'' : v)$, where $v' = \phi(\overline{x_k})$ with $\phi \in \mathcal{C}$ or $k < \mathsf{arity}(\phi)$, or $v' \in \mathcal{V}$ with $\Gamma'(v') = \mathsf{free}$.

1. If $v' \in \mathcal{V}$ or $v' = c(\overline{x_k})$ with $c \in \mathcal{C}$, then $e = v'$ by the definition of renaming, so that $\Gamma : e \Downarrow_1 \Gamma : e$ holds by rule (Value) and the claim follows for $\Delta'' = \Gamma$.
2. If $v' = f'(\overline{x_k})$ with $f' \in \mathcal{F}^{(n)}$ and $k < n$, then $e = f(\overline{x_k})$ with $f \in \mathcal{F}^{(n)}$ by the definition of renaming. Then $\Gamma : f(\overline{x_k}) \Downarrow_1 \Gamma : f(\overline{x_k})$ holds by rule (Value) and the claim follows for $\Delta'' = \Gamma$.

We continue with the inductive cases and assume as the induction hypothesis that the claim holds for the premises of the assumed derivation.

(VarExp) We have to show that $\Gamma'[x \mapsto e'] : x \Downarrow_1 \Delta'[x \mapsto v'] : v'$ where $e' \notin \{\mathsf{free}, \blacksquare\}$ and $\Gamma'[x \mapsto e'] : x = \mathsf{ren}_\rho(\Gamma[x \mapsto e] : x)$ imply $\Gamma[x \mapsto e] : x \Downarrow_1 \Delta[x \mapsto v] : v$ with $\Delta''[x \mapsto v] : v \geqslant_{\mathsf{ext}} \Delta[x \mapsto v] : v$ and $\mathsf{ren}_\rho(\Delta''[x \mapsto v] : v) = \Delta'[x \mapsto v'] : v'$. Since $\mathsf{ren}_\rho(\Gamma[x \mapsto e] : x) = \Gamma'[x \mapsto e'] : x$, we have $\mathsf{ren}_\rho(\Gamma[x \mapsto \blacksquare] : e) = \Gamma'[x \mapsto \blacksquare] : e'$. We can thus apply the induction hypothesis stating that $\Gamma'[x \mapsto \blacksquare] : e' \Downarrow_1 \Delta' : v'$ and $\mathsf{ren}_\rho(\Gamma[x \mapsto \blacksquare] : e) = \Gamma'[x \mapsto \blacksquare] : e'$ imply $\Gamma[x \mapsto \blacksquare] : e \Downarrow_1 \Delta : v$ with $\Delta'' : v \geqslant_{\mathsf{ext}} \Delta : v$ and $\mathsf{ren}_\rho(\Delta'' : v) = \Delta' : v'$. We can then construct the following derivation by application of rule (VarExp):

$$\frac{\Gamma[x \mapsto \blacksquare] : e \Downarrow_1 \Delta : v}{\Gamma[x \mapsto e] : x \Downarrow_1 \Delta[x \mapsto v] : v}$$

Furthermore, $\Delta'' : v \geqslant_{\mathsf{ext}} \Delta : v$ implies $\Delta''[x \mapsto v] : v \geqslant_{\mathsf{ext}} \Delta[x \mapsto v] : v$ and $\mathsf{ren}_\rho(\Delta'' : v) = \Delta' : v'$ implies $\mathsf{ren}_\rho(\Delta''[x \mapsto v] : v) = \Delta'[x \mapsto v'] : v'$.

(Flatten) We have to show that $\Gamma' : \phi'(\overline{e'_k}) \Downarrow_1 \Delta' : v'$ and $\mathsf{ren}_\rho(\Gamma : e) = \Gamma' : \phi'(\overline{e'_k})$ imply $\Gamma : e \Downarrow_1 \Delta : v$ with $\Delta'' : v \geqslant_{\mathsf{ext}} \Delta : v$ and $\mathsf{ren}_\rho(\Delta'' : v) = \Delta' : v'$, where $\exists i \in \{1, \ldots, k\}$ such that $e'_i \notin \mathcal{V}$ and $(\overline{y_l}, \overline{e'''_l}, \overline{x_k}) = \mathsf{splitArgs}(\overline{e'_k})$. We distinguish two cases for ϕ' in the following.

1. If $\phi' \in \mathcal{C}$ or $k < \mathsf{arity}(\phi')$, then $\mathsf{ren}_\rho(e) = \phi'(\overline{e'_k})$ implies $e = \phi(\overline{e_k})$, $\mathsf{ren}_\rho(\phi(\overline{x_k})) = \phi'(\overline{x_k})$ and $\mathsf{ren}_\rho(e_i) = e'_i$ for all $i \in \{1, \ldots, k\}$. Furthermore, $(\overline{y_l}, \overline{e''_l}, \overline{x_k}) = \mathsf{splitArgs}(\overline{e_k})$ since splitArgs will extract subexpressions at the same positions according to Lemma B.14, and thus $\Gamma'[\overline{y_l \mapsto e'''_l}] : \phi'(\overline{x_k}) = \mathsf{ren}_\rho(\Gamma[\overline{y_l \mapsto e''_l}] : \phi(\overline{x_k}))$. We can then apply the induction hypothesis stating that $\Gamma'[\overline{y_l \mapsto e'''_l}] : \phi'(\overline{x_k}) \Downarrow_1 \Delta' : v'$ and $\Gamma'[\overline{y_l \mapsto e'''_l}] : \phi'(\overline{x_k}) = \mathsf{ren}_\rho(\Gamma[\overline{y_l \mapsto e''_l}] : \phi(\overline{x_k}))$ imply $\Gamma[\overline{y_l \mapsto e''_l}] : \phi(\overline{x_k}) \Downarrow_1 \Delta : v$ with $\Delta'' : v \geqslant_{\mathsf{ext}} \Delta : v$ and $\mathsf{ren}_\rho(\Delta'' : v) = \Delta' : v'$, so that we can construct the following derivation using rule (Flatten)

$$\frac{\Gamma[\overline{y_l \mapsto e''_l}] : \phi(\overline{x_k}) \Downarrow_1 \Delta : v}{\Gamma : \phi(\overline{e_k}) \Downarrow_1 \Delta : v}$$

where $\Delta'' : v \geqslant_{\mathsf{ext}} \Delta : v$ and $\mathsf{ren}_\rho(\Delta'' : v) = \Delta' : v'$ hold by the induction hypothesis.

2. If $\phi' \in \mathcal{F}^{(k)}$, then $\phi'(\overline{x_k}) = \sigma(e')$ for $e' = \phi'(\overline{y_k}) \in E'$ and a variable substitution σ, since $\phi'(\overline{e'_k})$ is E'-closed and E' contains only linear left-hand sides. Since $\mathsf{ren}_\rho(e) = \phi'(\overline{e'_k})$, then e must be an instance of $e' \in E$ by the definition of renaming. We then assume $e = \theta(\sigma(e'))$ such that $\mathcal{D}om(\theta) \cap \mathcal{D}om(\sigma) = \varnothing$, $\mathcal{D}om(\theta) \cap \mathcal{R}an(\sigma) = \varnothing$, and $\theta = \{\overline{y_l \mapsto e''_l}\}$ contains only non-variable substitutions, so that $\mathsf{ren}_\rho(e''_j) = e'''_j$ for $j \in \{1, \ldots, l\}$ and $\mathsf{ren}_\rho(\Gamma[\overline{y_l \mapsto e''_l}] : \sigma(e')) = \Gamma'[\overline{y_l \mapsto e'''_l}] : \phi'(\overline{x_k})$. In this case the induction hypothesis states that $\Gamma'[\overline{y_l \mapsto e'''_l}] : \phi'(\overline{x_k}) \Downarrow_1 \Delta' : v'$ and $\Gamma'[\overline{y_l \mapsto e'''_l}] : \phi'(\overline{x_k}) = \mathsf{ren}_\rho(\Gamma[\overline{y_l \mapsto e''_l}] : \sigma(e'))$ imply $\Gamma[\overline{y_l \mapsto e''_l}] : \sigma(e') \Downarrow_1 \Delta''' : v$ with $\Delta'' : v \geqslant_{\mathsf{ext}} \Delta''' : v$ and $\mathsf{ren}_\rho(\Delta'' : v) = \Delta' : v'$. Since $e = \theta(\sigma(e'))$, $\overline{y_l}$ are fresh and every variable in $\overline{y_l}$ occurs once in $\sigma(e')$, it furthermore holds that $\Gamma[\overline{y_l \mapsto e''_l}] : \sigma(e') \geqslant_{\mathsf{ext}} \Gamma : e$, and thus $\Gamma[\overline{y_l \mapsto e''_l}] : \sigma(e') \Downarrow_1 \Delta''' : v$ and $\Gamma[\overline{y_l \mapsto e''_l}] : \sigma(e') \geqslant_{\mathsf{ext}} \Gamma : e$ imply $\Gamma : e \Downarrow_1 \Delta : v$ with $\Delta''' : v \geqslant_{\mathsf{ext}} \Delta : v$ by Lemma B.18. Hence, $\Delta'' : v \geqslant_{\mathsf{ext}} \Delta''' : v \geqslant_{\mathsf{ext}} \Delta : v$ holds by Lemma 7.27 and freshness of $\overline{y_l}$, and $\mathsf{ren}_\rho(\Delta'' : v) = \Delta' : v'$ holds by the induction hypothesis.

(FunEval) We have to show that $\Gamma' : f'(\overline{x_n}) \Downarrow_1 \Delta' : v'$ where $f'(\overline{x_n}) = e'''$ is a variable instance of a rule in P' and $\mathsf{ren}_\rho(\Gamma : e) = \Gamma' : f'(\overline{x_n})$ imply $\Gamma : e \Downarrow_1 \Delta : v$ with $\Delta'' : v \geqslant_{\mathsf{ext}} \Delta : v$ and $\Delta' : v' = \mathsf{ren}_\rho(\Delta'' : v)$. Since $f'(\overline{x_n}) = \mathsf{ren}_\rho(e)$ and $f'(\overline{x_n})$ is E'-closed, it must hold that $e = \sigma(e')$ for a variable substitution σ and $e' \in E$. From the pre-partial evaluation $e' \to e''$ we conclude that the definition of f' is constructed with $\rho(e') = f'(\overline{x_n})$ and $e''' = \mathsf{ren}_\rho(e'')$, so that we have the following derivation in P':

$$\frac{\Gamma' : \sigma(e''') \Downarrow_1 \Delta' : v'}{\Gamma' : f'(\overline{x_n}) \Downarrow_1 \Delta' : v'}$$

Since σ is a variable substitution, we have $\sigma(e''') = \mathsf{ren}_\rho(\sigma(e''))$ and can thus apply the induction hypothesis stating that $\Gamma' : \mathsf{ren}_\rho(\sigma(e'')) \Downarrow_1 \Delta' : v'$ and $\mathsf{ren}_\rho(\Gamma : \sigma(e'')) = \Gamma' : \mathsf{ren}_\rho(\sigma(e''))$ imply $\Gamma : \sigma(e'') \Downarrow_1 \Delta : v$ with $\Delta'' : v \geqslant_{\mathsf{ext}} \Delta : v$ and $\Delta' : v' = \mathsf{ren}_\rho(\Delta'' : v)$. Since $e' \to e''$ is a pre-partial evaluation, so is $\sigma(e') \to \sigma(e'')$ by Lemma B.12, and thus $\Gamma : \sigma(e'') \Downarrow_1 \Delta : v$ implies $\Gamma : \sigma(e') \Downarrow_1 \Delta : v$ and thus $\Gamma : e \Downarrow_1 \Delta : v$ by Lemma B.10, and $\Delta'' : v \geqslant_{\mathsf{ext}} \Delta : v$ and $\Delta' : v' = \mathsf{ren}_\rho(\Delta'' : v)$ hold by the induction hypothesis.

(Let) For this case we have to show that $\Gamma' : \mathtt{let}\ \{\overline{x_k = e'_k}\}\ \mathtt{in}\ e' \Downarrow_1 \Delta' : v'$ and $\mathsf{ren}_\rho(\Gamma : \mathtt{let}\ \{\overline{x_k = e_k}\}\ \mathtt{in}\ e) = \Gamma' : \mathtt{let}\ \{\overline{x_k = e'_k}\}\ \mathtt{in}\ e'$ imply that the statement $\Gamma : \mathtt{let}\ \{\overline{x_k = e_k}\}\ \mathtt{in}\ e \Downarrow_1 \Delta : v$ holds with $\Delta'' : v \geqslant_{\mathsf{ext}} \Delta : v$ and $\Delta' : v' = \mathsf{ren}_\rho(\Delta'' : v)$, where the induction hypothesis states that $\Gamma'[\overline{x_k \mapsto e'_k}] : e' \Downarrow_1 \Delta' : v'$ and $\mathsf{ren}_\rho(\Gamma[\overline{x_k \mapsto e_k}] : e) = \Gamma'[\overline{x_k \mapsto e'_k}] : e'$ imply $\Gamma[\overline{x_k \mapsto e_k}] : e \Downarrow_1 \Delta : v$ with $\Delta'' : v \geqslant_{\mathsf{ext}} \Delta : v$ and $\Delta' : v' = \mathsf{ren}_\rho(\Delta'' : v)$. Since $\mathsf{ren}_\rho(\Gamma : \mathtt{let}\ \{\overline{x_k = e_k}\}\ \mathtt{in}\ e) = \Gamma' : \mathtt{let}\ \{\overline{x_k = e'_k}\}\ \mathtt{in}\ e'$ implies $\mathsf{ren}_\rho(\Gamma[\overline{x_k \mapsto e_k}] : e) = \Gamma'[\overline{x_k \mapsto e'_k}] : e'$, we can

construct the following derivation using rule (Let)

$$\frac{\Gamma[\overline{x_k \mapsto e_k}] : e \Downarrow_1 \Delta : v}{\Gamma : \texttt{let } \{\overline{x_k = e_k}\} \texttt{ in } e \Downarrow_1 \Delta : v}$$

where $\Delta'' : v \geqslant_{\mathsf{ext}} \Delta : v$ and $\Delta' : v' = \mathsf{ren}_\rho(\Delta'' : v)$ follow from the induction hypothesis.

(Or), (Free) These cases follow with the same reasoning as for rule (Let).

(Select) In this case we have to show that $\Gamma' : \texttt{case } e' \texttt{ of } \{ \overline{p_k \to e'_k} \} \Downarrow_1 \Theta' : v'$ and $\mathsf{ren}_\rho(\Gamma : \texttt{case } e \texttt{ of } \{ \overline{p_k \to e_k} \}) = \Gamma' : \texttt{case } e' \texttt{ of } \{ \overline{p_k \to e'_k} \}$ imply that $\Gamma : \texttt{case } e \texttt{ of } \{ \overline{p_k \to e_k} \} \Downarrow_1 \Theta : v$ holds with $\Theta'' : v \geqslant_{\mathsf{ext}} \Theta : v$ and $\Theta' : v' = \mathsf{ren}_\rho(\Theta'' : v)$. Since $\mathsf{ren}_\rho(\Gamma : \texttt{case } e \texttt{ of } \{ \overline{p_k \to e_k} \}) = \Gamma' : \texttt{case } e' \texttt{ of } \{ \overline{p_k \to e'_k} \}$ implies $\mathsf{ren}_\rho(\Gamma : e) = \Gamma' : e'$, we can apply the induction hypothesis stating that $\Gamma' : e' \Downarrow_1 \Delta' : c(\overline{x_n})$ and $\mathsf{ren}_\rho(\Gamma : e) = \Gamma' : e'$ imply $\Gamma : e \Downarrow_1 \Delta : c(\overline{x_n})$ with $\Delta'' : c(\overline{x_n}) \geqslant_{\mathsf{ext}} \Delta : c(\overline{x_n})$ and $\mathsf{ren}_\rho(\Delta'' : c(\overline{x_n})) = \Delta' : c(\overline{x_n})$, where $c(\overline{x_n}) = \sigma(p_i)$ for $i \in \{1, \ldots, k\}$. Then $\mathsf{ren}_\rho(\Delta'' : e_i) = \Delta' : e'_i$, and since σ is a variable substitution, we have $\mathsf{ren}_\rho(\Delta'' : \sigma(e_i)) = \Delta' : \sigma(e'_i)$. We can thus apply the induction hypothesis stating that $\Delta' : \sigma(e'_i) \Downarrow_1 \Theta' : v'$ and $\mathsf{ren}_\rho(\Delta'' : \sigma(e_i)) = \Delta' : \sigma(e'_i)$ imply $\Delta'' : \sigma(e_i) \Downarrow_1 \Theta''' : v$ with $\Theta'' : v \geqslant_{\mathsf{ext}} \Theta''' : v$ and $\Theta' : v' = \mathsf{ren}_\rho(\Theta'' : v)$. Furthermore, all unbound variables in e_i must be bound in Γ or introduced in p_i, so that $\Delta'' : c(\overline{x_n}) \geqslant_{\mathsf{ext}} \Delta : c(\overline{x_n})$ implies $\Delta'' : \sigma(e_i) \geqslant_{\mathsf{ext}} \Delta : \sigma(e_i)$. By Lemma B.18, $\Delta'' : \sigma(e_i) \geqslant_{\mathsf{ext}} \Delta : \sigma(e_i)$ and $\Delta'' : \sigma(e_i) \Downarrow_1 \Theta''' : v$ imply $\Delta : \sigma(e_i) \Downarrow_1 \Theta : v$ with $\Theta''' : v \geqslant_{\mathsf{ext}} \Theta : v$. We can then construct the following derivation by application of rule (Select)

$$\frac{\Gamma : e \Downarrow_1 \Delta : c(\overline{x_n}) \quad \Delta : \sigma(e_i) \Downarrow_1 \Theta : v}{\Gamma : \texttt{case } e \texttt{ of } \{ \overline{p_k \to e_k} \} \Downarrow_1 \Theta : v}$$

with $\Theta'' : v \geqslant_{\mathsf{ext}} \Theta''' : v \geqslant_{\mathsf{ext}} \Theta : v$ and $\Theta' : v' = \mathsf{ren}_\rho(\Theta'' : v)$.

(Guess) This case follows with the same reasoning as for rule (Select) with the simplification that no variable substitution has to be considered and the fact that the extension cannot comprise the free variable that gets instantiated. □

The proof of completeness proceeds in a similar way like the proof of soundness, but requires a more complex relation for well-founded induction.

Theorem B.26 (Completeness of Partial Evaluation)**.** *Let P be a program, E a finite set of linear and partially evaluable expressions, ρ an independent renaming for E, and P' a partial evaluation of E in P under ρ such that P' is E'-closed, where $E' = \rho(E)$. If $\Gamma : e \Downarrow_1 \Delta : v$ in P and $\mathsf{ren}_\rho(\Gamma : e)$ is E'-closed, then there exists a heap Δ' such that $\mathsf{ren}_\rho(\Gamma : e) \Downarrow_1 \mathsf{ren}_\rho(\Delta' : v)$ in P' with $\Delta' : v \geqslant_{\mathsf{ext}} \Delta : v$ and $\mathsf{ren}_\rho(\Delta' : v)$ is E'-closed.*

Proof. Note that $\mathsf{ren}_\rho(e)$ is E'-closed by assumption, so that e must be E-closed by Lemma 7.23. We prove the claim by well-founded induction on the pair of the size

of the assumed derivation and the complexity of the in-configuration of the proved statement, which is defined as

$$|D|^* = (|D|, \mathcal{M}_{\Gamma:e}) \quad \text{if } D \text{ is a derivation for } \Gamma : e \Downarrow_1 \Delta : v \,.$$

Two derivations D_1 and D_2 are then ordered by the lexicographical ordering on $|\cdot|^*$, i. e., for $(s_1, c_1) = |D_1|^*$ and $(s_2, c_2) = |D_2|^*$ it holds

$$(s_1, c_1) < (s_2, c_2) :\Leftrightarrow s_1 < s_2 \vee (s_1 = s_2 \wedge c_1 <_{\mathsf{mul}} c_2) \,.$$

(Value) For the base case of rule (Value) we have to show that $\Gamma : v \Downarrow_1 \Gamma : v$ and $\mathsf{ren}_\rho(\Gamma : v)$ is E'-closed imply $\mathsf{ren}_\rho(\Gamma : v) \Downarrow_1 \mathsf{ren}_\rho(\Gamma' : v)$ with $\Gamma' : v \geqslant_{\mathsf{ext}} \Gamma : v$ and $\mathsf{ren}_\rho(\Gamma' : v)$ is E'-closed, where $v = \phi(\overline{x_k})$ with $\phi \in \mathcal{C}$ or $k < \mathsf{arity}(\phi)$, or $v \in \mathcal{V}$ with $\Gamma(v) = \mathsf{free}$.

1. If $v \in \mathcal{V}$ or $v = c(\overline{x_k})$ with $c \in \mathcal{C}$, then $\mathsf{ren}_\rho(v) = v$ and $\mathsf{ren}_\rho(\Gamma : v) \Downarrow_1 \mathsf{ren}_\rho(\Gamma : v)$ holds by rule (Value). Furthermore, $\mathsf{ren}_\rho(\Gamma : v)$ is E'-closed by assumption and $\Gamma : v \geqslant_{\mathsf{ext}} \Gamma : v$ holds by equality.
2. If $v = f(\overline{x_k})$ with $f \in \mathcal{F}^{(n)}$, $k < n$, and v is E-closed, then $f(\overline{y_n}) \in E$ and $\mathsf{ren}_\rho(v) = f'(\overline{x_k})$ for $\rho(f(\overline{y_n})) = f'(\overline{y_n})$ and $f' \in \mathcal{F}^{(n)}$. Thus, $\mathsf{ren}_\rho(\Gamma : v) \Downarrow_1 \mathsf{ren}_\rho(\Gamma : v)$ holds by rule (Value), $\mathsf{ren}_\rho(\Gamma : v)$ is E'-closed by assumption, and $\Gamma : v \geqslant_{\mathsf{ext}} \Gamma : v$ holds by equality.

We continue with the inductive cases where we consider a derivation D for $\Gamma : e \Downarrow_1 \Delta : v$ and assume as the induction hypothesis that the claim holds for all derivations D' such that $|D'|^* < |D|^*$. We first cover the general case that e is an instance of some expression in E.

Instance If $e = \sigma'(e')$ is an instance of $e' \in E$ where $\mathcal{R}an(\sigma')$ is E-closed, then we assume $\sigma' = \theta \circ \sigma$ such that $\mathcal{D}om(\theta) \cap \mathcal{D}om(\sigma) = \varnothing$ and $\mathcal{D}om(\theta) \cap \mathcal{R}an(\sigma) = \varnothing$, and σ contains only variable substitutions while θ contains only non-variable substitutions. We distinguish two cases in the following.

1. If $\theta = id$, then $e = \sigma(e')$, and thus $\mathsf{ren}_\rho(e) = \sigma(f'(\overline{x_n}))$ for $f'(\overline{x_n}) = \rho(e')$ by the definition of renaming. Thus, we have to show that $\Gamma : e \Downarrow_1 \Delta : v$ and $\mathsf{ren}_\rho(\Gamma) : \sigma(f'(\overline{x_n}))$ is E'-closed imply $\mathsf{ren}_\rho(\Gamma) : \sigma(f'(\overline{x_n})) \Downarrow_1 \mathsf{ren}_\rho(\Delta') : \mathsf{ren}_\rho(v)$ with $\Delta' : v \geqslant_{\mathsf{ext}} \Delta : v$ and $\mathsf{ren}_\rho(\Delta' : v)$ is E'-closed. By the definition of partial evaluation, the function f' is defined as $f'(\overline{x_n}) = \mathsf{ren}_\rho(e'')$ where $e' \to e''$ is the pre-partial evaluation of e'. Then $\sigma(e') \to \sigma(e'')$ is also a pre-partial evaluation by Lemma B.12, and thus the derivation D for $\Gamma : e \Downarrow_1 \Delta : v$ implies $\sigma(e'') = e'_1 \;?\; \ldots \;?\; e'_n$ and a derivation D' for $\Gamma : e'_i \Downarrow_1 \Delta : v$ with $i \in \{1, \ldots, n\}$ such that $|D'| < |D|$ by Lemma B.11, and thus $|D'|^* < |D|^*$. Since $e'_1 \;?\; \ldots \;?\; e'_n$ is not partially evaluable for $n \geqslant 2$, we have $\mathsf{ren}_\rho(\sigma(e'')) = \mathsf{ren}_\rho(e'_1) \;?\; \ldots \;?\; \mathsf{ren}_\rho(e'_n)$ by the definition of renaming. Furthermore, P' and $\mathsf{ren}_\rho(\Gamma)$ are E'-closed by assumption, so that $\mathsf{ren}_\rho(\sigma(e''))$, $\mathsf{ren}_\rho(e'_i)$ and $\mathsf{ren}_\rho(\Gamma : e'_i)$ must be E'-closed.

We can thus apply the induction hypothesis stating that $\Gamma : e'_i \Downarrow_1 \Delta : v$ and $\mathsf{ren}_\rho(\Gamma : e'_i)$ is E'-closed imply $\mathsf{ren}_\rho(\Gamma : e'_i) \Downarrow_1 \mathsf{ren}_\rho(\Delta' : v)$ with $\Delta' : v \geqslant_{\mathsf{ext}} \Delta : v$ and $\mathsf{ren}_\rho(\Delta' : v)$ is E'-closed. Since $\mathsf{ren}_\rho(\Gamma : e'_i) \Downarrow_1 \mathsf{ren}_\rho(\Delta' : v)$ furthermore implies $\mathsf{ren}_\rho(\Gamma : \sigma(e'')) \Downarrow_1 \mathsf{ren}_\rho(\Delta' : v)$ by multiple applications of rule (Or), we can construct the derivation

$$\frac{\mathsf{ren}_\rho(\Gamma) : \mathsf{ren}_\rho(\sigma(e'')) \Downarrow_1 \mathsf{ren}_\rho(\Delta') : \mathsf{ren}_\rho(v)}{\mathsf{ren}_\rho(\Gamma) : \sigma(f'(\overline{x_n})) \Downarrow_1 \mathsf{ren}_\rho(\Delta') : \mathsf{ren}_\rho(v)}$$

where $\Delta' : v \geqslant_{\mathsf{ext}} \Delta : v$ and E'-closedness of $\mathsf{ren}_\rho(\Delta' : v)$ follow from the induction hypothesis.

2. If $\theta \neq id$, then we assume $\theta = \{\overline{y_l \mapsto e'_l}\}$. Then for $f'(\overline{x_n}) = \rho(e')$ we have $\mathsf{ren}_\rho(e) = f'(\overline{\mathsf{ren}_\rho(\sigma'(x_n))})$, and thus have to show that $\Gamma : e \Downarrow_1 \Delta : v$ and $\mathsf{ren}_\rho(\Gamma : e)$ is E'-closed imply $\mathsf{ren}_\rho(\Gamma) : f'(\overline{\mathsf{ren}_\rho(\sigma'(x_n))}) \Downarrow_1 \mathsf{ren}_\rho(\Delta') : \mathsf{ren}_\rho(v)$ with $\Delta' : v \geqslant_{\mathsf{ext}} \Delta : v$ and $\mathsf{ren}_\rho(\Delta' : v)$ is E'-closed. Since $e = \theta(\sigma(e'))$, $\overline{y_l}$ are fresh, and every variable in $\overline{y_l}$ occurs once in $\sigma(e')$, it holds that $\Gamma[\overline{y_l \mapsto e'_l}] : \sigma(e') \geqslant_{\mathsf{ext}} \Gamma : e$, and thus the derivation D for $\Gamma : e \Downarrow_1 \Delta : v$ and $\Gamma[\overline{y_l \mapsto e'_l}] : \sigma(e') \geqslant_{\mathsf{ext}} \Gamma : e$ imply a derivation D' for $\Gamma[\overline{y_l \mapsto e'_l}] : \sigma(e') \Downarrow_1 \Delta'' : v$ with $\Delta'' : v \geqslant_{\mathsf{ext}} \Delta : v$ and $|D'| = |D|$ by Lemma B.21. Furthermore, $\mathcal{M}_{\Gamma[\overline{y_l \mapsto e'_l}] : \sigma(e')} <_{\mathsf{mul}} \mathcal{M}_{\Gamma : e}$ since θ contains non-variable substitutions, so that $|D'|^* < |D|^*$. Since $\mathsf{ren}_\rho(\Gamma : e)$ is E'-closed by assumption, so must be $\mathsf{ren}_\rho(\sigma(e'))$ and $\overline{\mathsf{ren}_\rho(e'_l)}$, and thus also $\mathsf{ren}_\rho(\Gamma[\overline{y_l \mapsto e'_l}] : \sigma(e'))$. We can then apply the induction hypothesis stating that $\Gamma[\overline{y_l \mapsto e'_l}] : \sigma(e') \Downarrow_1 \Delta'' : v$ and $\mathsf{ren}_\rho(\Gamma[\overline{y_l \mapsto e'_l}] : \sigma(e'))$ is E'-closed imply $\mathsf{ren}_\rho(\Gamma[\overline{y_l \mapsto e'_l}] : \sigma(e')) \Downarrow_1 \mathsf{ren}_\rho(\Delta' : v)$ with $\Delta' : v \geqslant_{\mathsf{ext}} \Delta'' : v$ and $\mathsf{ren}_\rho(\Delta' : v)$ is E'-closed. We can then construct the following derivation using rule (Flatten)

$$\dfrac{\begin{array}{c}\mathsf{ren}_\rho(\Gamma[\overline{y_l \mapsto e'_l}] : \sigma(e')) \Downarrow_1 \mathsf{ren}_\rho(\Delta' : v) \\ = \quad \mathsf{ren}_\rho(\Gamma)[\overline{y_l \mapsto \mathsf{ren}_\rho(e'_l)}] : f'(\overline{\sigma(x_n)}) \Downarrow_1 \mathsf{ren}_\rho(\Delta') : \mathsf{ren}_\rho(v)\end{array}}{\begin{array}{c}\mathsf{ren}_\rho(\Gamma) : f'(\overline{\mathsf{ren}_\rho(\sigma'(x_n))}) \Downarrow_1 \mathsf{ren}_\rho(\Delta') : \mathsf{ren}_\rho(v) \\ = \quad \mathsf{ren}_\rho(\Gamma : e) \Downarrow_1 \mathsf{ren}_\rho(\Delta' : v)\end{array}}$$

which is valid since σ is a variable substitution, so that $\sigma(\mathsf{ren}_\rho(e')) = \mathsf{ren}_\rho(\sigma(e'))$ and $(\overline{y_l}, \overline{\mathsf{ren}_\rho(e'_l)}, \overline{\sigma(x_n)}) = \mathsf{splitArgs}(\overline{\mathsf{ren}_\rho(\sigma'(x_n))})$ by Lemma B.14. Furthermore, $\Delta' : v \geqslant_{\mathsf{ext}} \Delta'' : v \geqslant_{\mathsf{ext}} \Delta : v$ holds by Lemma 7.27 and freshness of $\overline{y_l}$, and $\mathsf{ren}_\rho(\Delta' : v)$ is E'-closed by the induction hypothesis.

We continue with the remaining cases where we consider a derivation D for $\Gamma : e \Downarrow_1 \Delta : v$ such that e is not an instance of any $e' \in E$.

(VarExp) We have to show that $\Gamma[x \mapsto e] : x \Downarrow_1 \Delta[x \mapsto v] : v$ where $e \notin \{\mathsf{free}, \blacksquare\}$ and $\mathsf{ren}_\rho(\Gamma[x \mapsto e] : x)$ is E'-closed imply $\mathsf{ren}_\rho(\Gamma[x \mapsto e] : x) \Downarrow_1 \mathsf{ren}_\rho(\Delta'[x \mapsto v] : v)$ with $\Delta'[x \mapsto v] : v \geqslant_{\mathsf{ext}} \Delta[x \mapsto v] : v$ and $\mathsf{ren}_\rho(\Delta'[x \mapsto v] : v)$ is E'-closed. Since $\mathcal{M}_{\Gamma[x \mapsto \blacksquare] : e} <_{\mathsf{mul}} \mathcal{M}_{\Gamma[x \mapsto e] : x}$, and E'-closedness of $\mathsf{ren}_\rho(\Gamma[x \mapsto e] : x)$ implies E'-closedness of $\mathsf{ren}_\rho(\Gamma[x \mapsto \blacksquare] : e)$, we can apply the induction hypothesis stating

that $\Gamma[x \mapsto \blacksquare] : e \Downarrow_1 \Delta : v$ and $\mathsf{ren}_\rho(\Gamma[x \mapsto \blacksquare] : e)$ is E'-closed imply $\mathsf{ren}_\rho(\Gamma[x \mapsto \blacksquare] : e) \Downarrow_1 \mathsf{ren}_\rho(\Delta' : v)$ with $\Delta' : v \geqslant_{\mathsf{ext}} \Delta : v$ and $\mathsf{ren}_\rho(\Delta' : v)$ is E'-closed. We can then construct the following derivation using rule (VarExp):

$$\dfrac{\begin{array}{rc} & \mathsf{ren}_\rho(\Gamma[x \mapsto \blacksquare] : e) \Downarrow_1 \mathsf{ren}_\rho(\Delta' : v) \\ = & \mathsf{ren}_\rho(\Gamma)[x \mapsto \blacksquare] : \mathsf{ren}_\rho(e) \Downarrow_1 \mathsf{ren}_\rho(\Delta') : \mathsf{ren}_\rho(v) \end{array}}{\begin{array}{rc} & \mathsf{ren}_\rho(\Gamma)[x \mapsto \mathsf{ren}_\rho(e)] : x \Downarrow_1 \mathsf{ren}_\rho(\Delta')[x \mapsto \mathsf{ren}_\rho(v)] : \mathsf{ren}_\rho(v) \\ = & \mathsf{ren}_\rho(\Gamma[x \mapsto e] : x) \Downarrow_1 \mathsf{ren}_\rho(\Delta'[x \mapsto v] : v) \end{array}}$$

Furthermore, since $\mathsf{ren}_\rho(\Delta' : v)$ is E'-closed, so is $\mathsf{ren}_\rho(\Delta'[x \mapsto v] : v)$, and $\Delta' : v \geqslant_{\mathsf{ext}} \Delta : v$ implies $\Delta'[x \mapsto v] : v \geqslant_{\mathsf{ext}} \Delta[x \mapsto v] : v$.

(Flatten) We have to show that $\Gamma : \phi(\overline{e_k}) \Downarrow_1 \Delta : v$ and $\mathsf{ren}_\rho(\Gamma : \phi(\overline{e_k}))$ is E'-closed imply $\mathsf{ren}_\rho(\Gamma : \phi(\overline{e_k})) \Downarrow_1 \mathsf{ren}_\rho(\Delta' : v)$ with $\Delta' : v \geqslant_{\mathsf{ext}} \Delta : v$ and $\mathsf{ren}_\rho(\Delta' : v)$ is E'-closed, where $\exists i \in \{1, \ldots, k\}$ such that $e_i \notin \mathcal{V}$ and $(\overline{y_l}, \overline{e'_l}, \overline{x_k}) = \mathsf{splitArgs}(\overline{e_k})$. We distinguish two cases for $\phi(\overline{e_k})$.

1. If $\phi \in \mathcal{C}$ or $k < \mathsf{arity}(\phi)$, then $\mathsf{ren}_\rho(\phi(\overline{e_k})) = \phi'(\overline{\mathsf{ren}_\rho(e_k)})$ for $\mathsf{ren}_\rho(\phi(\overline{x_k})) = \phi'(\overline{x_k})$ by the definition of renaming. Furthermore, since $\mathsf{ren}_\rho(\Gamma : \phi(\overline{e_k}))$ is E'-closed, so must be $\phi(\overline{x_k})$ and $\overline{e_k}$, and thus also $\mathsf{ren}_\rho(\Gamma[\overline{y_l \mapsto e'_l}] : \phi(\overline{x_k}))$. Since $\mathcal{M}_{\Gamma[\overline{y_l \mapsto e'_l}] : \phi(\overline{x_k})} < \mathcal{M}_{\Gamma : \phi(\overline{e_k})}$, we can apply the induction hypothesis stating that $\Gamma[\overline{y_l \mapsto e'_l}] : \phi(\overline{x_k}) \Downarrow_1 \Delta : v$ and $\mathsf{ren}_\rho(\Gamma[\overline{y_l \mapsto e'_l}] : \phi(\overline{x_k}))$ is E'-closed imply $\mathsf{ren}_\rho(\Gamma[\overline{y_l \mapsto e'_l}] : \phi(\overline{x_k})) \Downarrow_1 \mathsf{ren}_\rho(\Delta' : v)$ with $\Delta' : v \geqslant_{\mathsf{ext}} \Delta : v$ and $\mathsf{ren}_\rho(\Delta' : v)$ is E'-closed. We can thus construct the following derivation using rule (Flatten)

$$\dfrac{\begin{array}{rc} & \mathsf{ren}_\rho(\Gamma[\overline{y_l \mapsto e'_l}] : \phi(\overline{x_k})) \Downarrow_1 \mathsf{ren}_\rho(\Delta' : v) \\ = & \mathsf{ren}_\rho(\Gamma)[\overline{y_l \mapsto \mathsf{ren}_\rho(e'_l)}] : \phi'(\overline{x_k}) \Downarrow_1 \mathsf{ren}_\rho(\Delta') : \mathsf{ren}_\rho(v) \end{array}}{\begin{array}{rc} & \mathsf{ren}_\rho(\Gamma) : \phi'(\overline{\mathsf{ren}_\rho(e_k)}) \Downarrow_1 \mathsf{ren}_\rho(\Delta') : \mathsf{ren}_\rho(v) \\ = & \mathsf{ren}_\rho(\Gamma : \phi(\overline{e_k})) \Downarrow_1 \mathsf{ren}_\rho(\Delta' : v) \end{array}}$$

since $(\overline{y_l}, \overline{\mathsf{ren}_\rho(e'_l)}, \overline{x_k}) = \mathsf{splitArgs}(\overline{\mathsf{ren}_\rho(e_k)})$ by Lemma B.14. Furthermore, E'-closedness of $\mathsf{ren}_\rho(\Delta' : v)$ and $\Delta' : v \geqslant_{\mathsf{ext}} \Delta : v$ hold by the induction hypothesis.

2. If $\phi \in \mathcal{F}^{(k)}$ and $\mathsf{ren}_\rho(\Gamma : \phi(\overline{e_k}))$ is E'-closed, then $\phi(\overline{e_k})$ is E-closed and thus an instance of some $e' \in E$, so that this case has already been covered before.

(FunEval) If $\Gamma : f(\overline{x_n}) \Downarrow_1 \Delta : v$ and $\mathsf{ren}_\rho(\Gamma : f(\overline{x_n}))$ is E'-closed, then $f(\overline{x_n})$ is E-closed and thus must be instance of some $e' \in E$, so that this case has also been covered before.

(Let) In this case we have to show that $\Gamma : \texttt{let } \{\overline{x_k = e_k}\} \texttt{ in } e \Downarrow_1 \Delta : v$ and $\mathsf{ren}_\rho(\Gamma : \texttt{let } \{\overline{x_k = e_k}\} \texttt{ in } e)$ is E'-closed imply $\mathsf{ren}_\rho(\Gamma : \texttt{let } \{\overline{x_k = e_k}\} \texttt{ in } e) \Downarrow_1 \mathsf{ren}_\rho(\Delta' : v)$ with $\Delta' : v \geqslant_{\mathsf{ext}} \Delta : v$ and $\mathsf{ren}_\rho(\Delta' : v)$ is E'-closed, where $\texttt{let } \{\overline{x_k = e_k}\} \texttt{ in } e$ is recursively renamed. Then $\mathsf{ren}_\rho(\Gamma) : \texttt{let } \{\overline{x_k = \mathsf{ren}_\rho(e_k)}\} \texttt{ in } \mathsf{ren}_\rho(e)$ is E'-closed, and thus also $\mathsf{ren}_\rho(\Gamma[\overline{x_k \mapsto e_k}] : e)$. Furthermore, $\mathcal{M}_{\Gamma[\overline{x_k \mapsto e_k}] : e} < \mathcal{M}_{\Gamma : \texttt{let } \{\overline{x_k = e_k}\} \texttt{ in } e}$,

so that we can apply the induction hypothesis stating that $\Gamma[\overline{y_k \mapsto e_k}] : e \Downarrow_1 \Delta : v$ and $\mathsf{ren}_\rho(\Gamma[\overline{x_k \mapsto e_k}] : e)$ is E'-closed imply $\mathsf{ren}_\rho(\Gamma[\overline{y_k \mapsto e_k}] : e) \Downarrow_1 \mathsf{ren}_\rho(\Delta' : v)$ with $\Delta' : v \geqslant_{\mathsf{ext}} \Delta : v$ and $\mathsf{ren}_\rho(\Delta' : v)$ is E'-closed. Hence, we can construct the following derivation using rule (Let)

$$\dfrac{\begin{array}{c}\mathsf{ren}_\rho(\Gamma[\overline{y_k \mapsto e_k}] : e) \Downarrow_1 \mathsf{ren}_\rho(\Delta' : v) \\ = \quad \mathsf{ren}_\rho(\Gamma)[\overline{y_k \mapsto \mathsf{ren}_\rho(e_k)}] : \mathsf{ren}_\rho(e) \Downarrow_1 \mathsf{ren}_\rho(\Delta') : \mathsf{ren}_\rho(v)\end{array}}{\begin{array}{rc}& \mathsf{ren}_\rho(\Gamma) : \mathtt{let}\ \{\overline{x_k = \mathsf{ren}_\rho(e_k)}\}\ \mathtt{in}\ \mathsf{ren}_\rho(e) \Downarrow_1 \mathsf{ren}_\rho(\Delta') : \mathsf{ren}_\rho(v) \\ = & \mathsf{ren}_\rho(\Gamma : \mathtt{let}\ \{\overline{x_k = e_k}\}\ \mathtt{in}\ e) \Downarrow_1 \mathsf{ren}_\rho(\Delta' : v)\end{array}}$$

where $\Delta' : v \geqslant_{\mathsf{ext}} \Delta : v$ and $\mathsf{ren}_\rho(\Delta' : v)$ is E'-closed hold by the induction hypothesis.

(Or), (Free) These cases follow with the same reasoning as for rule (Let).

(Select) In this case we have to show that $\Gamma : \mathtt{case}\ e\ \mathtt{of}\ \{\ \overline{p_k \to e_k}\ \} \Downarrow_1 \Theta : v$ and $\mathsf{ren}_\rho(\Gamma : \mathtt{case}\ e\ \mathtt{of}\ \{\ \overline{p_k \to e_k}\ \})$ is E'-closed imply $\mathsf{ren}_\rho(\Gamma : \mathtt{case}\ e\ \mathtt{of}\ \{\ \overline{p_k \to e_k}\ \}) \Downarrow_1 \mathsf{ren}_\rho(\Theta' : v)$ with $\Theta' : v \geqslant_{\mathsf{ext}} \Theta : v$ and $\mathsf{ren}_\rho(\Theta' : v)$ is E'-closed, where $\mathtt{case}\ e\ \mathtt{of}\ \{\ \overline{p_k \to e_k}\ \}$ is recursively renamed, so that $\mathsf{ren}_\rho(\Gamma : e)$ and $\overline{\mathsf{ren}_\rho(e_k)}$ must be E'-closed. We can thus apply the induction hypothesis stating that $\Gamma : e \Downarrow_1 \Delta : c(\overline{x_n})$ and $\mathsf{ren}_\rho(\Gamma : e)$ is E'-closed imply $\mathsf{ren}_\rho(\Gamma : e) \Downarrow_1 \mathsf{ren}_\rho(\Delta' : c(\overline{x_n}))$ with $\Delta' : c(\overline{x_n}) \geqslant_{\mathsf{ext}} \Delta : c(\overline{x_n})$ and $\mathsf{ren}_\rho(\Delta' : c(\overline{x_n}))$ is E'-closed, where $c(\overline{x_n}) = \sigma(p_i)$ for $i \in \{1, \ldots, k\}$. Furthermore, e_i may only reference variables bound in Γ or p_i, so that $\Delta' : e_i \geqslant_{\mathsf{ext}} \Delta : e_i$, and because σ is a variable substitution, also $\Delta' : \sigma(e_i) \geqslant_{\mathsf{ext}} \Delta : \sigma(e_i)$. Thus, $\Delta : \sigma(e_i) \Downarrow_1 \Theta : v$ and $\Delta' : \sigma(e_i) \geqslant_{\mathsf{ext}} \Delta : \sigma(e_i)$ imply $\Delta' : \sigma(e_i) \Downarrow_1 \Theta'' : v$ with the same derivation size and $\Theta'' : v \geqslant_{\mathsf{ext}} \Theta : v$ by Lemma B.21. Furthermore, since $\mathsf{ren}_\rho(\Delta')$ and $\overline{\mathsf{ren}_\rho(e_k)}$ are E'-closed and σ is a variable substitution, then $\mathsf{ren}_\rho(\Delta' : \sigma(e_i))$ is E'-closed. We can then apply the induction hypothesis stating that $\Delta' : \sigma(e_i) \Downarrow_1 \Theta'' : v$ and $\mathsf{ren}_\rho(\Delta' : \sigma(e_i))$ is E'-closed imply $\mathsf{ren}_\rho(\Delta' : \sigma(e_i)) \Downarrow_1 \mathsf{ren}_\rho(\Theta' : v)$ with $\Theta' : v \geqslant_{\mathsf{ext}} \Theta'' : v$ and $\mathsf{ren}_\rho(\Theta' : v)$ is E'-closed. Since σ is a variable substitution, we have $\mathsf{ren}_\rho(\sigma(e_i)) = \sigma(\mathsf{ren}_\rho(e_i))$, and can thus construct the following derivation by rule (Select)

$$\dfrac{\mathsf{ren}_\rho(\Gamma) : \mathsf{ren}_\rho(e) \Downarrow_1 \mathsf{ren}_\rho(\Delta') : c(\overline{x_n}) \quad \mathsf{ren}_\rho(\Delta') : \sigma(\mathsf{ren}_\rho(e_i)) \Downarrow_1 \mathsf{ren}_\rho(\Theta') : \mathsf{ren}_\rho(v)}{\begin{array}{rc}& \mathsf{ren}_\rho(\Gamma) : \mathtt{case}\ \mathsf{ren}_\rho(e)\ \mathtt{of}\ \{\ \overline{p_k \to \mathsf{ren}_\rho(e_k)}\ \} \Downarrow_1 \mathsf{ren}_\rho(\Theta') : \mathsf{ren}_\rho(v) \\ = & \mathsf{ren}_\rho(\Gamma : \mathtt{case}\ e\ \mathtt{of}\ \{\ \overline{p_k \to e_k}\ \}) \Downarrow_1 \mathsf{ren}_\rho(\Theta' : v)\end{array}}$$

where $\Theta' : v \geqslant_{\mathsf{ext}} \Theta'' : v \geqslant_{\mathsf{ext}} \Theta : v$ and $\mathsf{ren}_\rho(\Theta' : v)$ is E'-closed hold by the induction hypothesis.

(Guess) This case follows with the same reasoning as for rule (Select) with the simplification that no variable substitution has to be considered, and the fact that the instantiation of a logic variable to a constructor-rooted value does neither affect E'-closedness nor renaming. □

Finally, we can combine the results on soundness and completeness of partial evaluation to show its correctness.

Theorem 7.29 (Correctness of Partial Evaluation)**.** *Let P be a program, E a finite set of linear and partially evaluable expressions, ρ an independent renaming for E, P' a partial evaluation of E in P under ρ such that P' is E'-closed where $E' = \rho(E)$, and $\Gamma : e$ a configuration such that $\mathsf{ren}_\rho(\Gamma : e)$ is E'-closed. Then $\Gamma : e \Downarrow_1 \Delta : v$ in P if and only if $\mathsf{ren}_\rho(\Gamma : e) \Downarrow_1 \mathsf{ren}_\rho(\Delta' : v)$ in P' such that $\mathsf{ren}_\rho(\Delta' : v)$ is E'-closed and $\Delta' : v \geqslant_{\mathsf{ext}} \Delta : v$.*

Proof. Direct consequence of Theorem B.25 and Theorem B.26. □

B.3 Non-Embedding Abstraction

After the correctness proofs for partial evaluations we show the correctness results for the non-embedding abstraction operator. The following auxiliary lemma states that for two comparable expressions with a non-trivial most-specific linear generalization, the complexity of the generalization and the range of the substitutions is smaller than the complexity of the original expressions, so that the mslg computation reduces complexity.

Lemma B.27 (Complexity of Non-Trivial Most-Specific Linear Generalization)**.** *Let q and e be two comparable expressions with $q \trianglelefteq^* e$, and $(g, \sigma, \theta) = \mathsf{mslg}(q, e)$ with $g \notin \mathcal{V}$ their non-trivial most-specific linear generalization. Then for the multisets $S_\sigma = \{e \in \mathcal{R}an(\sigma) \mid e \notin \mathcal{V}\}$, $S_\theta = \{e \in \mathcal{R}an(\theta) \mid e \notin \mathcal{V}\}$, and $\{q, e\}$, it holds $\mathcal{M}_{\{g\} \cup S_\sigma \cup S_\theta} <_{\mathsf{mul}} \mathcal{M}_{\{q,e\}}$.*

Proof. We distinguish three cases based on the relation of $\mathsf{depth}(g)$ and $\mathsf{depth}(q)$.

1. $\mathsf{depth}(g) > \mathsf{depth}(q)$ cannot occur, since $q = \sigma(g)$ by the definition of mslg.

2. We consider the case of $\mathsf{depth}(g) = \mathsf{depth}(q)$. Let X and Y be the multiset complexities of $S_\sigma \cup S_\theta$ and $\{e\}$, respectively, and let X_1 and X_2 be the multiset complexities of S_σ and S_θ. Since $q = \sigma(g)$ and g is non-trivial, we have $\mathsf{depth}(d) < \mathsf{depth}(q)$ for all $d \in S_\sigma$. Since $q \trianglelefteq^* e$, then $\mathsf{depth}(q) \leqslant \mathsf{depth}(e)$ and thus $\mathsf{depth}(d) < \mathsf{depth}(e)$ for all $d \in S_\sigma$, so that $\forall n \in X_1 : \exists n' \in Y : n < n'$. Since $e = \theta(g)$ and g is non-trivial, we also have $\mathsf{depth}(d) < \mathsf{depth}(e)$ for all $d \in S_\theta$, and therefore $\forall n \in X_2 : \exists n' \in Y : n < n'$. In summary, we have $\forall n \in X : \exists n' \in Y : n < n'$, and thus $X <_{\mathsf{mul}} Y$.

3. If $\mathsf{depth}(g) < \mathsf{depth}(q)$, then let X and Y denote the multiset complexities of $\{g\} \cup S_\sigma \cup S_\theta$ and $\{q, e\}$, and let X_1, X_2 and X_3 denote the multiset complexities of $\{g\}$, S_σ and S_θ, respectively. Since $\mathsf{depth}(g) < \mathsf{depth}(q)$, for $X_1 = \{\mathsf{depth}(g)\}$ there trivially exists $\mathsf{depth}(q) \in Y : \mathsf{depth}(g) < \mathsf{depth}(q)$. For X_2 and X_3, the result can be proven with the same reasoning as for the case above. □

We turn our attention towards the correctness of the abstraction operator, where we follow the general proof structure of the narrowing-driven partial evaluation scheme [AFV98]. The following proposition states that the operation $\mathsf{abs}_{\mathsf{emb}}$ indeed is an abstraction operator in the sense of Definition 7.35.

Proposition 7.49 ($\mathsf{abs}^*_{\mathsf{emb}}$ is an Abstraction Operator)**.** *The function* $\mathsf{abs}^*_{\mathsf{emb}}$ *is an abstraction operator in the sense of Definition 7.35, i. e., given a sequence q of linear and partially evaluable expressions and a finite set of expressions E, it satisfies the following conditions for* $q' = \mathsf{abs}^*_{\mathsf{emb}}(q, E)$*:*

1. *if $e' \in S_{q'}$, then e' is linear and partially evaluable, and*
2. *if $e' \in S_{q'}$, then there exists an expression $e \in (S_q \cup E)$ such that $e|_p = \sigma(e')$ for some non-variable position p and substitution σ, and*
3. *for all $e \in (S_q \cup E)$, e is closed with respect to $S_{q'}$*

where S_q denotes the set of expressions contained in the sequence q.

Proof. Since only linear and partially evaluable expressions are added to the sequence, the first condition is trivially fulfilled. Furthermore, $\mathsf{abs}^*_{\mathsf{emb}}$ only decomposes expressions and applies the msl$\mathsf{g}$ operator, so that it is obvious that it does not introduce new expressions that do not appear in S_q or E, and thus satisfies the second condition.

We continue with the proof of the third condition, where we proceed by well-founded induction on the complexity of the multiset of expressions $S_q \cup E$. For the base case of $S_q \cup E = \varnothing$, we have $\mathsf{abs}^*_{\mathsf{emb}}(\varepsilon, \varnothing) = \varepsilon$, and the claim follows.

We assume as the induction hypothesis that the claim holds for all finite sequences q' and finite sets of expressions E' such that $\mathcal{M}_{S_{q'} \cup E'} <_{\mathsf{mul}} \mathcal{M}_{S_q \cup E}$ (the complexity is strictly smaller), i. e., all expressions in $S_{q'} \cup E'$ are closed with respect to the expressions in $\mathsf{abs}^*_{\mathsf{emb}}(q', E')$. For the inductive cases, we generally assume $S_q \cup E \neq \varnothing$. If the set E of expressions is empty, then $\mathsf{abs}^*_{\mathsf{emb}}(q, \varnothing) = q$, and the claim trivially holds. Thus, we assume $E = E_0 \cup \{e\}$ with $e \notin E_0$ for the remaining cases.

1. If e is neither a function call nor an annotated expression, then $q' = \mathsf{abs}^*_{\mathsf{emb}}(q, E) = \mathsf{abs}^*_{\mathsf{emb}}(q, E_0 \cup \{\overline{e_k}\})$ for $\{\overline{e_k}\} = \mathcal{NVS}ub_1(e)$. Since $\mathcal{M}_{S_q \cup E_0 \cup \{\overline{e_k}\}} <_{\mathsf{mul}} \mathcal{M}_{S_q \cup E_0 \cup \{e\}}$, then $S_q \cup E_0 \cup \{\overline{e_k}\}$ is $S_{q'}$-closed by the induction hypothesis and thus e by the definition of closedness, so that $S_q \cup E$ is $S_{q'}$-closed.

2. If e is a fully saturated function call or annotated expression that is no partial function call, then $q' = \mathsf{add}_{\mathsf{emb}}(q'', e')$ with $q'' = \mathsf{abs}^*_{\mathsf{emb}}(q, E_0)$ and $e' = \mathsf{lin}(e)$. Since $e = \sigma(e')$ for a variable substitution σ, closedness of e' also implies closedness of e.

   (a) If $e' \equiv q_i \in q''$, then $q' = q''$. Since $\mathcal{M}_{S_q \cup E_0} <_{\mathsf{mul}} \mathcal{M}_{S_q \cup E_0 \cup \{e\}}$, then $S_q \cup E_0$ is $S_{q''}$-closed by the induction hypothesis, and $e' \equiv q_i$ is $S_{q''}$-closed since $q_i \in q''$, so that $S_q \cup E$ is $S_{q''}$-closed.

(b) If e' is partially evaluable and does no embed any comparable $q_i \in S_{q''}$, then $q' = q'' \cdot e'$. Because $\mathcal{M}_{S_q \cup E_0} <_{\mathsf{mul}} \mathcal{M}_{S_q \cup E_0 \cup \{e\}}$, then $S_q \cup E_0$ is $S_{q''}$-closed by the induction hypothesis and thus also $S_{q'}$-closed. Furthermore, e' is $S_{q'}$-closed since $e' \in S_{q'}$, so that $S_q \cup E$ is $S_{q'}$-closed.

(c) If e' is not partially evaluable or q_i is the last expression in q'' such that $\mathsf{comparable}(q_i, e)$ and $q_i \trianglelefteq^* e$ with $(g, \sigma, \theta) = \mathsf{mslg}(q_i, e')$ and g is a variable, then e' cannot be a function application, such that the closedness of e' also follows from the closedness of its subexpressions. Let $\{\overline{e_k}\} = \mathcal{NVS}ub_1(e')$ denote the non-variable subexpressions of e', so that $q' = \mathsf{abs}^*_{\mathsf{emb}}(q'', \{\overline{\langle\!\langle e_k \rangle\!\rangle}\})$ by the definition of $\mathsf{abs}^*_{\mathsf{emb}}$. Since $\mathcal{M}_{S_q \cup E_0 \cup \{\overline{e_k}\}} <_{\mathsf{mul}} \mathcal{M}_{S_q \cup E}$, then $S_q \cup E_0 \cup \{\overline{e_k}\}$ is $S_{q'}$-closed by the induction hypothesis, and therefore also e' as well as $S_q \cup E$.

(d) Otherwise, it holds that e' is partially evaluable and q_i is the last expression in q'' such that $\mathsf{comparable}(q_i, e)$ and $q_i \trianglelefteq^* e$ and $(g, \sigma, \theta) = \mathsf{mslg}(q_i, e')$ with $g \notin \mathcal{V}$. Then for $q''' = q'' \setminus q_i$ and $S = \{\langle\!\langle s \rangle\!\rangle \mid s \in \{g\} \cup \mathcal{R}an(\sigma) \cup \mathcal{R}an(\theta), s \notin \mathcal{V}\}$ we have $q' = \mathsf{abs}_{\mathsf{emb}}(q''', S)$. Because $\mathcal{M}_{S_q \cup E_0} <_{\mathsf{mul}} \mathcal{M}_{S_q \cup E_0 \cup \{e\}}$, then $S_q \cup E_0$ is $S_{q''}$-closed by the induction hypothesis. Furthermore, q_i and e' are S-closed by definition of closedness and mslg, such that $S_{q''}$ is $(S_{q'''} \cup S)$-closed, and therefore $S_q \cup E_0$ is $(S_{q'''} \cup S)$-closed by Lemma 7.19. By Lemma B.27, we have $\mathcal{M}_{S_{q'''} \cup S} <_{\mathsf{mul}} \mathcal{M}_{S_{q'''} \cup \{q_i, e'\}} = \mathcal{M}_{S_{q''} \cup \{e'\}}$. Furthermore, by the definition of $\mathsf{abs}^*_{\mathsf{emb}}$, it is obvious that $\mathcal{M}_{S_{q''}} \leqslant_{\mathsf{mul}} \mathcal{M}_{S_q \cup E_0}$, since q'' can at most contain the original expressions in q and E_0 or a set of expressions with a lower complexity. Thus, we have $\mathcal{M}_{S_{q'''} \cup S} <_{\mathsf{mul}} \mathcal{M}_{S_{q''} \cup E_0 \cup \{e\}} = \mathcal{M}_{S_q \cup E}$, and can conclude by the induction hypothesis that $S_{q'''} \cup S$ is $S_{q'}$-closed, so that $S_q \cup E_0$ is $S_{q'}$-closed by Lemma 7.19. Finally, since e' is S-closed, $S_q \cup E$ is also $S_{q'}$-closed.

3. If $e = f(\overline{e_k})$ or $e = \langle\!\langle f(\overline{e_k}) \rangle\!\rangle$ with $f \in \mathcal{F}^{(n)}$ and $k < n$, we distinguish two cases. If there exists a non-variable expression e_i for $i \in \{1, \ldots, k\}$, then the claim follows from the induction hypothesis and the definition of closedness, otherwise it follows from closedness of $f(\overline{x_n})$ with the same reasoning as for the previous case. □

Finally, we show that the non-embedding abstraction operator satisfies the non-embedding property.

Lemma 7.52 ($\mathsf{abs}^*_{\mathsf{emb}}$ preserves the Non-Embedding Property)**.** *Let q be a finite sequence of linear and partially evaluable expressions satisfying the non-embedding property, and E a finite set of expressions. Then $q' = \mathsf{abs}^*_{\mathsf{emb}}(q, E)$ satisfies the non-embedding property.*

Proof. We prove the claim by well-founded induction on the complexity of the multiset of expressions $S_q \cup E$. For the base case of $S_q \cup E = \varnothing$ we have $\mathsf{abs}^*_{\mathsf{emb}}(\varepsilon, \varnothing) = \varepsilon$, and the claim holds.

We assume as the induction hypothesis that the claim holds for all sequences q' and finite sets of expressions E' such that $\mathcal{M}_{S_{q'} \cup E'} <_{\mathsf{mul}} \mathcal{M}_{S_q \cup E}$ (the complexity

is strictly smaller), i. e., $\mathsf{abs}^*_{\mathsf{emb}}(q', E')$ satisfies the non-embedding property. For the inductive cases we generally assume $S_q \cup E \neq \varnothing$. If the set E of expressions is empty, we have $\mathsf{abs}^*_{\mathsf{emb}}(q, \varnothing) = q$, and the claim holds by assumption. Thus, we assume $E = E_0 \cup \{e\}$ with $e \notin E_0$ for the remaining cases.

1. If e is neither a function call nor an annotated expression, then $q' = \mathsf{abs}^*_{\mathsf{emb}}(q, E) = \mathsf{abs}^*_{\mathsf{emb}}(q, E_0 \cup \{\overline{e_k}\})$ for $\{\overline{e_k}\} = \mathcal{NVS}ub_1(e)$. Since $\mathcal{M}_{S_q \cup E_0 \cup \{\overline{e_k}\}} <_{\mathsf{mul}} \mathcal{M}_{S_q \cup E_0 \cup \{e\}}$, then q' satisfies the non-embedding property by the induction hypothesis.

2. If e is a fully saturated function application or an annotated expression that is no partial function call, then $q' = \mathsf{add}_{\mathsf{emb}}(q'', e')$ with $q'' = \mathsf{abs}^*_{\mathsf{emb}}(q, E_0)$ and $e' = \mathsf{lin}(e)$, and q'' satisfies the non-embedding property by the induction hypothesis.

   (a) If $e' \equiv q_i \in q''$, then $q' = q''$, and the claim holds.

   (b) If e' is partially evaluable and does not embed any $q_i \in q''$, then $q' = q'' \cdot e'$. Since q'' satisfies the non-embedding property and e' does not embed any of the comparable expressions in $S_{q''}$, q' must also satisfy the non-embedding property.

   (c) If e' is not partially evaluable or q_i is the last expression in q'' such that $\mathsf{comparable}(q_i, e)$ and $q_i \trianglelefteq^* e$ with $(g, \sigma, \theta) = \mathsf{mslg}(q_i, e')$ and $g \in \mathcal{V}$, then let $\{\overline{e_k}\} = \mathcal{NVS}ub_1(e')$ denote the non-variable subexpressions of e'. In this case we have $q' = \mathsf{abs}^*_{\mathsf{emb}}(q'', \{\overline{\langle\!\langle e_k \rangle\!\rangle}\})$ by the definition of $\mathsf{abs}^*_{\mathsf{emb}}$, and since $\mathcal{M}_{S_q \cup E_0 \cup \{\overline{e_k}\}} <_{\mathsf{mul}} \mathcal{M}_{S_q \cup E}$, then q' satisfies the non-embedding property by the induction hypothesis.

   (d) Otherwise, it holds that e' is partially evaluable and q_i is the last expression in q'' such that $\mathsf{comparable}(q_i, e)$ and $q_i \trianglelefteq^* e$ with $(g, \sigma, \theta) = \mathsf{mslg}(q_i, e')$ and $g \notin \mathcal{V}$. Then for $q''' = q'' \setminus q_i$ and $S = \{\langle\!\langle s \rangle\!\rangle \mid s \in \{g\} \cup \mathcal{R}an(\sigma) \cup \mathcal{R}an(\theta), s \notin \mathcal{V}\}$ we have $q' = \mathsf{abs}^*_{\mathsf{emb}}(q''', S)$. Since q'' satisfies the non-embedding property, so must q''', and since $\mathcal{M}_{S_{q'''} \cup S} <_{\mathsf{mul}} \mathcal{M}_{S_{q'''} \cup \{q_i, e'\}} = \mathcal{M}_{S_{q''} \cup \{e'\}}$ by Lemma B.27, then q' satisfies the non-embedding property by the induction hypothesis.

3. If $e = f(\overline{e_k})$ or $e = \langle\!\langle f(\overline{e_k}) \rangle\!\rangle$ with $f \in \mathcal{F}^{(n)}$ and $k < n$, we distinguish two cases. If there exists a non-variable expression e_i for $i \in \{1, \ldots, k\}$, then the claim follows from the induction hypothesis, otherwise it follows with the same reasoning as for the previous case. □

Bibliography

[AAH+99] Elvira Albert, María Alpuente, Michael Hanus, and Germán Vidal. "A partial evaluation framework for Curry programs". In: *Logic Programming and Automated Reasoning, 6th International Conference, LPAR'99, Tbilisi, Georgia, September 6-10, 1999, Proceedings*. Ed. by Harald Ganzinger, David A. McAllester, and Andrei Voronkov. Vol. 1705. Lecture Notes in Computer Science. Springer, 1999, pp. 376–395.

[AAV00] Elvira Albert, Sergio Antoy, and Germán Vidal. "Measuring the effectiveness of partial evaluation in functional logic languages". In: *Logic Based Program Synthesis and Transformation, 10th International Workshop, LOPSTR 2000 London, UK, July 24-28, 2000, Selected Papers.* Ed. by Kung-Kiu Lau. Vol. 2042. Lecture Notes in Computer Science. Springer, 2000, pp. 103–124.

[AEH00] Sergio Antoy, Rachid Echahed, and Michael Hanus. "A needed narrowing strategy". In: *Journal of the ACM* 47.4 (2000), pp. 776–822.

[AEH97] Sergio Antoy, Rachid Echahed, and Michael Hanus. "Parallel evaluation strategies for functional logic languages". In: *Proceedings of the 14th International Conference on Logic Programming (ICLP'97)*. MIT Press, 1997, pp. 138–152.

[AFI+97] María Alpuente, Moreno Falaschi, Pascual Julián Iranzo, and Germán Vidal. "Specialization of lazy functional logic programs". In: *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '97), Amsterdam, The Netherlands, June 12-13, 1997*. Ed. by John P. Gallagher, Charles Consel, and A. Michael Berman. ACM, 1997, pp. 151–162.

[AFV98] María Alpuente, Moreno Falaschi, and Germán Vidal. "Partial evaluation of functional logic programs". In: *ACM Transactions on Programming Languages and Systems* 20.4 (1998), pp. 768–844.

[AH05] Sergio Antoy and Michael Hanus. "Declarative programming with function patterns". In: *Proceedings of the International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'05)*. Springer LNCS 3901, 2005, pp. 6–22.

[AH06] Sergio Antoy and Michael Hanus. "Overlapping rules and logic variables in functional logic programs". In: *Proceedings of the 22nd International Conference on Logic Programming (ICLP 2006)*. Springer LNCS 4079, 2006, pp. 87–101.

[AH09] Sergio Antoy and Michael Hanus. "Set functions for functional logic programming". In: *Proceedings of the 11th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'09)*. ACM Press, 2009, pp. 73–82.

[AHH+05] Elvira Albert, Michael Hanus, Frank Huch, Javier Oliver, and Germán Vidal. "Operational semantics for declarative multi-paradigm languages". In: *Journal of Symbolic Computation* 40.1 (2005), pp. 795–829.

[AHL+04] María Alpuente, Michael Hanus, Salvador Lucas, and Germán Vidal. "Specialization of functional logic programs based on needed narrowing". In: *Computing Research Repository (CoRR) at* `http://arXiv.org/abs/cs.PL/0403011` (2004).

[AHV00] Elvira Albert, Michael Hanus, and Germán Vidal. "Using an abstract representation to specialize functional logic programs". In: *Proceedings of the 7th International Conference on Logic for Programming and Automated Reasoning (LPAR 2000)*. Springer LNCS 1955, 2000, pp. 381–398.

[AHV02] Elvira Albert, Michael Hanus, and Germán Vidal. "A practical partial evaluator for a multi-paradigm declarative language". In: *Journal of Functional and Logic Programming* 2002.1 (2002).

[AHV03] Elvira Albert, Michael Hanus, and Germán Vidal. "A residualizing semantics for the partial evaluation of functional logic programs". In: *Information Processing Letters* 85.1 (2003), pp. 19–25.

[AJ13] Sergio Antoy and Andy Jost. "Compiling a functional logic language: the fair scheme". In: *Logic-Based Program Synthesis and Transformation, 23rd International Symposium, LOPSTR 2013, Madrid, Spain, September 18-19, 2013, Revised Selected Papers*. Ed. by Gopal Gupta and Ricardo Peña. Vol. 8901. Lecture Notes in Computer Science. Springer, 2013, pp. 202–219.

[ALH+05] María Alpuente, Salvador Lucas, Michael Hanus, and Germán Vidal. "Specialization of functional logic programs based on needed narrowing". In: *Theory and Practice of Logic Programming* 5.3 (2005), pp. 273–303.

[Ant01] Sergio Antoy. "Constructor-based conditional narrowing". In: *Proceedings of the 3rd International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP 2001)*. ACM Press, 2001, pp. 199–206.

[Ant05] Sergio Antoy. "Evaluation strategies for functional logic programming". In: *Journal of Symbolic Computation* 40.1 (2005), pp. 875–903.

[Ant92] Sergio Antoy. "Definitional trees". In: *Proceedings of the 3rd International Conference on Algebraic and Logic Programming*. Ed. by Hélène Kirchner and Giorgio Levi. Volterra, Italy: Springer LNCS 632, Sept. 1992, pp. 143–157.

[Ant97] Sergio Antoy. “Optimal non-deterministic functional logic computations”. In: *Algebraic and Logic Programming, 6th International Joint Conference, ALP '97 - HOA '97, Southampton, U.K., Spetember 3-5, 1997, Proceedings*. Ed. by Michael Hanus, Jan Heering, and Karl Meinke. Vol. 1298. Lecture Notes in Computer Science. Springer, 1997, pp. 16–30.

[AP12] Sergio Antoy and Arthur Peters. “Compiling a functional logic language: the basic scheme”. In: *Functional and Logic Programming - 11th International Symposium, FLOPS 2012, Kobe, Japan, May 23-25, 2012. Proceedings*. Ed. by Tom Schrijvers and Peter Thiemann. Vol. 7294. Lecture Notes in Computer Science. Springer, 2012, pp. 17–31.

[Bar84] Hendrik Pieter Barendregt. *The lambda calculus: its syntax and semantics*. Studies in logic and the foundations of mathematics. North-Holland, 1984.

[BD77] Rod M. Burstall and John Darlington. “A transformation system for developing recursive programs”. In: *Journal of the ACM* 24.1 (1977), pp. 44–67.

[BH05] Bernd Braßel and Michael Hanus. “Nondeterminism analysis of functional logic programs”. In: *Logic Programming, 21st International Conference, ICLP 2005, Sitges, Spain, October 2-5, 2005, Proceedings*. Ed. by Maurizio Gabbrielli and Gopal Gupta. Vol. 3668. Lecture Notes in Computer Science. Springer, 2005, pp. 265–279.

[BHH04] Bernd Braßel, Michael Hanus, and Frank Huch. “Encapsulating nondeterminism in functional logic computations”. In: *Journal of Functional and Logic Programming* 2004.6 (2004).

[BHP+13] Bernd Braßel, Michael Hanus, Björn Peemöller, and Fabian Reck. “Implementing equational constraints in a functional language”. In: *Practical Aspects of Declarative Languages - 15th International Symposium, PADL 2013, Rome, Italy, January 21-22, 2013. Proceedings*. Ed. by Konstantinos F. Sagonas. Vol. 7752. Lecture Notes in Computer Science. Springer, 2013, pp. 125–140.

[BN98] Franz Baader and Tobias Nipkow. *Term rewriting and all that*. Cambridge University Press, 1998.

[BP10] Maximilian C. Bolingbroke and Simon L. Peyton Jones. “Supercompilation by evaluation”. In: *Proceedings of the 3rd ACM SIGPLAN Symposium on Haskell, Haskell 2010, Baltimore, MD, USA, 30 September 2010*. Ed. by Jeremy Gibbons. ACM, 2010, pp. 135–146.

[Bra11] Bernd Braßel. “Implementing functional logic programs by translation into purely functional programs”. PhD thesis. Kiel University, 2011.

[BSM92] Maurice Bruynooghe, Danny De Schreye, and Bern Martens. "A general criterion for avoiding infinite unfolding during partial deduction". In: *New Generation Computing* 11.1 (1992), pp. 47–79.

[CC77] Patrick Cousot and Radhia Cousot. "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints". In: *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL '77. Los Angeles, California: ACM, 1977, pp. 238–252.

[CL11] William R. Cook and Ralf Lämmel. "Tutorial on online partial evaluation". In: *Proceedings IFIP Working Conference on Domain-Specific Languages, DSL 2011, Bordeaux, France, 6-8th September 2011*. Ed. by Olivier Danvy and Chung-chieh Shan. Vol. 66. EPTCS. 2011, pp. 168–180.

[CSV10] Jan Christiansen, Daniel Seidel, and Janis Voigtländer. "Free theorems for functional logic programs". In: *4th Workshop on Programming Languages meets Program Verification, Madrid, Spain, Proceedings*. Ed. by Jean-Christophe Filliâtre and Cormac Flanagan. ACM Press, Jan. 2010, pp. 39–48.

[DJ90] Nachum Dershowitz and Jean-Pierre Jouannaud. "Rewrite systems". In: *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*. 1990, pp. 243–320.

[DM82] Luis Damas and Robin Milner. "Principal type-schemes for functional programs". In: *Proceedings of the 9th Annual Symposium on Principles of Programming Languages*. 1982, pp. 207–212.

[EJ97] Rachid Echahed and Jean-Christophe Janodet. *On constructor-based graph rewriting systems*. Research Report IMAG 985-I. IMAG-LSR, CNRS, Grenoble, 1997.

[FN88] Yoshihiko Futamura and Kenroku Nogi. "Generalized partial computation". In: *Proceedings of the International Workshop on Partial Evaluation and Mixed Computation*. Ed. by Dines Bjørner, Andrei Ershov, and Neil Jones. Amsterdam: North-Holland, 1988, pp. 133–151.

[FST+07] Sebastian Fischer, Josep Silva, Salvador Tamarit, and Germán Vidal. "Preserving sharing in the partial evaluation of lazy functional programs". In: *Logic-Based Program Synthesis and Transformation, 17th International Symposium, LOPSTR 2007, Kongens Lyngby, Denmark, August 23-24, 2007, Revised Selected Papers*. Ed. by Andy King. Vol. 4915. Lecture Notes in Computer Science. Springer, 2007, pp. 74–89.

[Fut71] Yoshihiko Futamura. "Partial evaluation of computation process – an approach to a compiler-compiler". In: *Systems, Computers, Controls* 2.5 (1971), pp. 45–50.

[GHL+99] Juan Carlos González-Moreno, Maria Teresa Hortalá-González, Francisco Javier López-Fraguas, and Mario Rodríguez-Artalejo. "An approach to declarative programming based on a rewriting logic". In: *Journal of Logic Programming* 40.1 (1999), pp. 47–87.

[GJM+96] Robert Glück, Jesper Jørgensen, Bern Martens, and Morten Heine Sørensen. "Controlling conjunctive partial deduction". In: *Programming Languages: Implementations, Logics, and Programs, 8th International Symposium, PLILP'96, Aachen, Germany, September 24-27, 1996, Proceedings*. Ed. by Herbert Kuchen and S. Doaitse Swierstra. Vol. 1140. Lecture Notes in Computer Science. Springer, 1996, pp. 152–166.

[GLP93] Andrew Gill, John Launchbury, and Simon L. Peyton Jones. "A short cut to deforestation". In: *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*. FPCA '93. Copenhagen, Denmark: ACM, 1993, pp. 223–232.

[GS94] Robert Glück and Morten Heine Sørensen. "Partial deduction and driving are equivalent". In: *Programming Language Implementation and Logic Programming, 6th International Symposium, PLILP'94, Madrid, Spain, September 14-16, 1994, Proceedings*. Ed. by Manuel V. Hermenegildo and Jaan Penjam. Vol. 844. Lecture Notes in Computer Science. Springer, 1994, pp. 165–181.

[GS96] Robert Glück and Morten Heine Sørensen. "A roadmap to metacomputation by supercompilation". In: *Partial Evaluation, International Seminar, Dagstuhl Castle, Germany, February 12-16, 1996, Selected Papers*. Ed. by Olivier Danvy, Robert Glück, and Peter Thiemann. Vol. 1110. Lecture Notes in Computer Science. Springer, 1996, pp. 137–160.

[HA77] Matthew Hennessy and Edward A. Ashcroft. "Parameter-passing mechanisms and nondeterminism". In: *Proceedings of the 9th Annual ACM Symposium on Theory of Computing, May 4-6, 1977, Boulder, Colorado, USA*. Ed. by John E. Hopcroft, Emily P. Friedman, and Michael A. Harrison. ACM, 1977, pp. 306–311.

[HAB+15] Michael Hanus, Sergio Antoy, Bernd Braßel, Martin Engelke, Klaus Höppner, Johannes Koj, Philipp Niederau, Björn Peemöller, Ramin Sadre, and Frank Steiner. *PAKCS: the Portland Aachen Kiel Curry System*. Available at `http://www.informatik.uni-kiel.de/~pakcs/`. 2015.

[Han12] Michael Hanus (ed.) *Curry: an integrated functional logic language (vers. 0.8.3)*. Available at `http://www.curry-language.org`. 2012.

[Han13] Michael Hanus. "Functional logic programming: from theory to Curry". In: *Programming Logics - Essays in Memory of Harald Ganzinger*. Springer LNCS 7797, 2013, pp. 123–168.

[Han94] Michael Hanus. "The integration of functions into logic programming: from theory to practice". In: *Journal of Logic Programming* 19/20 (1994), pp. 583–628.

[HBP+15] Michael Hanus, Bernd Braßel, Björn Peemöller, and Fabian Reck. *KiCS2: the Kiel Curry System (version 2)*. Available at `http://www-ps.informatik.uni-kiel.de/kics2/`. 2015.

[HHW15] Zhenjiang Hu, John Hughes, and Meng Wang. "How functional programming mattered". In: *National Science Review* (2015). eprint: `http://nsr.oxfordjournals.org/content/early/2015/08/24/nsr.nwv042.full.pdf+html`.

[HL91] Gérard P. Huet and Jean-Jacques Lévy. "Computations in orthogonal rewriting systems". In: *Computational Logic: Essays in Honor of Alan Robinson*. Ed. by Jean-Louis Lassez and Gordon D. Plotkin. MIT Press, 1991, pp. 395–443.

[HP14] Michael Hanus and Björn Peemöller. "A partial evaluator for Curry". In: *Proceedings of the 23rd International Workshop on Functional and (Constraint) Logic Programming (WFLP 2014)*. Universität Halle-Wittenberg, 2014, pp. 55–71.

[HP99] Michael Hanus and Christian Prehofer. "Higher-order narrowing with definitional trees". In: *Journal of Functional Programming* 9.1 (1999), pp. 33–75.

[HPR12] Michael Hanus, Björn Peemöller, and Fabian Reck. "Search strategies for functional logic programming". In: *Proceedings of the 5th Working Conference on Programming Languages (ATPS'12)*. Springer LNI 199, 2012, pp. 61–74.

[HS14] Michael Hanus and Fabian Skrlac. "A modular and generic analysis server system for functional logic programs". In: *Proceedings of the ACM SIGPLAN 2014 Workshop on Partial Evaluation and Program Manipulation*. PEPM '14. San Diego, California, USA: ACM, 2014, pp. 181–188.

[Hud89] Paul Hudak. "Conception, evolution, and application of functional programming languages". In: *ACM Computing Surveys* 21.3 (1989), pp. 359–411.

[Hug89] John Hughes. "Why Functional Programming Matters". In: *Computer Journal* 32.2 (1989), pp. 98–107.

[Hus92] Heinrich Hussmann. "Nondeterministic algebraic specifications and nonconfluent term rewriting". In: *Journal of Logic Programming* 12 (1992), pp. 237–255.

[JGS93] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial evaluation and automatic program generation*. Prentice Hall international series in computer science. Prentice Hall, 1993.

[Joh85] Thomas Johnsson. “Lambda lifting: transforming programs to recursive functions”. In: *Functional Programming Languages and Computer Architecture*. Springer LNCS 201, 1985, pp. 190–203.

[Kah87] Gilles Kahn. “Natural semantics”. In: *STACS 87, 4th Annual Symposium on Theoretical Aspects of Computer Science, Passau, Germany, February 19-21, 1987, Proceedings*. Ed. by Franz-Josef Brandenburg, Guy Vidal-Naquet, and Martin Wirsing. Vol. 247. Lecture Notes in Computer Science. Springer, 1987, pp. 22–39.

[KH91] Carsten Kehler Holst and John Hughes. “Towards binding-time improvement for free”. English. In: *Functional Programming, Glasgow 1990*. Ed. by Simon L. Peyton Jones, Graham Hutton, and Carsten Kehler Holst. Workshops in Computing. Springer London, 1991, pp. 83–100.

[Klo92] Jan Willem Klop. “Handbook of logic in computer science (vol. 2)”. In: ed. by S. Abramsky, Dov M. Gabbay, and S. E. Maibaum. New York, NY, USA: Oxford University Press, Inc., 1992. Chap. Term Rewriting Systems, pp. 1–116.

[KMP77] Donald E. Knuth, James H. Morris Jr., and Vaughan R. Pratt. “Fast pattern matching in strings”. In: *SIAM Journal on Computing* 6.2 (1977), pp. 323–350.

[KR10] Ilya Klyuchnikov and Sergei Romanenko. “Towards higher-level supercompilation”. In: *Second International Workshop on Metacomputation in Russia*. Ailamazyan University of Pereslavl, 2010, pp. 82–101.

[Lau93] John Launchbury. “A natural semantics for lazy evaluation”. In: *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’93. Charleston, South Carolina, USA: ACM, 1993, pp. 144–154.

[Leu98] Michael Leuschel. “On the power of homeomorphic embedding for online termination”. In: *Static Analysis, 5th International Symposium, SAS ’98, Pisa, Italy, September 14-16, 1998, Proceedings*. Ed. by Giorgio Levi. Vol. 1503. Lecture Notes in Computer Science. Springer, 1998, pp. 230–245.

[LMM88] Jean-Louis Lassez, Michael J. Maher, and Kim Marriott. “Unification revisited”. In: *Foundations of Deductive Databases and Logic Programming*. Ed. by Jack Minker. Morgan Kaufmann, 1988, pp. 587–625.

[LMS98] Michael Leuschel, Bern Martens, and Danny De Schreye. “Controlling generalization amd polyvariance in partial deduction of normal logic programs”. In: *ACM Transactions on Programming Languages and Systems* 20.1 (1998), pp. 208–258.

[LRS07a] Francisco Javier López-Fraguas, Juan Rodríguez-Hortalá, and Jaime Sánchez-Hernández. "A simple rewrite notion for call-time choice semantics". In: *Proceedings of the 9th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*. PPDP '07. Wroclaw, Poland: ACM, 2007, pp. 197–208.

[LRS07b] Francisco Javier López-Fraguas, Juan Rodríguez-Hortalá, and Jaime Sánchez-Hernández. "Narrowing for first order functional logic programs with call-time choice semantics". In: *Applications of Declarative Programming and Knowledge Management, 17th International Conference, INAP 2007, and 21st Workshop on Logic Programming, WLP 2007, Würzburg, Germany, October 4-6, 2007, Revised Selected Papers*. Ed. by Dietmar Seipel, Michael Hanus, and Armin Wolf. Vol. 5437. Lecture Notes in Computer Science. Springer, 2007, pp. 206–222.

[LS91] John Wylie Lloyd and John C. Shepherdson. "Partial evaluation in logic programming". In: *Journal of Logic Programming* 11 (1991), pp. 217–242.

[LS99] Francisco Javier López-Fraguas and Jaime Sánchez-Hernández. "TOY: a multiparadigm declarative system". In: *Proceedings of RTA'99*. Springer LNCS 1631, 1999, pp. 244–247.

[Mar10] Simon Marlow, ed. *Haskell 2010 language report*. 2010.

[McC61] John McCarthy. "A basis for a mathematical theory of computation, preliminary report". In: *Papers Presented at the May 9-11, 1961, Western Joint IRE-AIEE-ACM Computer Conference*. IRE-AIEE-ACM '61 (Western). Los Angeles, California: ACM, 1961, pp. 225–238.

[MG95] Bern Martens and John P. Gallagher. "Ensuring global termination of partial deduction while allowing flexible polyvariance". In: *Logic Programming, Proceedings of the Twelfth International Conference on Logic Programming, Tokyo, Japan, June 13-16, 1995*. Ed. by Leon Sterling. MIT Press, 1995, pp. 597–611.

[Mit10] Neil Mitchell. "Rethinking supercompilation". In: *Proceeding of the 15th ACM SIGPLAN international conference on Functional programming, ICFP 2010, Baltimore, Maryland, USA, September 27-29, 2010*. Ed. by Paul Hudak and Stephanie Weirich. ACM, 2010, pp. 309–320.

[MSS+14] Stefan Mehner, Daniel Seidel, Lutz Straßburger, and Janis Voigtländer. "Parametricity and proving free theorems for functional-logic languages". In: *Proceedings of the 16th International Symposium on Principles and Practice of Declarative Programming*. PPDP '14. Canterbury, United Kingdom: ACM, 2014, pp. 19–30.

[Pet77] Alberto Pettorossi. “Transformation of programs and use of tupling strategy”. In: *Proceedings of Informatica '77*. Bled, Yugoslavia, 1977, pp. 1–6.

[PM02] Simon L. Peyton Jones and Simon Marlow. “Secrets of the glasgow haskell compiler inliner”. In: *J. Funct. Program.* 12.4&5 (2002), pp. 393–433.

[PP94] Alberto Pettorossi and Maurizio Proietti. “Special issue: ten years of logic programming transformation of logic programs: foundations and techniques”. In: *The Journal of Logic Programming* 19 (1994), pp. 261–320.

[PPS96] Simon L. Peyton Jones, Will Partain, and André Santos. “Let-floating: moving bindings to give faster programs”. In: *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming (ICFP '96), Philadelphia, Pennsylvania, May 24-26, 1996*. Ed. by Robert Harper and Richard L. Wexelblat. ACM, 1996, pp. 1–12.

[PW93] Simon L. Peyton Jones and Philip Wadler. “Imperative functional programming”. In: *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '93. Charleston, South Carolina, USA: ACM, 1993, pp. 71–84.

[Rey72] John C. Reynolds. “Definitional interpreters for higher-order programming languages”. In: *Proceedings of the ACM Annual Conference - Volume 2*. ACM '72. Boston, Massachusetts, USA: ACM, 1972, pp. 717–740.

[Ses85] Peter Sestoft. “The structure of a self-applicable partial evaluator”. In: *Programs as Data Objects, Proceedings of a Workshop, Copenhagen, Denmark, October 17-19, 1985*. Ed. by Harald Ganzinger and Neil D. Jones. Vol. 217. Lecture Notes in Computer Science. Springer, 1985, pp. 236–256.

[Ses97] Peter Sestoft. “Deriving a lazy abstract machine”. In: *Journal of Functional Programming* 7.3 (1997), pp. 231–264.

[SG95] Morten Heine Sørensen and Robert Glück. “An algorithm of generalization in positive supercompilation”. In: *Proceedings of the 1995 International Logic Programming Symposium*. MIT Press, 1995, pp. 465–479.

[SGJ96] Morten Heine Sørensen, Robert Glück, and Neil D. Jones. “A positive supercompiler”. In: *Journal of Functional Programming* 6.6 (1996), pp. 811–838.

[Sla74] James R. Slagle. “Automated theorem-proving for theories with simplifiers, commutativity, and associativity”. In: *Journal of the ACM* 21.4 (1974), pp. 622–642.

[SS99] Jens P. Secher and Morten Heine Sørensen. “On perfect supercompilation”. In: *Perspectives of System Informatics, Third International Andrei Ershov Memorial Conference, PSI'99, Akademgorodok, Novosibirsk, Russia, July 6-9, 1999, Proceedings*. Ed. by Dines Bjørner, Manfred Broy, and Alexandre V. Zamulin. Vol. 1755. Lecture Notes in Computer Science. Springer, 1999, pp. 113–127.

[Tur86] Valentin F. Turchin. “The concept of a supercompiler”. In: *ACM Transactions on Programming Languages and Systems* 8.3 (1986).

[Tur96] Valentin F. Turchin. “Metacomputation: metasystem transitions plus supercompilation”. In: *Partial Evaluation, International Seminar, Dagstuhl Castle, Germany, February 12-16, 1996, Selected Papers*. Ed. by Olivier Danvy, Robert Glück, and Peter Thiemann. Vol. 1110. Lecture Notes in Computer Science. Springer, 1996, pp. 481–509.

[Wad71] Christopher Peter Wadsworth. “Semantics and pragmatics of the lambda calculus”. PhD thesis. University of Oxford, 1971.

[Wad87] Philip Wadler. “Efficient compilation of pattern-matching”. In: *The Implementation of Functional Programming Languages*. Ed. by Simon L. Peyton Jones. Prentice Hall, 1987, pp. 78–103.

[Wad89] Philip Wadler. “Theorems for free!” In: *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*. FPCA '89. Imperial College, London, United Kingdom: ACM, 1989, pp. 347–359.

[Wad90] Philip Wadler. “Deforestation: transforming programs to eliminate trees”. In: *Theoretical Computer Science* 73.2 (1990), pp. 231–248.

Index