INSTITUT FÜR INFORMATIK

Programmiersprachen und Rechenkonzepte

25. Workshop der GI-Fachgruppe „Programmiersprachen und Rechenkonzepte"

Bad Honnef, 5.-7. Mai 2008

Michael Hanus, Sebastian Fischer (Hrsg.)

Bericht Nr. 0811

Oktober 2008

CHRISTIAN-ALBRECHTS-UNIVERSITÄT

KIEL

Institut für Informatik der
Christian-Albrechts-Universität zu Kiel
Olshausenstr. 40
D – 24098 Kiel

Programmiersprachen und Rechenkonzepte

25. Workshop der GI-Fachgruppe „Programmiersprachen und Rechenkonzepte“ Bad Honnef, 5.-7. Mai 2008

Michael Hanus, Sebastian Fischer (Hrsg.)

Bericht Nr. 0811
Oktober 2008

e-mail: mh@informatik.uni-kiel.de,
sebf@informatik.uni-kiel.de

Dieser Bericht enthält eine Zusammenstellung der Beiträge des 25. Workshops Programmiersprachen und Rechenkonzepte, Physikzentrum Bad Honnef, 5.-7. Mai 2008.

Vorwort

Seit 1984 veranstaltet die GI–Fachgruppe „Programmiersprachen und Rechenkonzepte“, die aus den ehemaligen Fachgruppen 2.1.3 „Implementierung von Programmiersprachen“ und 2.1.4 „Alternative Konzepte für Sprachen und Rechner“ hervorgegangen ist, regelmäßig im Frühjahr einen Workshop im Physikzentrum Bad Honnef. Das Treffen dient in erster Linie dem gegenseitigen Kennenlernen, dem Erfahrungsaustausch, der Diskussion und der Vertiefung gegenseitiger Kontakte.

In diesem Forum werden Vorträge und Demonstrationen sowohl bereits abgeschlossener als auch noch laufender Arbeiten vorgestellt, unter anderem (aber nicht ausschließlich) zu Themen wie

- Sprachen, Sprachparadigmen
- Korrektheit von Entwurf und Implementierung
- Werkzeuge
- Software-/Hardware-Architekturen
- Spezifikation, Entwurf
- Validierung, Verifikation
- Implementierung, Integration
- Sicherheit (Safety und Security)
- eingebettete Systeme
- hardware-nahe Programmierung

In diesem Technischen Bericht sind die präsentierten Arbeiten des diesjährigen Workshops zusammen gestellt. Allen Teilnehmern des Workshops möchten wir danken, dass sie durch ihre Vorträge, Papiere und Diskusion den jährlichen Workshop zu einem interessanten Ereignis machen. Abschließend danken wir auch den Mitarbeitern des Physikzentrums Bad Honnef, die durch ihre umfassende Betreuung für eine angenehme und anregende Atmosphäre gesorgt haben.

Kiel, im Oktober 2008 — Michael Hanus, Sebastian Fischer

Inhaltsverzeichnis

Task Parallel Skeletons for Divide and Conquer
Michael Poldner 1

Extended Exceptions for Contingencies
Thorsten van Ellen 19

Männliche und weibliche Doppelstudenten: ein Härtetest für Programmiersprachen
Christian Heinlein 33

Statisches Typen von JavaScript Programmen
Phillip Heidegger 43

Typinferenz für Java(X)
Markus Degen 44

Resolving of Intersection Types in Java
Martin Plümicke 45

Graph Parser Combinators: A Challenge for Curry-Compilers
Steffen Mazanek 55

Adding Haskell-stlye Overloading to Curry
Wolfgang Lux 67

Ach, wie gut, dass niemand weiß, dass ich dreizehnsiebtel heiß
Jan Christiansen 77

A Debugger for Functional Logic Languages
Bernd Braßel 78

WCET Annotation Languages Reconsidered: The Annotation Language Challenge
Jens Knoop 93

Towards a Common WCET Annotation Language: Essential Ingredients
Albrecht Kadlec 104

TuBound - A Tool for Worst-Case Execution Time Analysis
Adrian Prantl 117

Applying the Component Paradigm to AUTOSAR Basic Software
Dietmar Schreiner 127

An Example of Source-To-Source Analysis with SATIrE
Markus Schordan 133

Fundamente der Programmierung
Hermann von Issendorff 142

Demonstrably Correct Compilation of Java Bytecode
Michael Leuschel 143

Modellreduktionstechniken für symbolische Kellersysteme
Dirk Richter 144

Wortbasierte Symbolische Simulation Hybrider Systeme in CLP
Elke Tetzner 154

Datenfluss in bedarfsgesteuerten Berechnungen
Sebastian Fischer 163

Empirischer Vergleich von Tools für das Testen deklarativer Programme
Herbert Kuchen 165

A Summary Function Model for the Validation of Interprocedural Analysis Results
Karsten Klohs 166

Parametricity for Haskell with Imprecise Error Semantics
Janis Voigtländer 177

The expression lemma
Ralf Lämmel 187

Stackless Stack Inspection – Portabler Fluchtweg aus dem Teufelskreis
Baltasar Trancón y Widemann 188

Task Parallel Skeletons for Divide and Conquer

Michael Poldner and Herbert Kuchen

Department of Information Systems, University of Münster
Leonardo Campus 3, D-48149 Münster
{poldner,kuchen}@uni-muenster.de

Abstract. Algorithmic skeletons intend to simplify parallel programming by providing recurring forms of program structure as predefined components. We present a fully distributed task parallel skeleton for a very general class of divide and conquer algorithms for MIMD machines with distributed memory. This approach is compared to a simple master-worker design. Based on experimental results for different example applications such as Mergesort, the Karatsuba multiplication algorithm and Strassen's algorithm for matrix multiplication, we show that the distributed workpool enables good runtimes and in particular scalability. Moreover, we discuss some implementation aspects for the distributed skeleton, such as the underlying data structures and load balancing strategy, in detail. In addition, we present another distributed skeleton which benefits from combining skeletal internal parallelism and stream parallelism. Based on experimental results for matrix chain multiplication problems, we show that this approach enables a better processor load and memory utilization for the engaged solvers, and reduces communication costs.

Key words: Algorithmic Skeletons, parallelism, divide and conquer, stream processing

1 Introduction

Parallel programming of MIMD machines with distributed memory is typically based on standard message passing libraries such as MPI [24], which leads to platform independent and efficient software. However, the programming level is still rather low and thus error-prone and time consuming. Programmers have to fight against low-level communication problems such as deadlocks, starvation, and termination detection. Moreover, the program is split into a set of processes which are assigned to the different processors, whereas each process only has a local view of the overall activity. A global view of the overall computation only exists in the programmer's mind, and there is no way to express it more directly on this level. For this reason many approaches have been suggested, which provide a higher level of abstraction and an easier program development. The skeletal approach to parallel programming proposes that typical communication and computation patterns for parallel programming should be offered to the user as predefined and application independent components, which can be combined

and nested by the user. These components are referred to as algorithmic skeletons [1, 7, 10, 18, 21, 23, 26]. Typically, algorithmic skeletons are offered to the user as higher-order functions, which get the details of the specific application problem as argument functions. In this way the user can adapt the skeletons to the considered parallel application without bothering about low-level implementation details such as synchronization, interprocessor communication, load balancing, and data distribution. Efficient implementations of many skeletons exist, such that the resulting parallel application can be almost as efficient as one based on low-level message passing.

Depending on the kind of parallelism used, algorithmic skeletons can roughly be classified into data parallel and task parallel ones. Data parallel skeletons [5, 21, 22] process a distributed data structure such as a distributed array or matrix as a whole, e.g. by applying a function to every element or by rotating or permuting its elements. Task-parallel skeletons [3, 9, 18, 21, 27–30] construct a system of processes communicating via streams of data. Such a system is mostly generated by nesting typical building blocks such as farms and pipelines. In the present paper, we will consider task-parallel skeletons for divide and conquer problems.

Divide and conquer is a common computation paradigm, in which the solution to a problem is obtained by dividing the original problem into smaller subproblems and solving the subproblems recursively. Then, solutions for the subproblems must be combined to form the final solution of the entire problem. A simple problem is solved directly without dividing it further. Examples of divide and conquer computations include various sorting methods such as mergesort and quicksort, computational geometry algorithms such as the construction of the convex hull, combinatorial search such as constraint satisfaction techniques, graph algorithmic problems such as graph coloring, numerical methods such as the Karatsuba multiplication algorithm, and linear algebra such as Strassen's algorithm for matrix multiplication.

In the present paper we will consider different design, implementation, and optimization issues of task parallel divide and conquer skeletons in the context of the skeleton library *Muesli* [21, 27–30]. *Muesli* is build on top of MPI [24] in order to inherit its platform independence. We will show that a master-worker design is less suited to handle divide and conquer problems on distributed memory machines. We have implemented a distributed scheme and present its functionality in detail. We will show that we can achieve a good load balance while minimizing communication costs. This is supported by several test results of three example applications. Moreover, we discuss how to optimize the skeleton for processing streams of divide and conquer problems.

The rest of this paper is structured as follows. In Section 2, we introduce different designs of divide and conquer skeletons in the framework of the skeleton library *Muesli*. Initially, we briefly describe a simple centralized design. Afterwards we will focus on a new fully distributed D&C-Skeleton. Moreover, we discuss how the distributed D&C-Skeleton can be optimized for processing streams of divide and conquer problems. Section 3 contains experimental results demon-

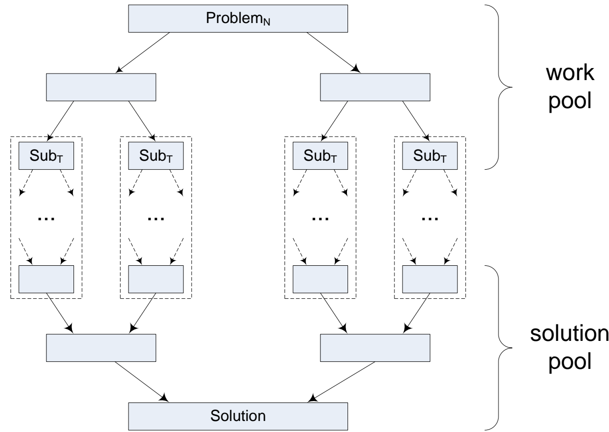

Fig. 1. A divide and conquer tree

strating the strength of the distributed design. In addition, selected experimental results for the stream optimized skeleton are presented. In Section 4 we compare our approach to related work. In Section 5, we conclude and point out future work.

2 Divide and Conquer Skeletons

A divide and conquer skeleton is based on an implementation scheme for divide and conquer and offers it to the user as predefined parallel component. Typically, the user has to provide the skeleton with four basic operators: `divide`, `combine`, `isSimple`, and `solve`. If `isSimple` indicates that a problem is simple enough, it can be solved directly by applying `solve`. Otherwise, the problem is divided into subproblems by calling `divide`. Solutions of subproblems can be combined to the solution of the corresponding parent problem by applying `combine`.

The computation can be viewed as a process of expanding and shrinking a tree, in which the nodes represent problem instances and partial solutions, respectively (Fig.1). Unprocessed subproblems are stored in a work pool and partial solutions are maintained in a solution pool. In the beginning the work pool only contains the initial problem, which is of size N, and the solution pool is empty. In each iteration one such problem is selected from the workpool corresponding to a particular traversal strategy such as depth first or breadth first. The problem is either divided into d subproblems, which are stored again in the workpool, or it is solved, and its solution is stored in the solution pool. It may happen that a problem of size s is reduced to d subproblems of sizes $s_1, \ldots, s_d$ with $\sum_{i=1}^{d} s_i > s$, e.g. for the Karatsuba or Strassen algorithm. At least in this case, a depth first strategy is recommended in order to avoid memory problems.

The order in which solutions are stored in the solution pool depends on the implemented traversal strategy. It is recommended to combine partial solutions

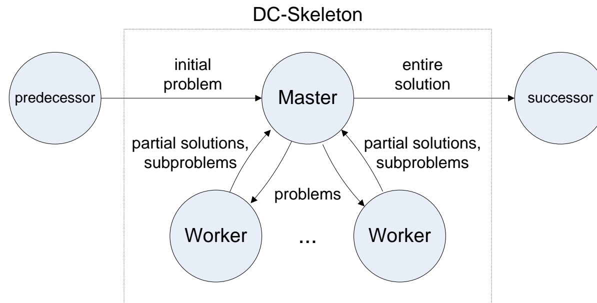

Fig. 2. A master/worker design

as soon as possible in order to free memory. If the solution pool contains d partial solutions, which can be combined, they can be replaced by the solution of the corresponding parent problem. In the end of the computation the workpool is empty and the solution pool only contains the solution of the initial problem.

2.1 Master/Worker design

The simplest approach to implement a divide and conquer skeleton is a kind of the master/worker design as depicted in Figure 2. This approach has been used in [1]. The work pool and the solution pool are maintained by the master, which distributes problems to the workers and receives subproblems and solutions from them. When a worker receives a problem, it either solves it or decomposes it into subproblems. The advantage of a single work and solution pool is that it provides a good overall picture of the work still to be done. Moreover, the master knows about all idle workers at any time, which makes it easy to provide each worker with work. The disadvantage is, that accessing the work pool and the solution pool tends to be a bottleneck, as the pools can only be accessed by one worker at a time. This may result in high idle times on the workers' site. Another disadvantage is that the master/worker approach incurs high communication costs, since each subproblem is sent from its producer to the master and propagated to its processing worker. Moreover, the communication time required to send a problem to a worker and to receive in return some subproblems or a solution may be greater than the time needed to do the computation locally. The master's limited memory capacity for maintaining the problems and solutions is another disadvantage of this architecture. As we have shown in [27, 28], a master/worker design is less suited for farms and branch and bound skeletons. Thus, it can be expected that a master/worker design is badly suited to divide and conquer skeletons as well. For this reason, this approach is not considered any further. A promising approach is a fully distributed D&C Skeleton, which is discussed in the following.

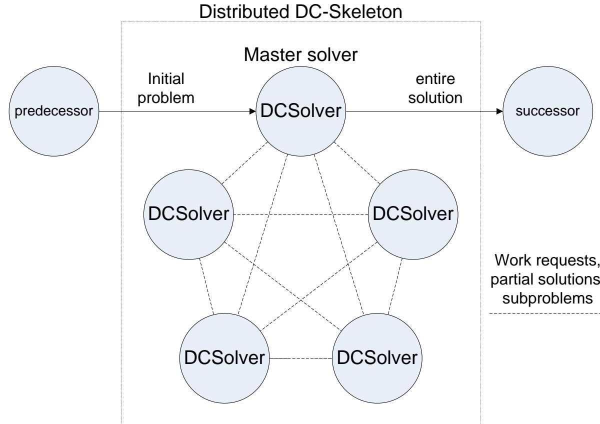

Fig. 3. A distributed design

2.2 A fully distributed D&C-Skeleton

Figure 3 illustrates the design of the distributed divide and conquer skeleton (DCSkeleton) provided by the *Muesli* skeleton library. It consists of a set of peer solvers, which exchange subproblems, partial solutions, and work requests. Several topologies for connecting the solvers are possible. In our implementation, the topology for connecting the solvers is exchangeable without having to adapt the load balancing or termination detection algorithm. To simplify matters in this paper, we will consider an all-to-all topology. For larger numbers of processors, topologies like torus or hypercube may reduce the communication overhead.

In the example shown in Fig. 3, $n = 5$ solvers are used. Each solver maintains its own local work pool and solution pool. Thus, the work and the solution pool are distributed among the solvers, which enables the skeleton to process D&C problems with much higher memory requirements compared to the ones that can be solved by a skeleton based on a master/worker design. Exactly one of the solvers, the *master solver*, serves as an interface to the DCSkeleton. The *master solver* receives new divide and conquer problems from the predecessor and delivers the solutions to its successor. The code fragment in Figure 4 illustrates the application of our DCSkeleton in the context of the *Muesli* skeleton library. It constructs the process topology shown in Fig. 3.

In a first step the process topology is created using C++ constructors. The process topology consists of an `initial` process, a `dc` process, and a `final` process connected by a `pipeline` skeleton. The `initial` process is parameterized by a `generateProblem` method returning the initial D&C problem that is to be solved. The constructor `DistributedDC` generates $n = 5$ solvers, which are provided with the four basic operators `divide`, `combine`, `solveSeq`, and `isSimple`. The function `isSimple` has to return `true` if the subproblem size has reached the threshold T, which indicates that the subproblem can be solved sequentially with `solveSeq`. If only one solver is used, it is recommended to enable

```
int main(int argc, char* argv[]) {
    InitSkeletons(argc,argv);
    // step 1: create process topology
    Initial<Problem> initial(generateProblem);
    DistributedDC<Problem,Solution>
         dc(divide, combine, solveSeq, isSimple, d, 5);
    Final<Problem> final(fin);
    Pipe pipe(initial,dc,final);
    // step 2: start process topology
    pipe.start();
    TerminateSkeletons();
}
```

Fig. 4. Example application using a distributed divide and conquer skeleton.

a purely sequential computation by setting T to the size of the initial problem. The parameter d corresponds to the degree of the D&C tree and describes how many subproblems are generated by `divide` and how many subproblems are required by `combine` to generate the solution of the corresponding parent problem.

The DC-skeleton consumes a stream of input values and produces a stream of output values. If the *master solver* receives a new D&C problem, the communication with the predecessor is blocked until the received problem is solved. This ensures that the skeleton processes only one D&C problem at a time.

There are different variants for the initialization of the skeleton with the objective of providing each `DCSolver` with a certain amount of work within the startup phase. Our skeleton uses the most common approach, namely *root initialization*, i.e. the initial D&C problem is inserted into the local work pool of the *master solver*. Subproblems are distributed according to the load balancing scheme applied by the solvers.

Each solver repeatedly executes two overlapping phases: a communication phase and a computation phase. The communication phase includes work requests, delegating problems, and sending solutions. Let us first consider the case in which each solver works locally on the work and solution pool, and no communication due to load balancing issues is needed.

The work pool and the solution pool are implemented in their own classes in order to allow an easy replacement of the underlying data structures and algorithms for e.g. the traversal strategy. In our skeleton, the work pool is implemented as a double ended queue (DEQ). Local work is taken from and written to the head, whereas problems which are delegated within the load distribution are taken from the tail. Thus, the DEQ behaves as a stack for local computations. The solution pool is implemented as a sorted array. If there are only local computations (as it is the case most of the time), it behaves very efficiently as a stack. Thus, we have preferred this solution to a heap.

Subproblems and partial solutions are encapsulated in *Frames*, which are each identified by a unique identifier. Subproblems and their corresponding solutions are marked with the same *id*. The initial problem is marked with $id = 0$. If the solution pool contains a solution with $id = 0$, this indicates that the initial problem has been solved. This problem based termination detection is independent

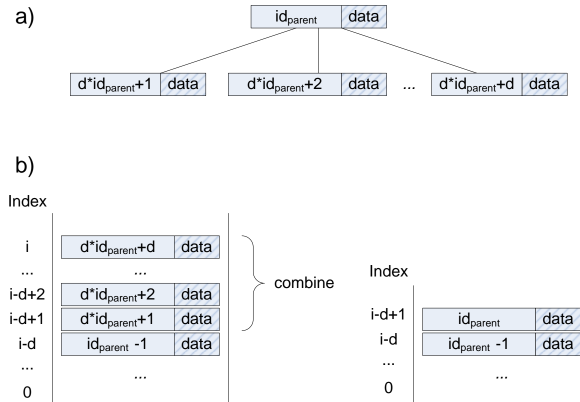

Fig. 5. a) A problem is divided into d subproblems; b) Solution pool represented as sorted array before (left) and after applying combine (right).

from the topology which is used for connecting the solvers. Moreover, it does not need any communication. The ids for parent nodes, and child nodes respectively, can for instance be deduced from the formula $parentNodeID = \frac{childNodeID-1}{d}$, with d equal to the degree of the divide and conquer tree (e.g. number of child nodes).

The solver takes and processes only one problem from the work pool per iteration. If a problem is divided by the solver, d new subproblems are generated which are marked with ids in ascending order (Fig.5a). These subproblems are successively inserted into the workpool again, starting with the subproblem tagged with the largest id. The subproblem marked with the lowest id is inserted last. Figure 6 illustrates the status of the workpool after the first four iterations with $d = 2$.

If a problem can be directly solved by the solver, the corresponding solution is written to the solution pool. The partial solutions stored in the solution pool are kept sorted by the ids in ascending order. Thus, whenever a solution is pushed to the solution pool, in a first step, the solution is written to the end of the sorted list, and then, a simple insertion sort is applied to preserve the order, if necessary.

Keeping the solution pool sorted enables a fast combination of partial solutions due to the fact that only two elements of the solution pool have to be inspected in order to detect if the top d elements can be combined. Assuming that a problem marked with the identifier id_{parent} is divided into d subproblems, as shown in Figure 5a, the leftmost child node is marked with $d \cdot id_{parent} + 1$, and the rightmost subproblem is marked with $d \cdot id_{parent} + d$. After the subproblems are solved, the solution pool contains solutions which are marked with the same ids than the subproblems. The solution of the most right child of id_{parent} is stored at index position i and represents the top element of the stack (Fig. 5b). The top d stack elements can be combined if they emanate from the same

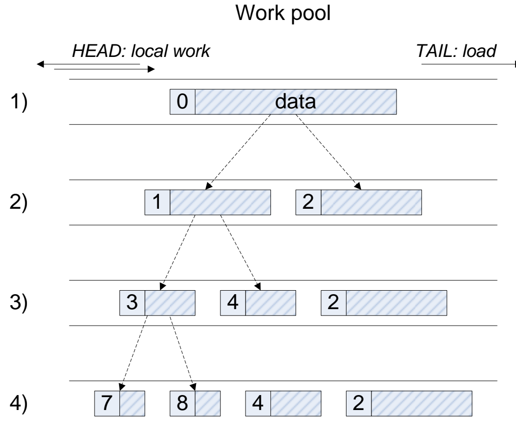

Fig. 6. States of the work pool

parent problem. This is the case iff the subproblem id at index position $i-d+1$ is $d-1$ less than the id at index i. Otherwise, one or more partial solutions are still missing, and `combine` can not be applied yet. If a combination is possible, the partial solutions are removed from, and the solution of the parent problem is written to the solution pool (Fig. 5b). This event triggers another inspection of the top d stack elements and a `combine` call, if applicable.

Let us now consider the communication phase. If several solvers are used, some load balancing mechanism is required. We have implemented a *random stealing* algorithm, which is provably efficient for divide and conquer algorithms in terms of space, time, and communication [11, 32]. In our case, a solver sends a work request to a randomly selected neighbour if its work pool is empty. If the neighbour has more than one subproblem in its work pool, it takes one (from the tail of the `DEQ`) and returns it to the requestor. Otherwise it sends a rejection message, which causes the requestor to ask another neighbour for some work to share. The advantage is that no communication is performed until a solver finds its own work pool empty, and the system behaves well under high loads. Moreover, problems which are taken from the tail of the `DEQ` are expected to be big since they stem from nodes of the upper levels of the divide and conquer tree. Thus, the receiver is supplied with a large amount of work. Obtained problems are processed locally in the same manner as described above. The corresponding solution must be sent back to the neighbour due to the fact that it is required there by `combine`. For this reason the solver records a pair $(problemID, neighbour)$ for each new problem which is received. If a new solution is combined whose id is equal to a previously recorded $problemID$, it is sent back to the corresponding *neighbour*, which stores it in the solution pool. Normally, this triggers an insertion sort call in order to restore the order by id. As a result, there may be combinable solutions deep in the stack, which are not combined immediately. Sooner or later, these solutions are inevitably combined

due to the implemented traversal strategy. However, in order to avoid idle times when the work pool is empty and new work is requested, a solver searches the solution pool for combinable solutions and combines them, if possible.

2.3 Optimizing the DCSkeleton for streams

MPI is internally based on a two-level communication protocol, the eager protocol for sending messages less than $32KB$ and the rendezvous protocol for larger messages [29]. For the asynchronous eager protocol the assumption is made that the receiving process can store the message if it is sent and no receive operation has been posted by the receiver. In this case the receiving process must provide a certain amount of buffer space to buffer the message upon its arrival. In contrast to the eager protocol, the rendezvous protocol writes the data directly to the receive buffer without intermediate buffering. This synchronous protocol requires an acknowledgment from a matching receive in order for the send operation to transmit the data. This protocol leads to a higher bandwidth but also to a higher latency due to the necessary handshaking between sender and receiver. In the following we assume problem sizes greater than $32KB$, such as multiplying at least two 64×64 integer matrices, which enables the rendezvous protocol. Moreover, we act on the assumption that the time between the arrivals of two problems is less than the time for solving it sequentially. Otherwise we are not able to speed up the overall computation because the divide and conquer skeleton cannot be a bottleneck of the process system.

Considering task parallel process systems, two forms of parallelism can be identified. The first one is the skeletal internal parallelism, which follows from processing one single problem by several workers in parallel. The DCSkeleton benefits from skeletal internal parallelism by solving one problem by all engaged solvers in parallel. The second form of parallelism is stream parallelism, which follows from the possibility of splitting up one data stream into many streams and processing these streams in parallel. Somewhere in the process system these streams have to be routed to a common junction point in order to reunite them again. The farm topology depicted in figure 7, which is offered by the *Muesli* skeleton library, benefits from stream parallelism. Each worker of the farm takes a new problem from its own stream, so that several problems can be processed independently from each other within the farm at the same time. In this paper, we consider a farm of DCSkeletons which are each configured to a purely sequential computation as described above.

Many applications require solving several divide and conquer problems in sequence. Examples here are the 2D or 3D triangulation of several geometric objects, matrix chain multiplication problems, in which parts of the chain can be computed independently from each other, or factoring of several large numbers. Using the *Muesli* skeleton library, the different tasks can be represented as a stream, which is routed to either a single DCSkeleton (fig. 3) or a farm (fig. 7).

The DCSkeleton processes one divide and conquer problem by N solvers at a time, while the fully distributed memory is available for the solution process. Figure 8 depicts the utilization of the DCSkeleton within the startup, main,

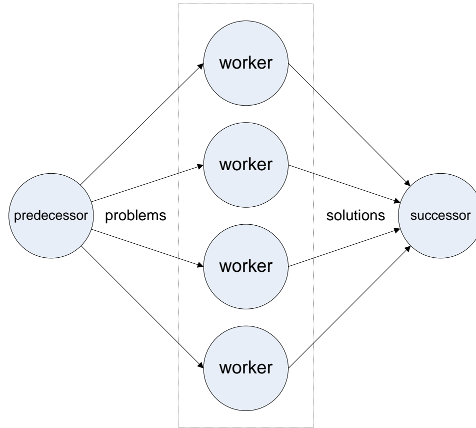

Fig. 7. A farm skeleton

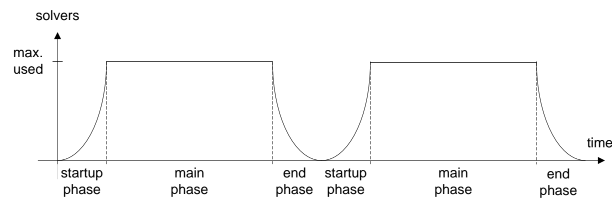

Fig. 8. utilization of the DCSkeleton

and end phase of solving such a problem. In the beginning, a certain amount of work has to be generated by divide calls and distributed among all solvers until each solver is provided with work. For this reason, this skeleton shows high idle times within the startup phase. In particular when the initial problem is divided by the master solver, all remaining solvers are idle. Within the main phase of the computation all solvers are working to full capacity. Moreover, this phase is characterized by low communication costs due to the fact that the solvers predominantly work on their local pools, which is essential to achieve good speedups. Within the end phase, partial solutions have to be collected and combined to parent solutions. At the end, only the master solver combines the entire solution, and all other solvers are idle. Thus, the end phase is characterized by high idle times as well. The duration of the startup and end phase results from both, the complexity of dividing problems and combining partial solutions, and the sizes of the subproblems and partial solutions which have to be sent over the network.

Processing streams of divide and conquer problems can be seen as a sequence of several startup, main, and end phases. If the arrival rate of new problems is high, the master solver quickly becomes a bottleneck of the system, because all engaged solvers, which are running idle within an end phase of a computation

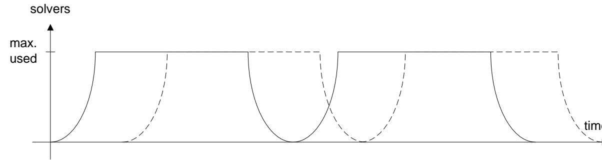

Fig. 9. Overlapped startup, main, and end phases

have to wait for new work which is not delegated to them until the following startup phase. This is caused by the fact that the master solver represents the only interface to the skeleton. The more solvers are used in the skeleton, the faster a problem will be solved in the main phase of the computation. If the arrival rate of new problems is low, the DCSkeleton can be adapted to this rate by adjusting the number of engaged solvers. Thus, the overall time for processing all problems in the stream is the sum of the subtracted times for solving the single problems.

The stream processing can be optimized by overlapping phases of high workload with phases of poor workload. In case of the DCSkeleton we find high work load within the main phases and poor workload within the startup und end phases of the computation. If the solution of a divide and conquer problem is in its startup or end phase, only few or even no subproblems exist in the system which can be distributed among the solvers. As shown in figure 9, the phases can be overlapped if more than one divide and conquer problem is processed by the skeleton at a time. If the computation of a solution is in its startup or end phase, the processing of another problem may be in its main phase. This leads to a more balanced processor load due to the fact that the amount of work is increased within the skeleton and thus idle times are reduced. The number of problems which are prepared for load distribution increases linearly with the number of divide and conquer problems solved in parallel. Thus, a less fine-granular decomposition of each divide and conquer problem is necessary to guarantee a sufficient amount of work for all solvers the more divide and conquer problems are solved in parallel. By generating fewer but bigger subproblems the efficiency of the skeleton can be increased not only by reducing the number of `divide` and `combine` operator calls, but also by raising the sequential proportion of the computation by applying `solve` on larger problems.

Figure 10 illustrates the design of an optimized divide and conquer skeleton for stream processing (StreamDC), which is based on the DCSkeleton. In contrast to the DCSkeleton it consists of n master solvers receiving new divide and conquer problems from the predecessor. Each solver maintains a multi-part work and solution pool to distinguish between subproblems which emanate from different initial problems. If the workpool stores subproblems which emanate from problems received from another solver, these problems are processed first. In this case, a master solver delegating a subproblem is supplied with its corresponded partial solution more quickly. This speeds up the overall computation time of

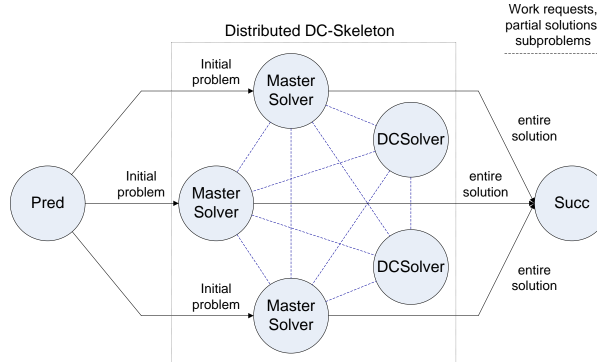

Fig. 10. A fully distributed divide and conquer skeleton for stream processing

an internally distributed problem, and an adequate supply of new work to the skeleton is guaranteed by receiving new initial problems from the predecessor more quickly as well.

If the arrival rate of new problems is low, this skeleton behaves like a DCSkeleton. An idle master solver sends work requests and receives subproblems from its neighbors. If the arrival rate of problems is high, each of the master solvers receives new problems from its predecessor, which increases the amount of work within the skeleton. For larger number of processors, this leads to a faster propagation of work to idle solvers in the beginning of the computation, and increases the utilization of solvers during the whole computation as shown above. Thus, by applying the StreamDC skeleton with application specific parameters, it can be configured to be a hybrid of a pure stream processing farm and the DCSkeleton. It can be adapted to the arrival rate and the size of the divide and conquer problems which are to be solved so that the distributed memory utilization is improved. For this reason, the StreamDC is able to solve problems, which cannot be solved by a sequential DCSkeleton used in farms due to the lack of memory. In comparison to the DCSkeleton the new StreamDC skeleton benefits from overlapping the startup and end phases of solving single problems by solving several problems in parallel. Moreover, fewer problems must be prepared for load distribution which reduces `divide` and `combine` operator calls and increases the sequential part of the computation.

3 Applications and Experimental Results

The parallel test environment for our experiments is an IBM workstation cluster [33] of sixteen uniform PCs connected by a Myrinet [25]. Each PC has an Intel Xeon EM64T processor (3.6 GHz), 1 MB L2 cache, and 4 GB memory, running Redhat Enterprise Linux 4, gcc version 3.4.6, and the MPICH-GM implementation of MPI.

#Solvers	Strassen	Karatsuba	Mergesort
1	295,94	43,52	43,19
2	146,52	22,10	28,25
3	98,13	14,94	24,78
4	73,93	11,41	22,79
5	59,86	9,32	21,91
6	50,58	7,89	21,04
7	42,85	6,92	20,60
8	38,53	6,16	20,03
9	34,92	5,58	19,95
10	32,10	5,14	19,79
11	29,82	4,76	19,58
12	27,72	4,47	19,42
13	26,28	4,18	19,27
14	24,79	3,96	19,15

Table 1. Runtimes for Strassen, Karatsuba, and Mergesort (in seconds).

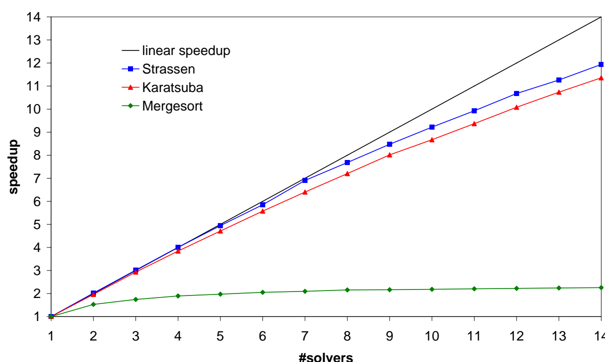

Fig. 11. Speedup for the Mergesort, Karatsuba, and Strassen algorithm.

In order to evaluate the performance and scalability of the distributed divide and conquer skeleton, we have considered three problems with different complexity classes. At first, we have implemented the standard mergesort algorithm [19], which is in $O(N \log N)$, to sort randomly generated integer arrays of size $N = 2^{26} = 67108864$. Moreover, we have implemented the Karatsuba multiplication algorithm for big integers ($O(N^{\log_2 3})$, where $\log_2 3 \approx 1,58$). The Karatsuba algorithm [20] reduces a multiplication of two N-digit integers to three multiplications of $\frac{N}{2}$-digit integers. In our experiments we have generated two numbers with $2^{20} = 1048576$ digits for each test run. Finally, we have implemented Strassen's algorithm for matrix multiplication ($O(N^{\log_2 7})$, where $\log_2 7 \approx 2,808$) in order to multiply two randomly generated 4096×4096 integer matrices. The Strassen algorithm [31] reduces a multiplication of two $N \times N$ matrices to seven multiplications of $\frac{N}{2} \times \frac{N}{2}$ matrices. The algorithms differ not only in their complexity classes, but also in the dynamically generated divide and conquer tree, which is of degree $d = 2$ for mergesort, $d = 3$ for Karatsuba, and $d = 7$ for Strassen. Note that the skeleton behaves non-deterministically in the way the load is distributed. Generating only a few big subproblems can lead

a) Mergesort	solver			
	1	2	3	4
processed problems	8193	8192	8191	8191
solved problems	4096	4096	4096	4096
distributed problems	2	1	0	0
received problems	0	1	1	1
# work requests	5	9	17	26
time for solve	4,24	4,24	4,28	4,29
time for combine	4,65	3,57	2,98	2,98
time for divide	1,79	1,39	1,10	1,10
Σ time	10,68	9,19	8,36	8,37

b) Karatsuba	solver							
	1	2	3	4	5	6	7	8
processed problems	3580	3707	3737	3664	3704	3727	3671	3734
solved problems	2466	2469	2490	2492	2446	2443	2391	2486
distributed problems	18	16	10	10	16	15	4	7
received problems	7	9	12	12	12	16	17	11
# work requests	43	57	60	56	95	95	98	58
time for solve	5,19	5,20	5,25	5,24	5,14	5,14	5,24	5,20
time for combine	0,24	0,19	0,18	0,16	0,16	0,15	0,15	0,16
time for divide	0,22	0,18	0,19	0,17	0,18	0,16	0,15	0,17
Σ time	5,65	5,57	5,63	5,58	5,48	5,46	5,54	5,53

c) Strassen	solver							
	1	2	3	4	5	6	7	8
processed problems	2450	2482	2549	2410	2415	2364	2477	2461
solved problems	2100	2128	2184	2123	2109	2065	2027	2071
distributed problems	11	8	7	13	15	18	11	5
received problems	10	12	1	12	12	13	16	12
# work requests	79	88	26	83	73	81	84	74
time for solve	29,56	30,67	30,93	30,22	29,82	29,53	29,76	29,76
time for combine	0,88	0,66	0,65	0,62	0,62	0,60	0,59	0,53
time for divide	2,75	2,01	2,13	2,03	2,03	2,00	2,02	1,84
Σ time	33,19	33,34	33,71	32,87	32,47	32,14	32,37	32,13

Table 2. Distribution of problems, work requests, and computation time for the Mergesort, Karatsuba, and Strassen algorithm

to an unbalanced workload and to high idle times. In order to get reliable results, we have repeated each run up to 50 times and computed the average runtimes, which are shown in Table 1. Figure 11 depicts the corresponding speedups.

Table 2 shows for a typical run the number of subproblems, which are distributed and received by the solvers, as well as the number of work requests, which have been sent by each solver. Moreover, the table shows the number of subproblems, which are processed locally by each solver, as well as the number of simple problems emanating from them, which are solved sequentially. As one can see, only few work requests have been sent. In our skeleton, the *master solver* undertakes the task of dividing the initial problem and combining the entire solution. At this time, all other solvers are idle. Thus, most of the work requests were sent within the startup and the end phase. However, despite of the low number of work requests and distributed subproblems, we noticed a well-balanced work distribution in all considered example applications. This is due to the fact that each solver fetches most problems from its own workpool, such that they require no communication. This is essential for achieving good runtimes and speedups. Note that this not only applies to divide and conquer but also to other skeletons with a similar characteristic such as branch and bound and other search skeletons [27].

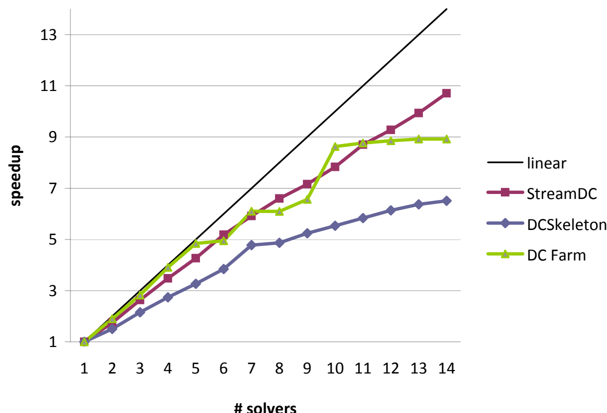

Fig. 12. Speedups for StreamDC, DCSkeleton and a sequential farm processing matrix multiplication problems

As one would expect, the skeleton reaches the lowest speedups for Mergesort. In this case, combining the entire solution takes a good deal of the overall computation time, which cannot be distributed among the solvers. This is supported by Table 2a, which shows the time for `divide`, `solve`, and `combine` consumed by the solvers in case of an equal distribution of the subproblems among the solvers. While the time for solving the subproblems is identically for all solvers, the *master solver* (solver 1) shows clearly higher computation times for `combine` and `divide`. Moreover, solving a problem locally is often faster than delegating it to a solver, because sending and receiving subproblems causes high communication costs. In our case, the best runtimes can be achieved if the threshold T for $6 \leq p \leq 14$ processors is chosen to be $T = 262144$ leading to 256 subproblems. For the Karatsuba and Strassen algorithms, the speedups are clearly better than for Mergesort, because the relation of computation time to communication time is significantly better (i.e. higher).

In order to evaluate the performance and scalability of the StreamDC skeleton, we have considered Strassen's algorithm for matrix multiplication in order to multiply two randomly generated 1024×1024 integer matrices. The stream consists of 20 matrix multiplication problems, which represents single matrix multiplications when solving a matrix chain multiplication problem $A_1 \cdot \ldots \cdot A_n$ [2, 16, 17]. Figure 12 depicts the corresponding speedups for the StreamDC, the DCSkeleton and the farm of sequential DCSkeletons. The StreamDC skeleton, which combines stream parallelism with internal task parallelism, is clearly superior to the DCSkeleton, which only provides internal task parallelism. This is by the fact that idle phases are reduced by overlapping the startup and end phases of a solution. Moreover, the number of subproblems is reduced which are prepared for load distribution. Thus, the overhead for `divide` and `combine` operator calls is decreased as well. The speedups for the farm show a kind of stairs

effect which is caused by a bad load balance due to the fact that the number of problems in the stream is only a little higher than the number of solvers. In this case the solvers are provided with a highly unbalanced amount of work. In contrast to the StreamDC skeleton, the farm is not able to do a load balancing.

4 Related work

Some related work on algorithmic skeletons for divide and conquer can be found in the literature. Recent skeleton libraries such as eSkel [9], skeTo [23], and MaLLBba [1, 12] include skeletons for divide and conquer. The MaLLBa implementation of the divide and conquer skeleton presented in [1] is based on a farm (master-slave) strategy. The distributed approach discussed in [12] offers the same user interface as the MaLLBa skeleton and can be integrated into the MaLLBa framework. The implementation differs considerably from our approach. The work and the solution pool are represented as a pointer based tree structure. Additionally, a queue with the nodes waiting to be explored is kept. Moreover, a global load balancing algorithm has been implemented, which causes high message traffic and requires a complicated communication protocol to cope with starvation problems. Unfortunately, no runtimes of the considered example applications are presented. In [8], Cole suggests to offer `divide` and `combine` as independent skeletons. But this approach has not been implemented in eSkel. The eSkel *Butterfly*-Skeleton [9] is based on *group partitioning* and supports divide and conquer algorithms in which all activity occurs in the divide phase. In contrast to our approach, the number of processors used for the *Butterfly* skeleton starts from a power of two due to the group partitioning strategy. The skeTo library [23] only provides data parallel skeletons and is based on the theory of *Constructive Algorithmics*. Restricted data parallel approaches are discussed in [4, 13]. In [13], a processor topology called N-*graph* is presented, which is used for a parallel implementation of a divide and conquer skeleton in a functional context. Hermann presents different general and special forms of divide and conquer skeletons in context of the purely functional programming language *HDC*, which is a subset of *Haskel* [14]. A distributed divide and conquer scheme is not considered there. A mixed data and task parallel approach can be found in [6].

5 Conclusions

We have considered two implementation schemes for divide and conquer skeletons. After briefly analyzing a centralized master/worker scheme as used in MaLLBa [1], we have focused on a distributed scheme. Important issues have been the demand-driven work-distribution scheme as well as an efficient approach to the combination of available partial solutions using a sorted array. Based on experimental results for Mergesort, the Karatsuba algorithm, and Strassen's algorithm, we have shown that our approach leads to good runtimes and speedups, and that it minimizes the communication overhead. Only very few problems need

to be exchanged between the different solvers. Moreover, we present a new divide and conquer skeleton optimized for stream processing. By applying the skeleton with application specific parameters, it can be configured to be a hybrid of a pure stream processing farm and the DCSkeleton, and it can range between both extremes. In comparison to the DCSkeleton the new StreamDC skeleton benefits from overlapping the startup and end phases of solving single problems by solving several problems in parallel. The advantage is, that only few problems must be prepared for load distribution which reduces divide and combine operator calls and increases the sequential part of the computation. As we have shown, the new StreamDC skeleton is clearly superior to the DCSkeleton. In comparison to a farm of sequentially working DCSkeletons it offers a better scalability, which is advantageous in particular when only few divide and conquer problems have to be solved. Moreover, the complete sharing of the distributed memory is a great advantage compared to a farm, in which the solvers only have access to their own local memory. Thus, the new StreamDC is able to solve problems, which cannot be solved by a sequential DCSkeleton used in farms due to the lack of memory. In future work we intend to investigate alternative stream based implementation schemes of skeletons for branch and bound and other search algorithms.

References

1. E.Alba, F.Almeida, et al.: "MALLBA: A library of Skeletons for combinatorial optimisation", in Euro-Par'02, LNCS 2400, pages 927-932, Springer, 2002.
2. G. Baumgartner: "A High-Level Approach to Synthesis of High-Performance Codes for Quantum Chemistry", In Proceedings of Supercomputing, 2002.
3. A.Benoit, M.Cole, S.Gilmore, J.Hillston: "Flexible Skeletal Programming with eSkel", in Proceedings of Euro-Par'05, LNCS 3648, pages 761-770, Springer, 2005.
4. H.Bischof: "Systematic Development of Parallel Programs Using Skeletons", PhD thesis, Shaker, 2005.
5. G.H.Botorog, H.Kuchen: "Efficient Parallel Programming with Algorithmic Skeletons", in Proceedings of Euro-Par'96, LNCS 1123, pages 718-731, Springer, 1996.
6. Y.Bai, R.Ward: "A Parallel Symmetric Block-Tridiagonal Divide-and-Conquer Algorithm", ACM Transactions on Mathematical Software, Vol. 33, No. 4, Article 25, 2007.
7. M.Cole: "Algorithmic Skeletons: Structured Management of Parallel Computation", Pitman/MIT Press, 1989.
8. M.Cole: "On Dividing and Conquering Independently", in Proceedings of Euro-Par'97, LNCS 1300, pages 634-637, Springer, 1997.
9. M.Cole: "Bringing skeletons out of the closet: A pragmatic manifesto for skeletal parallel programming. In Parallel Computing 30(3), pages 389-406, 2004.
10. J.Darlington, Y.Guo, H.To, J.Yang: "Parallel skeletons for Structured Composition", in Proceedings of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pages 19-28, ACM Press, 1995.
11. M.Eriksson, C.Kessler, M.Chalabine: "Load Balancing of Irregular Parallel Divide-and-Conquer Algorithms in Group-SPMD Programming Environments", in proceedings of ARCS'06 Workshop on Parallel Systems and Algorithms (PASA'06), Frankfurt, Germany, 2006.

12. J.R.González, C.León, C.Rodríguez: "A Distributed Parallel Divide and Conquer Skeleton", PARA'04, 2004.
13. S.Gorlatch: "N-graphs: scalable topology and design of balanced divide-and-conquer algorithms", Parallel Computing, 23(6), pages 687-698, 1997.
14. C.A.Herrmann: "The Skeleton-Based Parallelization of Divide-and-Conquer Recursions", PhD thesis, Logos-Verlag, 2000.
15. D.Henrich: "Initialization of parallel branch-and-bound algorithms", In proceedings of the 2nd International Workshop on Parallel Processing for Artificial Intelligence (PPAI-93), Elsevier, 1994.
16. T.C. Hu, M.T. Shing: "Computation of matrix chain products I", SIAM Journal on Computing, 11(2):362-373, 1982.
17. T.C. Hu, M.T. Shing: "Computation of matrix chain products II", SIAM Journal on Computing, 13(2):228-251, 1984.
18. H.Kuchen, M.Cole: The Integration of Task and Data Parallel Skeletons, Parallel Processing Letters 12(2), pages 141-155, 2002.
19. D.E.Knuth: "The Art of Computer Programming, Vol. 3: Sorting and Searching", Addison-Wesley, 1973.
20. A.Karatsuba, Y.Ofman: "Multiplikation of multidigit numbers on automata" Doklady Akademii Nauk SSSR, 145(2):293-294, 1962.
21. H.Kuchen: "A Skeleton Library", in Euro-Par'02, LNCS 2400, pages 620-629, Springer, 2002.
22. H.Kuchen: "Optimizing Sequences of Skeleton Calls", in Domain-Spezific Program Generation, LNCS 3016, pages 254-273, Springer, 2004.
23. K.Matsuzaki, K.Emoto, H.Iwasaki, Z.Hu: "A Library of Constructive Skeletons for Sequential Style of Parallel Programming", in proceedings of 1st international Conference on Scalable Information Systems (INFOSCALE). 2006.
24. Message Passing Interface Forum, MPI: A Message-Passing Interface Standard, http://www.mpi-forum.org/docs/mpi-11-html/mpi-report.html
25. The Myricom homepage. http://myri.com/.
26. S.Pelagatti: "Task and Data Parallelism in P3L", in Patterns and Skeletons for Parallel and Distributed Computing, eds. F.A.Rabhi and S.Gorlatch, pages 155-186, Springer, 2003.
27. M.Poldner, H.Kuchen: "Algorithmic Skeletons for Branch & Bound", in proceedings of 1st International Conference on Software and Data Technology (ICSOFT), Vol. 1, pages 291-300, Setubal, Protugal, 2006.
28. M.Poldner, H.Kuchen: "On Implementing the Farm Skeleton", Parallel Processing Letters, Vol. 18, No. 1, pages 117-131, 2008.
29. M.Poldner, H.Kuchen: "Skeletons for Divide and Conquer Algorithms", Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Networks (PDCN), ACTA Press, 2008.
30. M.Poldner, H.Kuchen: "Optimizing Skeletal Stream Processing for Divide and Conquer", Proceedings of the 3rd International Conference on Software and Data Technologies (ICSOFT), pages 181-189, INSTICC Press, 2008.
31. V.Strassen: "Gaussian Elimination is not optimal", Numerische Mathematik, 13:354-356, 1969.
32. R.van Nieuwpoort, T.Kielmann, H.Bal: "Efficient Load Balancing for Wide-Area Divide-and-Conquer Applications", Proceedings of the eighth ACM SIGPLAN symposium on Principles and practices of parallel programming PPoPP '01, Vol.36, Issue 7, 2001.
33. Ziv-Cluster. http://zivcluster.uni-muenster.de/.

Extended Exceptions for Contingencies

Thorsten van Ellen

Carl von Ossietzky University of Oldenburg
Software Engineering Group
26111 Oldenburg, Germany
thorsten.van.ellen@informatik.uni-oldenburg.de

Abstract. The set of situations that are specification compliant usually is called normal and won't be subdivided, but in this work a new fundamental refinement will be defined. The set of normal situations shall be subdivided into two subsets on the basis of special criteria. Only one subset shall be called normal. The situations of the other subset shall be called contingencies. Results of function calls that are contingencies, e.g., DiskFull, are work refusals from the point of view of the caller. They restrain the caller from reaching his postconditions which leads to a specification violation. Therefore, contingencies must be handled to prevent specification violations. But a successful handling is only reasonable if subsequent resumption is possible. In this work, significant properties of contingencies will be presented that hamper their handling. To handle contingencies, conventional exception mechanisms can be used, but are accompanied by considerable deficits for that purpose. For example, a correct resumption after a successful handling is extremely difficult if the language does not support it directly. Therefore, extended exception mechanisms are drafted here that should solve these deficits. A systematic inspection and handling of contingencies can diagnose and avoid subsets of specification violations effectively before runtime.

1 Motivation

Many programming languages offer exception mechanisms, but only allow the termination of the execution or a part of it in the case of an error (termination model). Only few languages offer language mechanisms for resumption (resumption model) [1], because resumption is controversial.

For example, the added value of resumption is challenged by empirical research [2]. The empirical research could not find examples that argue for resumption. Up to now it was difficult to find good examples for resumption systematically. Exactly that should happen here. By defining the new term contingency, many examples can be found for which resumption might be reasonable.

With the term contingency, the space of errors is divided to make the situations that are treatable accessible for complete handling. Conventional exception handling is not able to make this distinction and therefore, sufficient handling and resumption is not possible. It will be clarified that contingencies must be handled to avoid errors and resumption is needed. Perhaps a new orientation

and discussion of errors, exceptions, handling and resumption originates with this new perception.

2 Overview

Section 3 explains some terms and reasons why contingencies must especially be regarded. Section 4 presents the objectives of this work. Section 5 clarifies properties of contingencies and suggests why handling of contingencies within conventional programming languages with exception handling is reasonable. Section 6 pinpoints some significant decifits of exception handling for contingencies. It will also be illustrated that some alternative handling approaches can't be applied across the board. Accordingly, section 7 proposes extended exception mechanisms. Section 8 compares less distributed mechanisms to the mechanisms proposed here regarding their suitability. Finally, section 9 comes to conclusions.

3 Terms

This section describes some essential terms.

Partially, errors are defined as states, e.g., by [3]. But not only unintentional states exist, that should be called errors, e.g., if a traffic light shows red and green at the same time, but also unintentional transitions, e.g., if a traffic light changes directly from green to red, where no unintentional state is involved (see figure 1). .

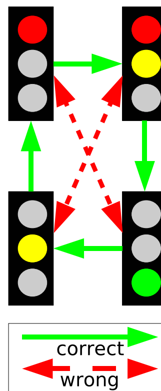

Fig. 1. Valid and invalid traffic light transitions

Therefore, the term situation should be used instead of state here and be defined as following:

Definition 1. ***Situation**: A situation is a sequence of n states where $n > 0$.*

A situation can be a sequence consisting of a single state and for the sake of simplicity situations can be illustrated as states.

Definition 2. ***Specification****: A specification is a complete decription of all situations that are allowed for a system, without contradictions of itself.*

Example 1. A specification of a function can be composed of a pre- and postcondition. The precondition for a traffic light describes the valid states, the postcondition the valid transitions starting from the valid states.

A specification that allows a situation at one place and forbids the same situation at the same time at another place, e.g., by preconditions, contradicts itself.

A specification of a non-trivial system, e.g., with the specification language B, is usually modular, similar to program code. A specification usually contains modules that build upon each other or are dependent on each other respectively, and share work or tasks that have to be completed.

Definition 3. ***Error****: An error is a situation the conditions of which contradict the specification.*

An error or specification violation occurs, for example, if the situation does not comply with the preconditions or postconditions, as Meyer states [4].

Example 2. A direct traffic light transition from green to red is an error.

The new term contingency should be defined as follows:

Definition 4. ***Contingency****: A contingency is a situation that is described within the specification of a module and represents a result where the essential work of the module other modules depend on was not performed.*

Example 3. If a fax should be sent via a modem controlled by software, the telephone line can be busy (`BusyLine`). Usually, a busy tone is not avoidable, even not by changing the program code of the software.

Contingencies are results of modules that indicate that the module could or should not fulfill its usual work. Such situations are no errors or specification violations, because they are unavoidable or intentional behaviors and therefore can be found within the specification. Contingencies are exactly and sufficiently described within the specification and therefore specification compliant. If no additional specific measures are taken, the expectations of dependent modules won't be fulfilled and they can't reach their postconditions so that as a result a specification violation will appear. From the perspective of the dependent module, contingencies are blockades or work refusals whose potential appearance is known in advance.

Contingencies differ from normal situations in that normal situations without additional specific measures do not necessarily run into specification violations.

4 Objectives

Contingencies must be handled, if following specification violations should be avoided. The objectives of this paper are

– to distinguish contingencies from normal situations and errors and
– to determine contingencies before runtime,
– to allow a simple communication, specific handling and simple resumption after a successful handling.
– Additionally, the context should be enabled to overwrite handlers.
– This should also be true for side effects that do not reside within the digital memory or do not work with ACID-transactions and without compensation mechanisms.
– Furthermore, information hiding should be kept and
– complications of interfaces, cumbersome cleanups and partial repititions should be avoided.

Only sequential operations should be regarded.

5 Properties of Contingencies

The following section presents significant properties of contingencies and describes why handling of contingencies within conventional programming languages with exception handling is reasonable.

5.1 Contingencies are very Numerous

Contingencies can be very numerous alone within one single function.

Example 4. Alone at the call of a function to load or save a file several contingencies can occur, e.g., Drive/Dir/File-NotFound/Locked/NameInvalid, DiskNotInDrive (e.g., USB-Stick), DiskNotFormatted, DiskFull, EndOfFile, NoAvailableFileHandles, NetworkDisconnected etc.

Contingencies are littered over very many functions within the whole system and all levels.

Example 5. Examples are UnknownPhoneNumber, ParticipantTemporarilyUnavailable, OutOfPaper, OutOfInk, PaperJam, ConcurrentAccess, ClassNotFound, OutOfMemory, AccountInvalid, PasswordInvalid etc.

A severe reason seems to subject the previous examples, where the system can not continue work quasi physically, but contingencies can also correspond to arbitrary semantic requirements where the function or system should not continue its work although it would physically be possible.

Example 6. To avoid financial risks and damages, banking houses avoid that customers can overdraw their account arbitrarily. They define a limit that can be different for each customer. If the limit is exceeded, the automatic teller machines refuse to work and also withdrawals as part of superordinated, automatic processes are refused. Further examples are PasswordExpired, AccountLocked, Drive/Dir/File-ReadOnly, QuotaOverflow etc.

The following example from the database domain shall illustrate how numerous contingencies actually are within everyday life. Presumably contingencies aren't less numerous within operating systems or other complex environments like ERP systems, only less documented and apparent. This shows how important it is to handle these situations explicitly.

Example 7. Oracle maintains a documentation ([5]) of all problem messages of the Oracle database. It encompasses over 2000 pages, each with several messages. Hence, it documents several thousands of entries and mostly contains detailed and specific (not abstracted) and therefore helpful hints for each single known situation that can occur at runtime. These entries are not at all only errors where the database is within an unknown or undefined state, because this would be disastrous for the database and Oracle. Rather, substantial amounts of them are numerous contingencies that are recognized, intercepted and communicated at runtime successfully and documented (quasi specified) before runtime. At runtime, they leave the database within a sufficiently defined state that is additionally still ready for full operation.

But usually the tasks of the caller or user are aborted with a corresponding message. If the superordinated tasks have not been prepared for the contingency, their postconditions can not be fulfilled anymore and a specification violation occurs.

This example illustrates that Oracle databases actually contain very many contingencies. However, not all of them can occur, if the calling program is implemented correctly, e.g., the message ORA-01747 ("‘invalid column"’) can't occur, if all DDL and DML statements are consistent.

5.2 Contingencies are Unavoidable

Refusals to work usually will be avoided intuitively at system development, only the unavoidable or semantically required ones remain as contingencies.

5.3 Contingencies are Better Treatable than Errors

If a function can or should not fulfill its work under conditions that are described within the specification exactly, then this is a contingency (no error) and is known in advance. If the function communicates the contingency to the caller specifically, e.g., as special value or exception, it is possible to handle it within sufficiently defined system environments and circumstances. Handling contingencies is therefore easier than handling errors, because errors violate the specification and are not described sufficiently within the specification. Hence, the system environment and circumstances of errors are only necessarily defined.

5.4 Use Exceptions for Contingencies

If every caller would handle every contingency immediately, the same problem would occur that is already known as "error code handling", i.e., every line of normal code would be mixed with many code lines for handling many contingencies. The handling code would be highly redundant for handling the same contingencies at many places. Error code handling has been found impracticable and has been replaced by exception handling. To avoid error code handling for contingencies, too, contingencies can also be commmunicated and handled with exceptions.

5.5 Contingencies Disclose Implementation Details and Must Not Be Abstracted

When passed to the callers in the caller hierarchy, contingencies disclose implementation details not immediately, but mostly after a few call levels. (This is also true for exceptions that represent errors).

Since it is mandatory to handle contingencies to avoid following specification violations, it must be assured that they remain unambiguous and won't be abstracted. If different contingencies are projected onto one abstract contingency, e.g., OutOfMemory and DiskFull onto OutOfResource, the different conditions of the different contingencies can't be distinguished and specific handling is no longer possible.

Example 8. A specific handling for OutOfMemory like swapping is neither applicable for DiskFull nor for the abstraction of both OutOfResource.

An abstraction of contingencies to keep the information hiding principle and to hide implementation secrets is therefore not recommended, because it would project the contingency of the current implementation and the potential contingencies of future implementations onto one abstracted situation for which a current handling won't be appropriate automatically.

5.6 Contingencies Accumulate

In the figure 2 routines are symbolized by small letters. Part A) illustrates: if in routine m which is called by routine g and i an exception occurs (error or contingency), that won't be handled within g and i, the exception will be passed to the next caller. Part B) illustrates: if each routine generates three special different exceptions, that won't be handled, then within the higher call levels an increasing number of exceptions accumulate. At the end all exceptions that haven't been handled can be found within the `main` routine. This accumulation of unhandled exceptions increases with the call hierarchy depth and broadness of the program or system. The accumulation exists independently of the used language mechanism. It's the same with error codes. It seems as if this was the driving force for replacing error code handling by exceptions.

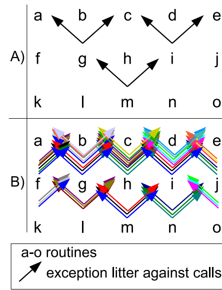

Fig. 2. Accumulation

6 Deficits of Conventional Exception Mechanisms

This section outlines deficits of conventional exception mechanisms to easily handle contingencies successfully and continue execution afterwards. It also illustrates that alternative approaches to resume execution after successful handling are not applicable across the board.

Contingencies can be handled more easily than errors and must be handled, if following specification violations have to be avoided. To handle them, they must be marked and be ascertainable at development time. Conventional languages have no mechanisms to determine all exceptions that can occur syntactically within a code fragment at development time, although that should be no complex problem.

Repairs of the lower call levels from the higher call levels require knowledge of and access to the implementation details of the lower levels within the higher levels. In the following example it is not possible to manipulate the variable `currentPath` with conventional mechanisms:

Example 9.

```
void main(String[] args) { // GUI-access here
  try {
    doTasks(args);
  } catch (HardDiskFull) {
    String alternative =
      AskUserForAlternative.execute().result();
    // set/repair alternative, but where and how?
  }
}

// many call levels lower:
void saveEditedData(Data edited)
throws HardDiskFull { // No GUI here
  if (getFree(currentPath) < edited.size()) {
```

```
    throw new HardDiskFull(getFree(currentPath),
                           edited.size());
  } // ... writing data on disk
}
```

After a successful handling the program should be resumed to fulfill the postconditions, but only few languages exist that offer resumption mechanisms [1]. Languages that only offer the termination model raise the following hurdles that complicate and restrict resumption in many cases:

– Repairs of the lower call levels after an exception require knowledge of the implementation details, which violates the information hiding principle. To allow arbitrary repairs of the lower call levels via interfaces, e.g., partial states [6], extreme interface complications would be required.
– No mechanisms are known to handle side effects that can't be controlled via ACID transactions or compensations, e.g., irreversible physical side effects.
– Partial repetitions generate performance and time losses.

Language mechanisms for resumption are the only known and generally applicable option to bypass the latter hurdles. The only language of those that offer resumption is Common Lisp and its language family that allow multiple resumption alternatives even on different call levels and, furthermore, to parametrize them.

But even these don't offer a mechanism to override handlings quasi polymorphic, to exploit additional context knowledge and access. Therefore, the following Java example can not override the handling of the exception `PaperJam` within the method `print` by the handling in the method `printAdvanced`.

Example 10.

```
public void print(Object document, int fromPage) {
  try {
    // print ...
  } catch (PaperJamDetected jamDetected) {
    // Default handling: cancel and logging
    logger.error(Paper jam: aborting print.);
  }
}

public void printAdvnc(Object doc, int fromPage) {
  try {
    print(document, fromPage);
  } catch (PaperJamDetected jamDetected) {
    // Advanced NOT possible: automatically repair!
    advancedPaperEmitter.removeJam();
    printAdvanced(document, jamDetected.atPage());
  }
}
```

7 Solving with Extended Exception Mechanisms

Since using exceptions for contingencies is reasonable and conventional exception handling shows deficits for contingencies and resumption, extended exception mechanisms are sketched for the language Java that offer handling and resumption of contingencies. The concepts are transferable for other languages. Significant ideas are coined by the language Common Lisp [7] and partially modified and extended. At first an example illustrating the extended mechanisms follows. Afterwards, the mechanisms are explained in more detail.

7.1 Summarizing Example

The following example was coined by [8] and shows all extended mechanisms in conjunction.

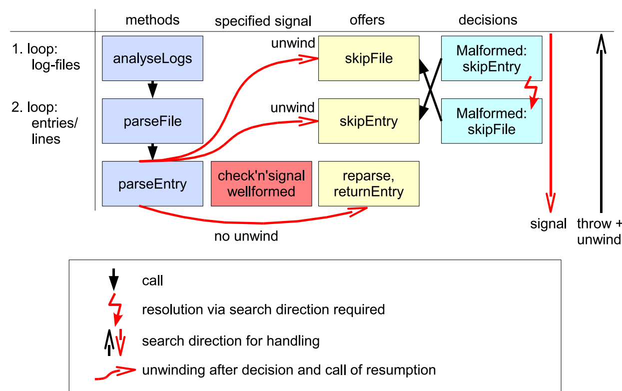

Fig. 3. Example with log-files

It will be presented within the figure 3 and as source code within example 11.

Within the example multiple log-files should be read. Within the log-files are multiple lines that should be checked for whether they are well-formed. For this purpose two nested loops are used. The first loop iterates over the files. The second loop iterates over the entries of one file. Both loops are implemented within two separated methods `analyseLogs` and `parseFile`, of which the first calls the second. For each line of the files a third method `parseEntry` will be called, that checks whether the line is well-formed. If not, it signals the contingency `MalformedLine` by the new keyword `signal` (see section 7.2).

Both loops offer an option to resume which abstraction corresponds to the according implementation level, i.e., the loop within `analyseLogs` offers `skipFile` and the loop within `parseFile` offers `skipEntry`, each without parameter. The

offers are introduced by the new keyword `offer` (see section 7.5). They are additional interfaces with name and optional parameters that are a side entrance to the corresponding method.

Furthermore, both loops decide what should happen in the case that a line is not well-formed (`MalformedLine`). For the choice of the contingency the new keyword `decide` is used (see section 7.4). Within the handling of the chosen contingency the choice of the repair and resumption offer is done with the new keyword `resume` (see section 7.6).

Within the method `parseFile` the repair offer `returnEntry` is called and the element `defaultEntry` is passed as parameter, which only exists there. This way both involved levels can cooperate (see section 7.8).

Two possibilities to decide or handle the contingency `MalformedLine` on two different call levels exist within the example. The higher method overrides the decision of the lower method, therefore, the decision with resumption of `skipEntry` is chosen (see section 7.7).

Example 11.

```
void analyzeLogs(Files openFiles) {
  for (File file: openFiles) {
    try {
      use(parseFile(file));
    } offer skipFile() {
      continue; // nothing else to do
    } decide (MalformedLine x) {
      if (x.firstStackFrame().startsWith("mylib"))
        resume skipEntry();
    }
  }
}


Entries parseFile(File openFile) {
  Entry defaultEntry = new Entry();
  Entries result;
  while (!openFile.EOF()) {
    Entry entry = null;
    try {
      String logTxt = openFile.line();
      entry = parseEntry(logTxt);
    } offer skipEntry() {
      // entry = null;
    } decide (MalformedLine) {
      // entry = null;
      resume returnEntry(defaultEntry);
    }
```

```
    if (entry != null)
      result.add(entry);
  }
  return result;
}

Entry parseEntry(String logTxt) {
  if (entryIsWellFormed(logTxt)) {
    return new Entry(logTxt);
  } else {
    try {
      signal new MalformedLine(logTxt);
    } offer reparse(String in) {
      return parseEntry(in); // retry other value
    } offer returnEntry(Entry out) {
      return out; // default value
    }
  }
}
```

Additionally, the method `parseEntry` contains two parametrized repair offers that present further applications of offers, e.g., repetition by recursive calls with change input data (`reparse`) and return of default results (`returnEntry`).

7.2 Mark and Recognize Contingencies before Runtime

Contingencies with sufficiently defined conditions shall be distinguished from errors with only necessarily defined conditions. For this purpose, contingencies shall be marked with the new keyword `signal` and distinguished from errors with the known keyword `throw`. As a result, all contingencies that occur syntactically within a code fragment, including all called levels, can be determined before runtime.

7.3 Choose Contingencies Interactively

In non-trivial cases not all contingencies can or should be handled everywhere, therefore, it should be possible to choose arbitrary contingencies at arbitrary places. Due to the expected huge amount of contingencies [5] only an interactive choice seems reasonable, i.e., the choice should not be supported by syntactical mechanisms, but by cooperation of all corresponding parser or compiler respectively, and development environments. The development environment should offer a sorting and filtering by diverse criteria, e.g., by type, class hierarchy, location, call chain, frequency, already registered occurrence and the existence of handlings of the contingency etc.

7.4 Distinguish Handling with and without Possibility of Resumption

When the contingency is chosen, the handling can be developed. The stack must not be unwound, if a resumption should be possible. To distinguish whether the stack is already unwound or not, instead of using the keyword `catch` where the stack is already unwound, the new keyword `decide` should be used for signaled contingencies where the stack is not unwound. If multiple call levels of the call chain exist that can handle the contingency, the handling that is the topmost in the call chain is executed. In this way, the handling can be overridden quasi polymorphically along the call chain.

7.5 Alternatives for Resumption

Multiple different possibilities for resumption of contingencies can be defined at every arbitrary level with separate name and parameters. These resumption possibilities are introduced with the new keyword `offer` which is followed by a name and formal parameter declarations with usual notation. These offers are side entrances into the interrupted methods that are still on the stack. They are like procedures (without result value), because they should resume and not return. Offers are only available for handlings of contingencies (decisions). They are additional interfaces and can keep the information hiding principle as conventional interfaces.

7.6 Resumption

To call offers from decisions the new keyword `resume` is used, followed by the name and the required actual parameters with usual notation. If multiple offers with the same name and parameters on different call levels exist, the topmost offer is chosen. For this reason the resumption is not always fixed to the level where the contingency is signaled originally. In this way, offers can be overriden quasi polymorphically along the call chain. When the offer and its level is chosen the stack will be unwound till there.

7.7 Reversal of Search Direction

By reversing the search direction for handlings top-down in contrast to the conventional search direction bottom-up, it becomes possible to override handlings, as it was intended by the example 10.

7.8 Cooperation of Levels

Furthermore, the approach presented here enables the cooperation of the involved levels, e.g., to use the context knowledge of the decision level (`advancedPaperEmitter` of example 10 or the user interface of the example 9) and also to repair the implementation (`currentPath` of example 9) on the offer level without violating its information hiding.

8 Comparison of Alternative Mechanisms

In this section the presented approach will be compared to alternative languages and mechanisms. The rows of the table 1 list the features of the presented approach. The columns show which features are supported by the alternatives.

Features	Common Lisp	Java with Resumption (Callbacks) [9]	Closures	Continuations	AOP
Distinction of contingencies	-	-	-	-	-
Ascertaining contingencies	-	-	-	-	-
Interactive choice	-	-	-	-	-
Reverse search direction	-	-	-	-	-
Cooperation of involved levels	x	x	x	x	-
Resumption	x	x	x	x	-
Resumption at arbitrary level	x	-	-	-	-
Multiple offers	x	x	x	-	-
Access to decision level after recognition	x	-	x	x	-
Repair of offer level	x	x	x	-	-
Parametrizing of offer/repair	x	x	x	-	-

Table 1. Comparison of alternative languages and mechanisms (x = possible, - = not possible)

Note for AOP: join points for throw statements are unknown, therefore, it can't be avoided that the stack is unwound, hence, resumption isn't possible.

9 Conclusions

Contingencies are known but undesired results and are specified as well as the desired results.

In this light, contingencies are the situations that can really be handled and where resumption is very reasonable. Hence, contingencies are the true exceptions. Based on their only necessary defined conditions, errors are more difficult to handle reasonably or even specifically.

The systematic consideration and handling of contingencies can diagnose and avoid subsets of specification violations effectively before runtime. Furthermore, specification violations are not detected by symptoms, but on the basis of their sources. Therefore, partially extensive analyses to draw conclusions from the symptoms to the sources can be saved.

References

1. Garcia, A.F., Rubira, C.M.F., Romanovsky, A., Xu, J.: A comparative study of exception handling mechanisms for building dependable object-oriented software. The Journal of Systems and Software **59**(2) (2001) 197–222
2. Stroustrup, B.: The Design and Evolution of C++. Addison-Wesley Longman (April 1994) ISBN 0201543303.
3. Avizienis, A., Laprie, J.C., Randell, B., Landwehr, C.E.: Basic concepts and taxonomy of dependable and secure computing. IEEE Trans. Dependable Sec. Comput. **1**(1) (2004) 11–33
4. Meyer, B.: Object-Oriented Software Construction. Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1988)
5. Oracle: Oracle9 i database error messages, release 2 (9.2) part no. a96525-01. (2002) http://download.oracle.com/docs/cd/B10501_01/server.920/a96525.pdf.
6. Miller, R., Tripathi, A.: Issues with exception handling in object-oriented systems. Lecture Notes in Computer Science **1241** (1997) 85–103
7. Pitman, K.M.: Exceptional situations in lisp. In: Proceedings for the First European Conference on the Practical Application of Lisp (EUROPAL'90), Cambridge, UK (1990)
8. Seibel, P.: Practical Common Lisp. Apress (September 2004) PDF at http://www.apress.com/resource/freeebook/9781590592397 and HTML at http://gigamonkeys.com/book/.
9. Gruler, A., Heinlein, C.: Exception handling with resumption: Design and implementation in java. In: PLC. (2005) 165–171

Männliche und weibliche Doppelstudenten: ein Härtetest für Programmiersprachen

Christian Heinlein

Studiengang Informatik
Fakultät Elektronik und Informatik
Hochschule Aalen – Technik und Wirtschaft
vorname.nachname@htw-aalen.de

Abstract. Vererbung und Subtyp-Polymorphie sind zwei wesentliche Konzepte, die objektorientierte Programmiersprachen von klassischen prozeduralen Sprachen unterscheiden. Mit Hilfe einfacher Vererbung, die von allen objektorientierten Sprachen angeboten wird, lassen sich z. B. Männer, Frauen und Studenten jeweils als Untertypen eines allgemeineren, eventuell abstrakten Obertyps Person definieren. Um männliche und weibliche Studenten möglichst einfach und direkt definieren zu können, ohne bereits vorhandene Definitionen wiederholen zu müssen, benötigt man mehrfache Vererbung, die jedoch nur von wenigen Sprachen (z. B. C++, Eiffel und CLOS) unterstützt wird und in der Praxis zu zahlreichen Komplikationen führt (z. B. Namenskonflikte, Unterscheidung von virtuellen und nicht-virtuellen Basisklassen etc.). Um schließlich Klassen definieren zu können, die die Eigenschaften anderer Klassen mehrfach besitzen – wie z. B. Doppelstudenten, die quasi zweimal Student in unterschiedlichen Studiengängen sind –, benötigt man wiederholte Vererbung, die von keiner gängigen Sprache direkt unterstützt wird, aber z. B. durch künstliche Hilfsklassen in C++ oder durch Umbenennen von Features in Eiffel erreicht werden kann. Spätestens beim Versuch, mehrfache und wiederholte Vererbung miteinander zu kombinieren – um beispielsweise männliche und weibliche Doppelstudenten zu modellieren –, stößt man jedoch an die Grenzen gängiger Programmiersprachen.
Mit Hilfe sogenannter offener Typen und bidirektionaler Relationen, einem Kernkonzept meiner Forschungsarbeit über „verbesserte prozedurale Programmiersprachen“, lassen sich nicht nur mehrfache und wiederholte Vererbung einfacher handhaben als mit heutigen objektorientierten Sprachen, sie erlauben darüber hinaus auch deren beliebige Kombination, d. h. die erwähnten männlichen und weiblichen Doppelstudenten lassen sich ohne Probleme definieren. Der Schlüssel zu dieser erhöhten Flexibilität liegt in der Kombination der üblicherweise inkompatiblen Konzepte von Vererbung und Aggregation zu einem einzigen einheitlichen Konzept.

1 Einfache, mehrfache und wiederholte Vererbung in C++

Abbildung 1 zeigt eine Vererbungshierarchie mit einer Wurzelklasse `Person`, direkten Unterklassen `Mann`, `Frau` und `Stud` (Student/in) sowie indirekten Unterklassen `MnlStud` (männlicher Student), `WblStud` (weiblicher Student) und `DplStud` (Doppelstudent). Ein Doppelstudent sei hierbei ein Student, der in zwei Studiengängen

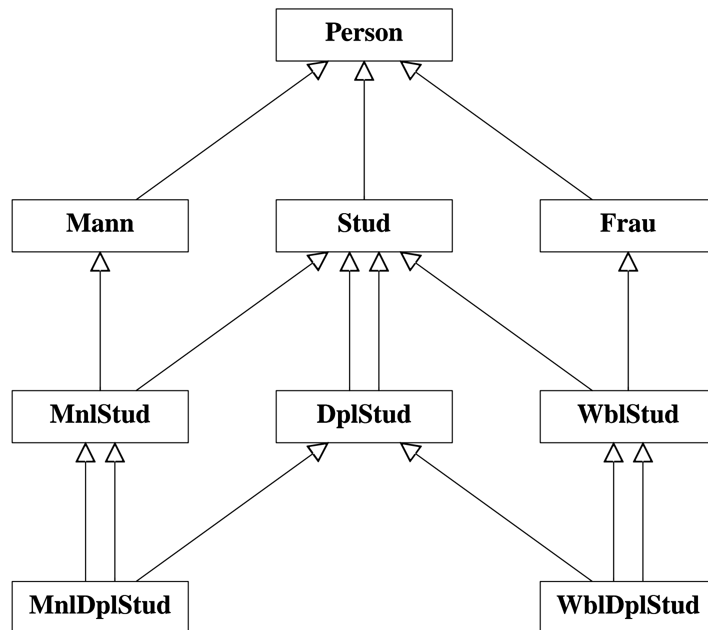

Abbildung 1: Beispiel einer Vererbungshierarchie

(z. B. Mathematik und Informatik) gleichzeitig eingeschrieben ist und dementsprechend zwei unabhängige `Stud`-Teile (mit Daten wie Studiengang, Matrikelnummer, bis jetzt erworbenen Credit points etc.) besitzen soll. Außerdem soll ein `DplStud`-Objekt zweifach polymorph als `Stud`-Objekt verwendbar sein, um beispielsweise als Mathematikstudent in eine Liste aller Mathematikstudenten und als Informatikstudent in eine Liste aller Informatikstudenten eingetragen werden zu können.

Abbildung 2 skizziert C++-Code zur Implementierung der bis jetzt beschriebenen Klassen sowie die polymorphe Verwendung eines Doppelstudenten in seinen beiden Studentenrollen. Zur Implementierung der Klassen ist folgendes anzumerken:

- Das Schlüsselwort `struct` ist äquivalent zu `class`, mit dem einzigen Unterschied, dass alle Bestandteile einer so definierten Klasse automatisch öffentlich sind.
- Die konkreten Daten und Operationen der Klassen sind für die weiteren Betrachtungen irrelevant.
- Damit Objekte der Klassen `MnlStud` und `WblStud` jeweils nur *ein* `Person`-Teilobjekt enthalten, obwohl sie indirekt jeweils zweimal von `Person` abgeleitet sind, muss `Person` eine *virtuelle* Basisklasse von `Mann`, `Frau` und `Stud` sein [4].
- Um eine Klasse wie `DplStud` zu definieren, die zweimal von derselben Basisklasse `Stud` abgeleitet werden soll, benötigt man künstliche Zwischenklassen `StudTeil1` und `StudTeil2` [4].

```
// Personen.
struct Person { ...... };

// Männer, Frauen und Studenten als spezielle Personen.
struct Mann : virtual Person { ...... };
struct Frau : virtual Person { ...... };
struct Stud : virtual Person { ...... };

// Männliche und weibliche Studenten
// als Kombination von Mann/Frau und Stud.
struct MnlStud : Mann, Stud { ...... };
struct WblStud : Frau, Stud { ...... };

// Doppelstudenten.
struct StudTeil1 : Stud {};
struct StudTeil2 : Stud {};
struct DplStud : StudTeil1, StudTeil2 { ...... };

// Ein Doppelstudent, z. B. ein Mathe-Info-Student.
DplStud* ds = new DplStud(...);

// Polymorphe Verwendung des ersten Stud-Teils.
Stud* s1 = (StudTeil1*)ds;
list<Stud*> mathe_studs; mathe_studs.push_back(s1);

// Polymorphe Verwendung des zweiten Stud-Teils.
Stud* s2 = (StudTeil2*)ds;
list<Stud*> info_studs; info_studs.push_back(s2);
```

Abbildung 2: C++-Code zur teilweisen Implementierung der Vererbungshierarchie

Die ebenfalls in Abb. 1 dargestellten Klassen `MnlDplStud` und `WblDplStud` sollen männliche bzw. weibliche Doppelstudenten repräsentieren, die polymorph sowohl als normale Doppelstudenten als auch zweifach als männliche bzw. weibliche Studenten verwendbar sein sollen, damit beispielsweise ein männlicher Mathe-Info-Student einerseits wie ein gewöhnlicher Mathe-Info-Student (und damit auch als gewöhnlicher Mathe- und als gewöhnlicher Info-Student) und andererseits sowohl als männlicher Mathe-Student (und auf diese Weise wiederum auch als normaler Mathe-Student) als auch als männlicher Info-Student (und damit auch wiederum als normaler Info-Student) verwendet werden kann. Trotzdem soll ein männlicher Doppelstudent, wie in Abb. 3 gezeigt, insgesamt natürlich nur zwei verschiedene `Stud`-Teilobjekte besitzen. (Diese Abbildung sollte am besten dreidimensional betrachtet werden: die „obere“ Ebene `Person – Stud|Stud – DplStud` beschreibt einen gewöhnlichen Doppelstudenten mit einem Person- und zwei Studententeilen, während die „untere“ Ebene `Mann – MnlStud|MnlStud – MnlDplStud` die zugehörigen männlichen Spezialisierungen enthält.)

Damit ein männlicher Doppelstudent einerseits als gewöhnlicher Doppelstudent und andererseits zweifach als männlicher Student verwendbar ist, muss die Klasse

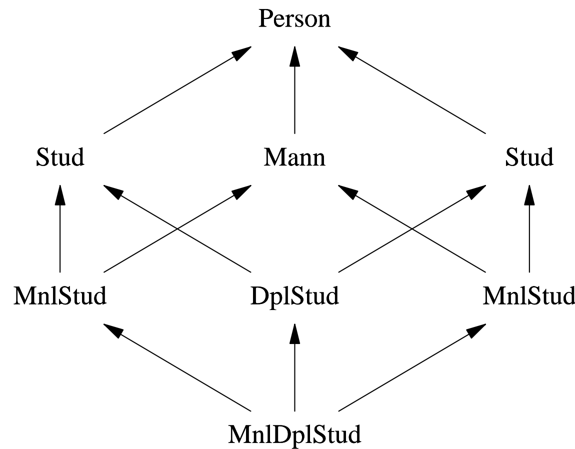

Abbildung 3: Teilobjekte eines `MnlDplStud`-Objekts

`MnlDplStud` sowohl von `DplStud` als auch zweifach (via Hilfsklassen `MnlStudTeil1` und `MnlStudTeil2`) von `MnlStud` abgeleitet werden:

```
struct MnlStudTeil1 : MnlStud {};
struct MnlStudTeil2 : MnlStud {};
struct MnlDplStud : DplStud, MnlStudTeil1, MnlStudTeil2 {};
```

Damit enthält ein Objekt dieser Klasse jedoch insgesamt vier unabhängige `Stud`-Teilobjekte! Um dies zu vermeiden, könnte man versuchen, einzelne nicht-virtuelle Unterklassenbeziehungen durch virtuelle zu ersetzen. Doch egal welche Kombination von virtuellen und nicht-virtuellen Basisklassen man wählt, man erreicht nie die in Abb. 3 dargestellte Struktur von `MnlDplStud`-Objekten, d. h. dieses Vererbungsproblem scheint in C++ unlösbar zu sein.

Andere Sprachen mit Mehrfachvererbung (konkret wurden CLOS und Eiffel untersucht) scheitern zum Teil bereits an der adäquaten Modellierung von Doppelstudenten.

2 Vererbung mit offenen Typen und bidirektionalen Relationen

Typen und Attribute. *Offene Typen* [3, 2] sind ein alternatives Datenmodell für prozedurale und objektorientierte Programmiersprachen, das im Rahmen des Projekts APPLEs (Advanced Procedural Programming Language Elements) [1] an der Universität Ulm entwickelt wurde. Ihr Grundprinzip besteht darin, dass Typ- und Attributdeklarationen syntaktisch voneinander getrennt werden. Dadurch ist es möglich, die Attribute eines Typs inkrementell zu definieren, d. h. je nach Bedarf schrittweise zu einem vorhandenen Typ hinzuzufügen. Alle so definierten Attribute eines Typs sind optional, d. h. ein konkretes Objekt des Typs muss nicht notwendigerweise Werte für alle Attribute des Typs besitzen. Wenn man auf ein nicht vorhandenes Attribut zugreift, er-

hält man als Ergebnis einen wohldefinierten Nullwert, der die Abwesenheit eines echten Werts anzeigt. Die Objekte offener Typen besitzen Referenzsemantik, ähnlich wie Objekte in Java, es gibt eine automatische Speicherbereinigung, und offene Typen sind statisch typsicher.

Abbildung 4 zeigt die Definition eines offenen Typs `Person` mit zwei einwertigen Attributen `name` und `vorname` (jeweils vom Typ `string`) und einem benutzerdefinierten Konstruktor zur Erzeugung und Initialisierung eines `Person`-Objekts sowie eine schematische Darstellung des so erzeugten Objekts `p`.

```
typename Person;                // Offener Typ.
Person -> string name;          // Einwertiges Attribut.
Person -> string vorname;       // Dto.

// Konstruktor erzeugt Person-Objekt und initialisiert
// Attribut vorname mit v und Attribut name mit n.
Person (string v, string n) {   // Konstruktor.
  return Person(@vorname, v)(@name, n);
}

// Aufruf des Konstruktors und Verwendung des Objekts p.
p = Person("Christian", "Heinlein");
print(p@vorname, p@name);
```

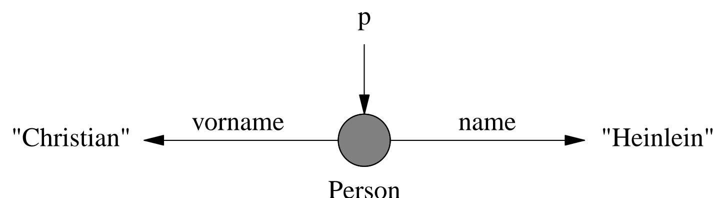

Abbildung 4: Offener Typ mit Attributen und Konstruktor

Bidirektionale Relationen. Neben ein- und mehrwertigen Attributen, bieten offene Typen auch eine direkte Unterstützung für *bidirektionale Relationen*, d. h. Paare von Attributen, die zueinander inverse Abbildungen darstellen. Das besondere hierbei ist, dass bei einer Änderung einer Richtung einer Relation die Gegenrichtung automatisch mitgeändert wird. Abbildung 5 zeigt neben zwei weiteren offenen Typen `Auto` und `Motor` zwei unterschiedliche Arten von Relationen sowie eine exemplarische Beziehungsstruktur zwischen Objekten. (Bei einer anonymen Relation fehlen die Namen der beiden Attribute; in diesem Fall können die jeweiligen Typnamen als „Rollennamen“ verwendet werden.) Wie am Anfang dieses Abschnitts erwähnt, sind Attribute und damit auch Relationen grundsätzlich optional, d. h. ein Auto muss nicht notwendigerweise mit einem Motor in Beziehung stehen und umgekehrt.

```
typename Auto;
Person besitzer <->> Auto autos;         // 1:N-Relation.

typename Motor;
Auto <-> Motor;                  // Anonyme 1:1-Relation.
```

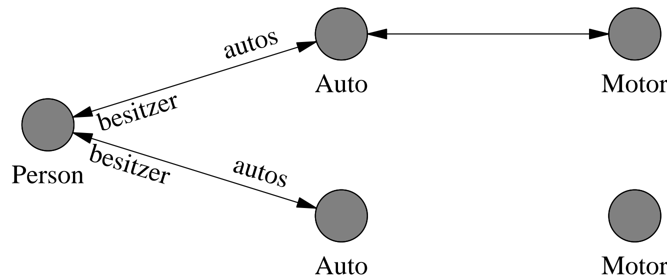

Abbildung 5: Bidirektionale Relationen

Einfache „Vererbung“. Interessanterweise lassen sich mit bidirektionalen Relationen auch Vererbungsbeziehungen modellieren, ohne dass man hierfür spezielle Vererbungsmechanismen benötigt. Die einzige notwendige Erweiterung sind sog. *automatische* Relationen, die vom Compiler bei Bedarf automatisch angewandt werden, um ein Objekt des einen Typs in ein Objekt des anderen Typs umzuwandeln (ähnlich wie in objektorientierten Sprachen ein Objekt eines Untertyps bei Bedarf automatisch in ein Objekt eines Obertyps umgewandelt wird).

Abbildung 6 zeigt, wie sich auf diese Weise `Stud` quasi als Untertyp von `Person` definieren lässt und wie ein Student schrittweise aus einem `Person`- und einem `Stud`-Teilobjekt zusammengesetzt werden kann. Abbildung 7 zeigt zwei benutzerdefinierte Konstruktoren, die diese Objektkonstruktion kapseln, einen Aufruf des zweiten Konstruktors sowie typische „objektorientierte“ Verwendungen eines `Stud`-Objekts. (Die Zweckmäßigkeit des ersten Konstruktors wird später ersichtlich.)

Mehrfache Vererbung. Abbildung 8 zeigt am Beispiel `MnlStud`, wie sich mehrfache Vererbung mit bidirektionalen Relationen modellieren lässt. Die verschiedenen Konstruktoren des Typs und die zugehörigen Abbildungen zeigen, dass man je nach Bedarf sowohl nicht-virtuelle als auch virtuelle Basisklassen nachbilden kann, wobei zur Implementierung des letzten Konstruktors die Konstruktion von `Mann` und `Stud` „aus existierenden Teilobjekten“ (erster Konstruktor in Abb. 7) verwendet wird.

Wiederholte Vererbung. Die Abbildungen 9 und 10 zeigen die Definition der Typen `DplStud` und `MnlDplStud` sowie die zugehörigen Konstruktoren. Im Gegensatz zur C++-Lösung fällt auf, dass keinerlei künstliche Hilfsklassen benötigt werden und dass sich männliche Doppelstudenten problemlos definieren lassen.

```
typename Stud;          // Offener Typ.
Stud -> string fach;    // Attribut.
Stud -> integer matr;   // Attribut.
Stud <->! Person;       // Automat. 1:1-Relationen zu Person.

// Person- und Stud-Objekt erzeugen und miteinander verbinden.
Person p = Person(@vname, "Christian")(@name, "Heinlein");
Stud s = Stud(@fach, "Info")(@matr, 123456);
s(@Person, p);
```

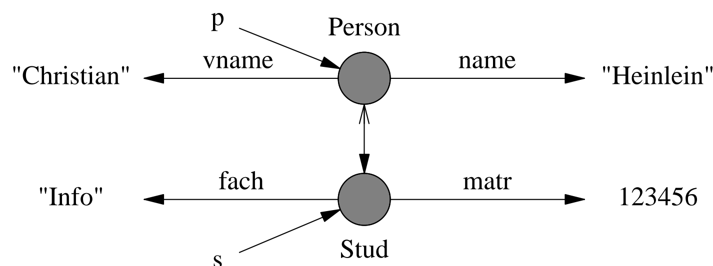

Abbildung 6: Einfache Vererbung mit bidirektionalen Relationen

```
// Konstruktion eines Studenten aus einem exist. Person-Obj.
Stud (Person p, string f, integer m) {
  return Stud(@Person, p)(@fach, f)(@matr, m);
}

// Konstruktion eines Studenten aus atomaren Werten.
Stud (string v, string n, string f, string m) {
  // Obigen Konstruktor aufrufen.
  return Stud(Person(v, n), f, m);
}

// Aufruf des zweiten Konstr. und Verwendung des Objekts s.
Stud s = Stud("Christian", "Heinlein", "Info", 123456);
print(s@fach, s@matr);

// Polymorphe Verwendung von Stud s als Person p.
print(s@name);          // Entspricht: print(s@Person@name)
Person p = s;           // Entspricht: Person p = s@Person;

// Dynamischer Typtest: Ist p ein Student?
if (p@Stud) {
  print(p@Stud@matr); // Expliziter "Downcast" Person -> Stud.
}
```

Abbildung 7: Konstruktion und Verwendung eines `Stud`-Objekts

```
typename Mann;
Mann -> boolean bart;
Mann <->! Person;   // Automatische 1:1-Beziehung zu Person.

// Konstruktoren analog zu Stud.
Mann (Person p, boolean b) { ...... }
Mann (string v, string n, boolean b) { ...... }

typename MnlStud;
MnlStud <->! Mann;  // Automatische 1:1-Beziehung zu Mann.
MnlStud <->! Stud;  // Automatische 1:1-Beziehung zu Stud.

// Konstruktion aus existierenden Teilobjekten.
MnlStud (Mann m, Stud s) {
  return MnlStud(@Mann, m)(@Stud, s);
}

// Konstr. aus atomaren Werten ("schizophren", Abb. links).
MnlStud (string v1, string n1, boolean b,
         string v2, string n2, string f, integer m) {
  Mann m = Mann(v1, n1, b);
  Stud s = Stud(v2, n2, f, m);
  return MnlStud(m, s); // Ersten Konstruktor aufrufen.
}

// Konstruktion aus atomaren Werten (normal, Abb. rechts).
MnlStud (string v, string n, boolean b, string f, integer m)
{
  Person p = Person(v, n);
  Mann m = Mann(p, b);
  Stud s = Stud(p, f, m);
  return MnlStud(m, s); // Ersten Konstruktor aufrufen.
}
```

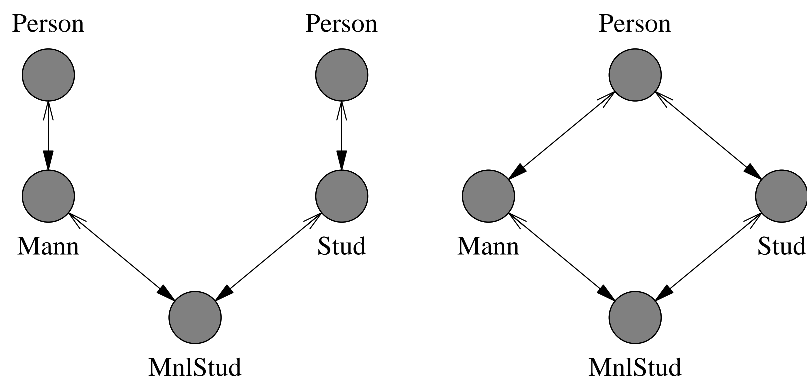

Abbildung 8: Mehrfache Vererbung mit bidirektionalen Relationen

```
typename DplStud;
DplStud <-> Stud StudTeil1; Stud StudTeil1 <->! Person;
DplStud <-> Stud StudTeil2; Stud StudTeil1 <->! Person;

// Konstruktion aus existierenden Teilobjekten.
DplStud (Stud s1, Stud 2) {
  return DplStud(@StudTeil1, s1)(@StudTeil2, s2);
}

// Konstruktion aus atomaren Werten.
DplStud (string v, string n, string f1, integer m1,
                             string f2, integer m2) {
  Stud s1 = Stud(Person(), f1, m1);
  Stud s2 = Stud(Person(), f2, m2);
  Person p = Person(v, n)(@StudTeil1, s1)(@StudTeil2, s2);
  return DplStud(s1, s2); // Obigen Konstruktor aufrufen.
}
```

Abbildung 9: Doppelstudenten

3 Zusammenfassung

Anhand des Beispiels männlicher (und weiblicher) Doppelstudenten wurde gezeigt, dass sich bestimmte Vererbungsprobleme mit gängigen Programmiersprachen (höchstwahrscheinlich) nicht lösen lassen. Offene Typen und bidirektionale Relationen hingegen bieten die notwendige Flexibilität, um derartige Probleme zu lösen, da Objekte von „Untertypen“ nach Belieben aus Objekten der „Obertypen“ zusammengesetzt werden können.

Referenzen

[1] C. Heinlein: "APPLE: Advanced Procedural Programming Language Elements." In: W. Goerigk (ed.): *Programmiersprachen und Rechenkonzepte* (21. Workshop der GI-Fachgruppe; Bad Honnef, Mai 2004). Bericht Nr. 0410, Institut für Informatik, Christian-Albrechts-Universität zu Kiel, January 2005, 59–66.

[2] C. Heinlein: "Open Types and Bidirectional Relationships as an Alternative to Classes and Inheritance." In: M. Hanus, F. Huch (eds.): *Programmiersprachen und Rechenkonzepte* (22. Workshop der GI-Fachgruppe 2.1.4; Bad Honnef, Mai 2005). Bericht Nr. 0513, Institut für Informatik, Christian-Albrechts-Universität zu Kiel, October 2005, 30–39.

[3] C. Heinlein: "Open Types and Bidirectional Relationships as an Alternative to Classes and Inheritance." *Journal of Object Technology* 6 (3) March/April 2007, 101–151, http://www.jot.fm/issues/issue_2007_03/article3.

[4] B. Stroustrup: *The C++ Programming Language* (Special Edition). Addison-Wesley, Reading, MA, 2000.

```
typename MnlDplStud;
MnlDplStud <->! DplStud;
MnlDplStud <-> MnlStud MnlStudTeil1;
                            MnlStud MnlStudTeil1 <->! Mann;
MnlDplStud <-> MnlStud MnlStudTeil2;
                            MnlStud MnlStudTeil2 <->! Mann;

// Konstruktion aus existierenden Teilobjekten.
MnlDplStud (DplStud ds, MnlStud ms1, MnlStud ms2) {
  return MnlStud(@DplStud, ds)
   (@MnlStudTeil1, ms1)(@MnlStudTeil2, ms2);
}

// Konstruktion aus atomaren Werten.
MnlDplStud (string v, string n, boolean b,
            string f1, integer m1, string f2, integer m2) {
  DplStud ds = DplStud(v, n, f1, m1, f2, m2);
  MnlStud ms1 = MnlStud(Mann(), ds@StudTeil1);
  MnlStud ms2 = MnlStud(Mann(), ds@StudTeil2);
  Mann m =
   Mann(ds, b)(@MnlStudTeil1, ms1)(@MnlStudTeil2, ms2);
  return MnlDplStud(ds, ms1, ms2); // Ersten Konstr. aufrufen.
}
```

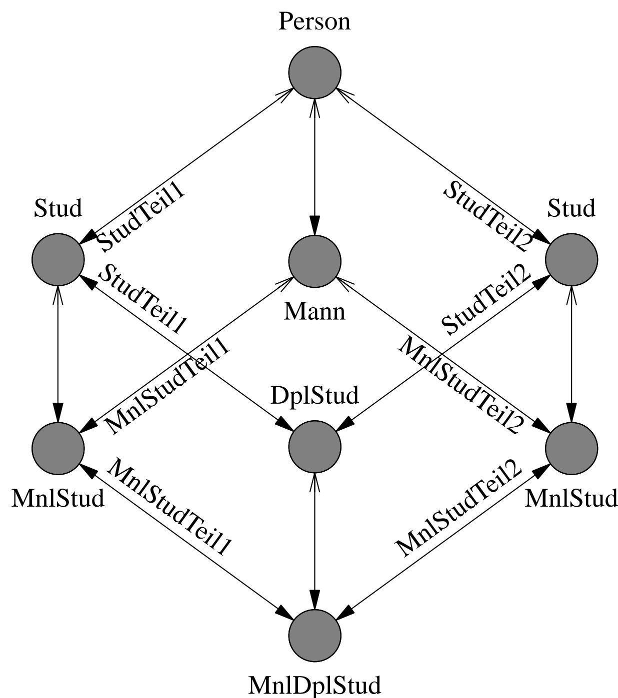

Abbildung 10: Männliche Doppelstudenten

Statisches Typen von JavaScript-Programmen

Phillip Heidegger

Institut für Informatik, Universität Freiburg, Georges-Köhler-Allee 079,
79110 Freiburg, Germany
heidegger@informatik.uni-freiburg.de

Der Internetuser ist in den letzten Jahren deutlich anspruchsvoller geworden. Als Folge reicht es nicht mehr aus, auf eine HTTP Anfrage mit statischem (X)HTML zu reagieren. Vielmehr ist es heute üblich dass Webseiten Ihre Daten dynamisch per HTTP Requests vom Server anfragen, falls diese gebraucht werden. Um dieses Verhalten zu programmieren gibt es zwei Ansätze. Zuerst einmal ist es möglich mit Webtoolkits wie z.B. GWT (Google Web Toolkit) JavaScript Code zu generieren, oder der entsprechende JavaScript Code wird von Hand vom Programmierer geschrieben. Der erste Ansatz bietet eine deutlich höhere Zuverlässigkeit, da eine große Anzahl Fehler durch die richtige Wahl der Quellsprache vermieden werden kann. Leider geht durch den Einsatz eines solchen Toolkits häufig ein Teil der Flexibilität verloren. Falls dies notwendig ist, muss der Programmierer auf JavaScript zurückgreifen.

JavaScript Programme statisch zu analysieren ist wegen vieler Eigenschaften der Programmiersprache schwierig. Im Vortrag werden zuerst kurz anhand von einigen wenigen Beispielen typische JavaScript Programmierfehler aufgezeigt.

Anschließend wird ein Typesystem vorgestellt, dass eine Teilmenge von JavaScript analysiert und dennoch übliche Muster der JavaScript Programmierer ermöglicht.

Progress und Preservation sind für das Typsystem bewiesen. Das Erstellen eines Inferenzalgorithmus ist zur Zeit in Arbeit und steht als Diskussionsthema zur Verfügung. Ziel der Inferenz ist es, JavaScript Code mit möglichst wenig Annotationen versehen zu müssen, und so die Migrationsarbeit von ungetypten JavaScript Code zu minimieren.

Type inference for Java(X)
[Abstract]

Markus Degen, Peter Thiemann, and Stefan Wehr

Institut für Informatik, Universität Freiburg
{degen,thiemann,wehr}@informatik.uni-freiburg.de

1 Abstract

JAVA(X) is a framework for type refinement. It extends Java's type language with annotations drawn from an algebra X and structural subtyping in terms of the annotations. Each instantiation of X yields a different refinement type system with guaranteed soundness [1].

JAVA(X) has a concept of activity annotations paired with the notion of droppability. An activity annotation is a capability which can grant exclusive write permission for a field in an object and thus facilitates a typestate change (strong update). Propagation of capabilities is either linear or affine (if they are droppable). Thus, JAVA(X) can perform protocol checking as well as refinement typing.

To enable a type inference algorithm for JAVA(X) we setup a constraint type system and a constraint solver. The main concerns were addressed to the behavior of the ternary splitting relation and its impact on the complexity of the constraint solver.

Luckily, against the first intuition, the splitting relation for alias handling does not increase the complexity of the constraint solver. Since the splitting may completely be forced by one of the components of the splitting relation, the corresponding constraint may be solved directly without additional guessing or further, potentially exponential many, constraints.

As prove of concept we implemented the provided type inference for JAVA(X) and gained a running system with useful error messages for the programmer.

References

1. M. Degen, P. Thiemann, and S. Wehr. Tracking linear and affine resources with java(x). In *21st ECOOP*, LNCS, pages 550–574, Berlin, Germany, July 2007. Springer.

Resolving of Intersection Types in Java

Martin Plümicke

University of Cooperative Education Stuttgart
Department of Information Technology
Florianstraße 15, D–72160 Horb
m.pluemicke@ba-horb.de

Abstract. In the past we analyzed typeless Java programs. One of our results was, that there may be different correct typings for one method. This means that the principal types of such methods are intersection types. We presented a type-inference algorithm. For typeless Java methods the algorithm infers its principal intersection type. Unfortunately, like Java byte-code, Java does not allow intersection types.
In this paper we present an algorithm, which resolves intersection types of Java methods, such that Java programs with standard typings are generated.
Additionally, we will refine the definition of Java method principal types.

1 Introduction

In [7, 6] we presented a type inference algorithm for a core Java language. The algorithm allows us to write typeless Java programs. The algorithm determines all correct possible typings. One of the results of [7] is that typeless Java methods can have more than one correct typing, which means that the method is typed by an intersection of function types. The type inference algorithm infers this intersection type.
We have implemented the algorithm as an Eclipse plugin. As the Java byte-code does not allow intersection types, we implemented the plugin, such that after type inference the user has to select one of the possible typings. The byte-code is generated only for this selected typing.
It is our purpose to improve the plugin such that type selection is no longer necessary. This means that byte-code can be generated for methods with inferred intersection types.
For this paper we need to consider some important definitions from [6]. Let $\mathsf{SType}_{TS}(\,BTV\,)$ be the set of usual Java types. In the formal definition ([6], Def. 3) BTV stands for the set of (bounded) type variables ([6], Def. 1). We call the elements of $\mathsf{SType}_{TS}(\,BTV\,)$ *simple types.* In Java programs simple types describe types of fields, types of methods' parameters, return types of methods, and types of local variables. For the type inference we need intersections of function types, which describe the types of methods. A *function type* is given as $\theta_1 \times \ldots \times \theta_n \to \theta_0$, where $\theta_i \in \mathsf{SType}_{TS}(\,BTV\,)$. An *intersection type* is an intersection $ty_1 \& \ldots \& ty_m$ of function types $ty_1, \ldots, ty_n$. There is a *subtyping*

ordering on simple types. We denote this by $\leq^*$ (cp. [6], Def. 5). Finally there are arguments of type terms, which are *wildcard types*. There are two kinds of wildcard types "`? extends` ty" respectively "`? super` ty". We abbreviate them by $_?ty$ respectively by $^?ty$. For further definitions we refer to [6], Section 2.
The type inference algorithm determines types of methods, which are given as Java code without types. The inferred method types probably contain intersection types. Unfortunately, standard Java does not include intersection types for methods. This means that there is no canonical strategy to generate byte-code for methods with intersection types. As there is an isomorphic mapping from the signature of the standard Java methods to the signature of the corresponding methods in the byte-code, the intersection types must be resolved for code generation. In this paper we present the resolving of intersection types by transforming the Java code with inferred intersection types to equivalent standard Java code (without intersection types).
The paper is organized as follows. In the next section we define the semantics of method calls for methods with intersection types. In the third section we give a first approach to resolve intersection types. We will see that this approach leads to incorrect and inefficient Java programs. In the fourth section we consider call-graphs of Java methods. Call-graphs are the base of our intersection type resolving algorithm. We present the corresponding algorithm in the fifth section. In the sixth section we give a refined definition for Java method's principal types, which is based on the ideas of intersection type resolving. Then we consider related work on intersection types and principal typings. Finally we close with a summary and an outlook.

2 Semantics of type-inferred Java programs

The semantics of a type-inferred Java program is defined straightforwardly. All control-structures have the same semantics as in standard Java (e.g. [4], [1]). Only the semantics of the method calls differ, as there are intersection types.
The main idea to define the semantics of method calls for methods with intersection types is the following: One typing of the intersection type of the method is determined by the argument types of the method calls. The method is then executed with this typing.

Definition 1 (Semantics of method calls). *Let a Java method `m` be given, where one typing of its intersection type is selected. This means that the method corresponds to a standard typed Java method.*

```
... m ( ... ) { ... receiver.method(t1, ..., tn); ... }
```

The method `m` contains a method call `receiver.method(t1, ..., tn)`, where `receiver` has the type recty and `t1` ... `t2` have the types $ty_1 \ldots ty_n$, respectively. Furthermore the result of the method call has the type rettype.
Then, the smallest class is determined, which is a supertype from rectty with the method `method`, where $ty'_1 \times \ldots \times ty'_n \to ty'$ is one element of its intersection

type and for $1 \leqslant i \leqslant n$ the types ty'_i are the smallest supertypes of ty_i, respectively, and ty' is a subtype of rettype. Then this method `method` *is executed with the corresponding typing.*

Example 1. Let the following typeless Java program be given:

```
class OL {
    Integer m(x) { return x + x; }
    Boolean m(x) { return x || x; }
}
```

```
class Main {
    main(x) { ol;
              ol = new OL();
              return ol.m(x); } }
```

By type-inference the following typings are determined:

$$\begin{aligned}&\texttt{OL.m} : \texttt{Integer} \to \texttt{Integer}\\&\texttt{OL.m} : \texttt{Boolean} \to \texttt{Boolean}\\&\texttt{Main.main} : \texttt{Integer} \to \texttt{Integer}\ \&\ \texttt{Boolean} \to \texttt{Boolean}\end{aligned}$$

Let the Java program be extended by the simple class `simpleClass` with the method `new_meth`:

```
class simpleClass {
    new_meth(x) {
        Main rec = new Main();
        Integer r = rec.main(x); } }
```

We consider the method call `rec.main(x)`. The typing is given as: `rec`:`Main`, `x`:`Integer`, and `main`:`Integer` → `Integer`.
The class `Main` itself is the smallest class, which is a supertype of `Main` with the method `main`, where `Integer` → `Integer` is one element of its intersection type, `Integer` is the smallest supertype of `Integer`, and the return type `Integer` of `main` is also a subtype of the demanded type `Integer`.
This means that the `main` method in the class `Main` is called with the following typing:

```
Integer main(Integer x) {
    OL ol;
    ol = new OL();
    return ol.m(x); }
```

3 First approach

As a first approach to resolve inferred intersection types a new overloaded Java method with standard typing is generated for each element of the intersection type.
Considering the following examples, we will recognize, that this strategy works only in some cases.

Example 2. Let the Java program from Example 1 be given again. As said above the following typings are determined by type-inference:

$$\texttt{OL.m} : \texttt{Integer} \rightarrow \texttt{Integer}$$
$$\texttt{OL.m} : \texttt{Boolean} \rightarrow \texttt{Boolean}$$
$$\texttt{Main.main} : \texttt{Integer} \rightarrow \texttt{Integer} \;\&\; \texttt{Boolean} \rightarrow \texttt{Boolean}$$

If we generate an own Java method for each element of the intersection type we get the `main` class:

```
class Main {
    Integer main(Integer x) {
        OL ol;
        ol = new OL();
        return ol.m(x); }
    Boolean main(Boolean x) {
        OL ol;
        ol = new OL();
        return ol.m(x); } }
```

The result is a correct Java program.

The next example shows that this strategy does not work in all cases.

Example 3. Let the following Java program be given, which implements the multiplication of two integer matrices.

```
class Matrix extends Vector<Vector<Integer>> {
    mul(m){
      ret; ret = new Matrix();
      Integer i = 0;
      while(i <size()) {
         v1; v1 = this.elementAt(i);
         v2; v2 = new Vector<Integer>();
         Integer j = 0;
         while(j < v1.size()) {
            Integer erg = 0;
            Integer k = 0;
            while(k < v1.size()) {
               erg = erg + v1.elementAt(k) * m.elementAt(k).elementAt(j);
               k++; }
            v2.addElement(new Integer(erg));
            j++; }
         ret.addElement(v2);
         i++; }
      return ret; }
}
```

By type-inference the following typings are determined: `mul:` $\&_{\beta,\alpha}(\beta \rightarrow \alpha)$, where

$\beta \leq^* \texttt{Vector<? extends Vector<? extends Integer>>}$,
$\texttt{Matrix} \leq^* \alpha$

This means, that for each pair of arguments and result types a new method `mul` would be generated:

```
class Matrix extends Vector<Vector<Integer>> {
  Matrix mul(Vector<? extends Vector<? extends Integer>> m) { ... }
  Matrix mul(Vector<? extends Vector<Integer>> m) { ... }
  Matrix mul(Vector<Vector<Integer>> m) { ... }
  ...
  Vector<Vector<Integer>> mul(Vector<Vector<Integer>> m) { ... }
  ...
  Vector<? extends Vector<? extends Integer>> mul(Matrix m) { ... } }
```

This is not a correct Java program. There are two conflicts. On the one hand it is not allowed for overloaded methods that argument types have different instances of the same generic type. For example this means for the method `mul`, that the argument type `Vector<?Vector<Integer>>` of one declaration is not allowed, if there is another method declaration with the argument type `Vector<?Vector<?Integer>>`. This is no type theoretical problem. But as the byte-code works only with raw types, there is no difference between both argument types during execution.
On the other hand different method declarations of `mul`, where the argument type is in all declarations equal, e.g. `Vector<Vector<Integer>>`, lead to ambiguity.

If we consider the generated method declarations more accurately, we recognize that in all declarations the same code is executed. One approach to solve the problem could be to generate one declaration with a supertype of all other declarations: `mul : Vector<?Vector<?Integer>>` $\rightarrow$ `Matrix`. The argument type `Vector<?Vector<?Integer>>` is the supertype of all other argument types and `Matrix` is the subtype of all other result types. Later on, we will call this function type the supertype of all possible typings.
Now, we will present the algorithm, which resolves the intersection types, such that incorrect and unnecessary copies of methods are avoided.

4 Call-graph

We consider call-graphs of Java methods as the base of the intersection type resolving algorithm. Call-graphs are graphs of method declarations, which contain all methods, that are called during the execution, for a given method with one typing.

Definition 2 (Call-graph). *Let p be a Java program containing one or more classes, where the (intersection) types of the methods are inferred. Furthermore let the triple $cl.m : \tau$ be given, where m is a declared method in the class cl and τ is an instance of one element of the inferred intersection type. The* call-graph $\mathcal{CG}(\, cl.m : \tau\,)$ *is given as the pair (M, MC), where M is the set of declared methods. $MC \subseteq M \times M$ is given as the smallest set with the following properties:*

- *Let $cl.m : ty \in M$, where the function type τ is an instance of an element of the intersection type ty. It holds $(cl.m : ty, cl'.m' : ty') \in MC$ for all methods m', which are called in $cl.m : ty$, if m has the function type τ.*

– *If* $(cl.m : ty, cl'.m' : ty') \in MC$ *and* $cl'.m' : ty'$ *is called with function type* τ'*, then* $(cl'.m' : ty', cl''.m'' : ty'') \in MC$ *for all methods* m''*, which are called in* $cl'.m' : ty'$*, if* m' *has the function type* τ'*.*

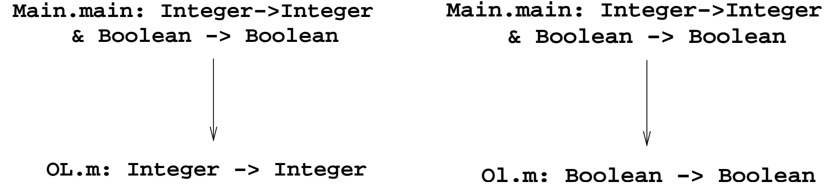

Fig. 1. $\mathcal{CG}($ `Main.main` : `Integer` $\rightarrow$ `Integer` $)$, $\mathcal{CG}($ `Main.main` : `Boolean` $\rightarrow$ `Boolean` $)$

Example 4. Let the Java program from Example 1 be given again. From the inferred typings the following call-graphs are determined:
$\mathcal{CG}($ `Main.main` : `Integer` $\rightarrow$ `Integer` $)$ is the left call-graph in Fig. 1. In this case $\tau =$ `Integer` $\rightarrow$ `Integer`. In `Main.main` with the type τ the method `m` with function type `Integer` $\rightarrow$ `Integer` is called.
$\mathcal{CG}($ `Main.main` : `Boolean` $\rightarrow$ `Boolean` $)$ is the right call-graph in Fig. 1. In this case $\tau =$ `Boolean` $\rightarrow$ `Boolean`. In `Main.main` with the type τ the method `m` with function type `Boolean` $\rightarrow$ `Boolean` is called. It is obvious that for each different type of `main` different methods are called.

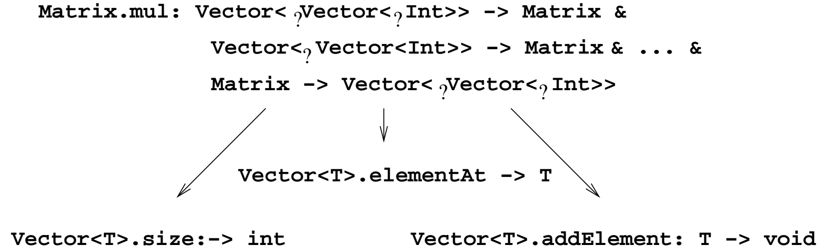

Fig. 2. The call-graph $\mathcal{CG}($ `Matrix.mul` : τ $)$ for all τ

Example 5. Let us consider again the Java program `Matrix` (Example 3). In this case for all function types τ of the triple `Matrix.mul`:τ the call-graphs $\mathcal{CG}($ `Matrix.mul` : τ $)$ are the same. The call-graph is given in Fig. 2.

In the two given examples the inferred types contain no generics. Therefore no instances of elements of intersection types are considered. The next example presents a call-graph of an instance of an element of an intersection type.

Example 6. Let the following Java program be given:

```
class Put {
    <T> putElement(T ele, Vector<T> v) { v.addElement(ele); }
    <T> putElement(T ele, Stack<T> s) { s.push(ele); }

    main(ele, x) { putElement(ele, x); } }
```

The inferred intersection type of `main` is

$$\texttt{main} : \texttt{T} \times \texttt{Vector<T>} \rightarrow \texttt{void} \ \& \ \texttt{T} \times \texttt{Stack<T>} \rightarrow \texttt{void}.$$

The call-graph $\mathcal{CG}(\texttt{Put.main} : \texttt{Integer} \times \texttt{Stack<Integer>})$ is given as:

```
Put.main: T x Vector<T> -> void
     & T x Stack<T> -> void
              |
              v
Put.putElement: T x Stack<T> -> void
```

As $\texttt{Integer} \times \texttt{Stack<Integer>} \rightarrow \texttt{void}$ is an instance of $\texttt{T} \times \texttt{Stack<T>} \rightarrow \texttt{void}$ in `main` the method $\texttt{putElement:T} \times \texttt{Stack<T>} \rightarrow \texttt{void}$ is called.

The following *intersection type resolving algorithm* bases on the call-graph.

5 The resolving algorithm

In this section we describe the algorithm, which resolves the intersection types, such that standard Java programs with standard types are generated.
Before we can present the algorithm we have to generalize the definition of the subtyping ordering to function types.

Definition 3 (Subtyping on function types). *Let the subtyping ordering $\leq^*$ on simple types ([6], Def. 5) be given. For two function types $ty_1 = \theta_1 \times \ldots \times \theta_n \rightarrow \theta'$ and $ty_2 = \theta'_1 \times \ldots \times \theta'_n \rightarrow \theta$ holds: ty_1 is a* subtype *of ty_2 if for $1 \leqslant i \leqslant n$ holds $\theta_i \leq^* \theta'_i$, respectively, and $\theta \leq^* \theta'$. We call the maximum in the subtyping ordering* supertype.

Example 7. The function type $\texttt{Matrix} \rightarrow \texttt{Vector<? extends Vector<Integer>>}$ is a subtype of $\texttt{Vector<Vector<Integer>>} \rightarrow \texttt{Matrix}$ as it holds

$$\texttt{Matrix} \leq^* \texttt{Vector<Vector<Integer>>} \text{ and}$$
$$\texttt{Matrix} \leq^* \texttt{Vector<? extends Vector<Integer>>}.$$

In contrast $\texttt{Integer} \rightarrow \texttt{Integer}$ is no subtype of $\texttt{Number} \rightarrow \texttt{Number}$, as `Integer` $\leq^*$ `Number`, but not vice versa.

Now we are able to present the algorithm.

The algorithm

Input: A Java program p consisting of different classes with inferred (intersection) types of its methods.

Output: A Java program p' consisting of the same classes as p, where the methods have standard Java types. The semantics of p and p' are equal.

1. Step: For every class cl in p consider for each method m the intersection type ty_m:
  - Build the call-graph $\mathcal{CG}(\, cl.m : \tau\,)$ for each function type τ of the intersection type ty_m.
  - Group all elements τ of ty_m, where $\mathcal{CG}(\, cl.m : \tau\,)$ is the same graph and there is a supertype.

2. Step: Determine the supertype of the respective group.

3. Step: Generate for each group of function types the corresponding Java code with the supertype as standard typing in p'.

Example 8. Let us consider the Java program from Example 1 as input p.

1. Step: The only method, where an intersection type is inferred, is `main` in the class `Main`. The intersection type is given as

$$\texttt{Integer} \rightarrow \texttt{Integer}\ \&\ \texttt{Boolean} \rightarrow \texttt{Boolean}.$$

  - The call-graphs are given in Example 4.
  - There are two groups, each with one element:
    $\{$ `Main.main` : `Integer` $\rightarrow$ `Integer` $\}$ and
    $\{$ `Main.main` : `Boolean` $\rightarrow$ `Boolean` $\}$, as there are different call-graphs.

2. Step: The respective supertypes are also
`Integer` $\rightarrow$ `Integer` and `Boolean` $\rightarrow$ `Boolean`.

3. Step: The corresponding Java code, which is added to p' is given as

```
Integer main(Integer x) {
    OL ol;
    ol = new OL();
    return ol.m(x); }
```

```
Boolean main(Boolean x) {
    OL ol;
    ol = new OL();
    return ol.m(x); }
```

If we consider the result it is obvious that the result is the same as in the first approach (cp. Example 2). In the following example we will see some differences in comparison to the first approach.

Example 9. Let the Java program `Matrix` from Example 3 be given as input.

1. Step: The intersection type of `mul` is given as: `mul:` $\&_{\beta,\alpha}(\beta \rightarrow \alpha)$, where

$\beta \leq^*$ `Vector<? extends Vector<? extends Integer>>`,
`Matrix` $\leq^* \alpha$

  - The call-graph for all elements τ of the inferred intersection type of `mul` is given in Fig. 2.

– As there is a supertype Vector<?Vector<?Integer>> $\rightarrow$ Matrix for all elements, there is only one group, which contains all elements τ.

2. Step: The supertype is determined as: Vector<?Vector<?Integer>> $\rightarrow$ Matrix.

3. Step: The corresponding Java code, which is added to p' is given as:

```
Matrix mul(Vector<? extends Vector<? extends Integer>> m) { ... }
```

The example shows that the algorithm solves the problems, which arose in the first approach (cp. Example 3).
The following theorem shows the correctness of the algorithm.

Theorem 1. *Let p be a Java program and p' be the result of applying the intersection type resolving algorithm. The semantics of p equals to the semantics of p'*

6 Principal type

In the following we will refine the definition of principal types (Def. 8 [6]) for Java methods.

Definition 4 (Principal type). *An intersection type of a method*
$\mathtt{m} : (\theta_{1,1} \times \ldots \times \theta_{1,n} \rightarrow \theta_1)$ & $\ldots$& $(\theta_{m,1} \times \ldots \times \theta_{m,n}, \rightarrow \theta_m)$
in a class Cl *is called* principal *if for any correct type annotated method declaration* ***rty*** m(***ty1*** a1, ..., ***tyn*** an) { ... } *there is an element* $(\theta_{i,1} \times \ldots \times \theta_{i,n}, \rightarrow \theta_i)$ *of the intersection type and there is a substitution* σ*, such that* $\sigma(\theta_i) \leq^* \mathbf{rty}$*,* $\mathbf{ty1} \leq^* \sigma(\theta_{i,1})$ *, ..., * $\mathbf{tyn} \leq^* \sigma(\theta_{i,n})$ *and*

$$\mathcal{CG}(\ Cl.m : \theta_{i,1} \times \ldots \times \theta_{i,n}, \rightarrow \theta_i\ ) = \mathcal{CG}(\ Cl.m :\ \mathbf{ty1} \times \ldots \times \mathbf{tyn} \rightarrow \mathbf{rty}\ )$$

This refined definition guarantees, that for each method, which is generated by the resolving algorithm, at least one typing is contained in the principal type.

Example 10. In Example 6 the principal type of main is
main : T $\times$ Vector<T> $\rightarrow$ void & T $\times$ Stack<T> $\rightarrow$ void.

7 Related Work

Besides our introduction of intersection types for methods in Java with type inference, in standard Java there are intersection types of simple types during compile-time [4], §4.9). Intersection types arise in the processes of capture conversion and type inference during method invocation. In Java it is not allowed to write an intersection type as a part of a program.
Basically, the intersection type discipline was introduced by Coppo and Dezani [2]. In the type system of Damas and Milner [3] some λ–terms are not typable. Therefore the type system is extended by intersection types. The type inference problem for these type systems is not decideable in general (e.g. [5]). Our Java

type system is a restriction of them. In comparison, our Java type system contains no λ–terms. This means that we do not have the function type constructor $\rightarrow$ and no higher-oder functions. Instead of that, Java has the function template $(ty_1, \ldots, ty_n) \rightarrow ty$.
In [8] a general definition of principal type property is given. Our definition (Def. 4) as well as the definition of Damas and Milner [3] satisfies the definition of [8].

8 Conclusion and Outlook

Beginning with the result of [7, 6] that the inferred principal types of typeless Java methods can be intersection types, we showed how they can be resolved. This means that type inferred Java programs with intersection types can be translated in Java byte-code. In the Eclipse plugin type selection is no longer necessary.
Some properties of the resolving algorithm lead to a refined definition of a Java method's principal type. The refined version guarantees that for each method, which is generated by the resolving algorithm, at least one element is contained in the principal type.
Further investigation is necessary to optimize the procedure: *type inference algorithm*, *intersection type resolving algorithm*, and *code generation*. At the moment the type inference algorithm infers some types for the methods, which are erased again in the second step of the resolving algorithm.

References

1. E. Börger and W. Schulte. A programmer friendly modular definition of the semantics of java. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *Lecture Notes in Computer Science*, pages 353–404. Springer, 1999.
2. M. Coppo and M. Dezani-Ciancaglini. An extension of the basic functionality theory for the λ–calculus. *Notre Dame J. Formal Logic*, 21(4):685–693, 1980.
3. L. Damas and R. Milner. Principal type-schemes for functional programs. *Proc. 9th Symposium on Principles of Programming Languages*, 1982.
4. J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java^{TM} Language Specification*. The Java series. Addison-Wesley, 3rd edition, 2005.
5. D. Leivant. Polymorphic type inference. *Proc. 10th Symposium on Principles of Programming Languages 1982*, 1983.
6. M. Plümicke. Typeless Programming in Java 5.0 with wildcards. In V. Amaral, L. Veiga, L. Marcelino, and H. C. Cunningham, editors, *5th International Conference on Principles and Practices of Programming in Java*, ACM International Conference Proceeding Series, pages 73–82, September 2007.
7. M. Plümicke and J. Bäuerle. Typeless Programming in Java 5.0. In R. Gitzel, M. Aleksey, M. Schader, and C. Krintz, editors, *4th International Conference on Principles and Practices of Programming in Java*, ACM International Conference Proceeding Series, pages 175–181. Mannheim University Press, August 2006.
8. S. van Bakel. Principal type schemes for the strict type assignment system. *Journal of Logic and Computing*, 3(6):643–670, 1993.

Graph Parser Combinators: A Challenge for Curry-Compilers

Steffen Mazanek, Mark Minas

Universität der Bundeswehr München, Germany,
{steffen.mazanek|mark.minas}@unibw.de

1 Introduction

In a recent paper [1] we have shown that graph parser combinators[1] are an ideal playing ground for functional-logic programming languages [2]. Our approach makes heavy use of key features of both the functional and the logic programming paradigm: Higher-order functions allow the treatment of parsers as first class citizens. Non-determinism and logical variables are beneficial for dealing with errors and incomplete information. Parsers can even be applied backwards and, thus, be used as generators or for graph completion. This feature is very useful in the domain of diagram editors where it can be exploited for the computation of diagram completions [3].

The framework proposed in [1] has been implemented in Curry[2], a popular, actively developed functional-logic programming language. There are several different Curry compilers[3]. For instance, the Münster Curry Compiler MCC is a native code compiler that generates efficient programs. The Portland Aachen Kiel Curry System PAKCS compiles Curry to Prolog, a high-level language for which several efficient compilers exist. The complementary approach, i.e. compiling to Haskell, has been realized with the Kiel Curry System KiCS recently [4]. While implementing our graph parser combinators we have noted that each of these compilers has specific strengths and weaknesses. Furthermore, they support different experimental language extensions, some of which have appeared to be very useful in our setting. Thus, unfortunately, we ended with three versions of our library.

In this report we summarize and share these experiences. Using a practically relevant application of functional-logic programming we hope to give implementors a hint which aspects we have found particularly important. Furthermore, our application can serve as a benchmark for Curry compilers: With graph parsing we can assess the performance of compilers in dealing with non-determinism and large numbers of logical variables. We also describe a meaningful application of extensive sets of disequality constraints in this context. All in all, we hope that our discussion will have an impact on the future development of the Curry programming language.

[1] Project website: `http://www.unibw.de/steffen.mazanek/grappa`, 30.06.2008

[2] `http://www.informatik.uni-kiel.de/~curry/report.html`, 30.06.2008

[3] `http://www.informatik.uni-kiel.de/~curry/implementations.html`, 30.06.2008

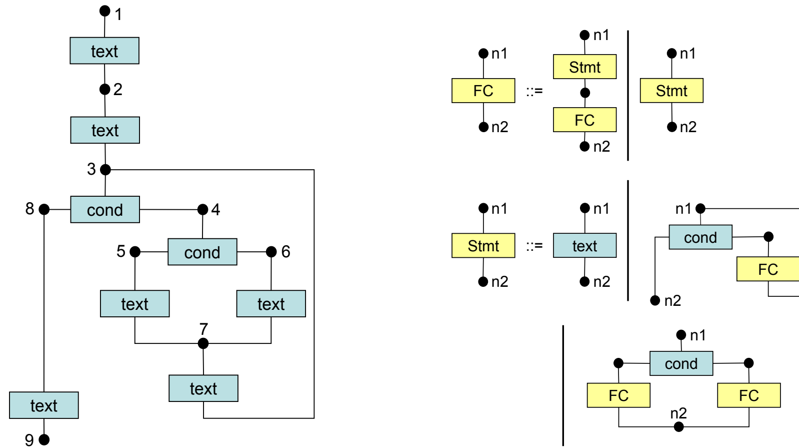

Fig. 1. a) Example flowgraph and b) HRG of flowgraphs

2 Graph Parser Combinators in brief

In this section we briefly introduce our application domain. We give an informal notion of graph grammars and introduce the combinators as our benchmark. A more precise introduction is given in [1].

2.1 Hypergraphs and Hyperedge Replacement Grammars

Hyperedge replacement grammars HRG are a well-known way of describing languages of hypergraphs, i.e., graphs where edges are allowed to visit an arbitrary number of nodes (depending on the label of the edge). Although restricted in power, this formalism comprises several beneficial properties: It is context-free and still quite powerful. Grammars are comprehensible, and reasonably efficient parsers can be defined for practical languages. Derivations are constructed by replacing special non-terminal hyperedges with new hypergraphs, which are glued to the given graph by fusing particular nodes [5].

We clarify this notion using flowgraphs as an example. This graph language is a natural model for the visual language of structured flowcharts, i.e. flowcharts representing structured programs. In Fig. 1a an example flowgraph is shown. Edges are represented by a rectangular box marked with the particular label. The black dots represent nodes that we have additionally marked with numbers. A line between an edge and a node indicates that the node is visited by this edge.

The language of flowgraphs can be described using an HRG in a straightforward way. A set of productions that generate flowgraphs is given in Fig. 1b. The used notation is similar to BNF rules as known from string grammars. Nonterminal edges are highlighted by using another color. Nodes in a production act

as variables. In order to apply a production they have to be instantiated with nodes actually occurring in the graph. We use labels to identify corresponding nodes of both sides of a production. As usual, a language defined by an HRG consists of all graphs whose edges are labeled only with terminal labels and that can be derived in an arbitrary number of steps from the start symbol [5].

2.2 Graph Parser Combinators in Curry

The following Curry code introduces the basic data structures for representing graphs. For the sake of simplicity, we represent nodes by integer numbers and edge labels by strings (although we do not rely on any particular type at all). We declare a graph as a list of labeled edges with the according visited nodes. The actual order of edges does not matter.

```
type Node = Int
type Edge = (String, [Node])
type Graph = [Edge]
```

Therewith, the flowgraph given in Fig. 1b can be represented as follows:

```
ex = [("text",[1,2]),("text",[2,3],("cond",[3,8,4]),("cond",[4,5,6]),
      ("text",[5,7]),("text",[6,7]),("text",[7,3]),("text",[8,9])]
```

Next, we provide a simple declaration of the type `Grappa` representing a graph parser. A graph parser is a (non-deterministic) function that maps graphs to the graphs that remain after successful parser application. In contrast to Haskell, we do not have to deal with parsing errors and backtracking explicitly (no need for "lists of successes"). Instead, similar to [6], we rely on the non-deterministic notion of functions inherent to functional-logic programming languages like Curry.

```
type Grappa = Graph -> Graph
```

We proceed by defining some important primitives for the construction of graph parsers. An especially important primitive parser is `edge`. It only succeeds if the given edge `e` is part of the particular graph `g`. In this case, `e` has to be consumed. The primitive `edge` can be implemented in a logic programming style making use of an equational constraint:

```
edge::Edge -> Grappa
edge e g | g=:=(g1++e:g2) = g1++g2
           where g1, g2 free
```

The primitive `eoi` can be used to enforce that the whole graph is consumed:

```
eoi::Grappa
eoi [] = []
```

```
stmts::(Node,Node) -> Grappa
stmts (n1,n2) = stmt (n1,n2)
stmts (n1,n2) = stmt (n1,n) <*>
                stmts (n,n2)
                where n free

stmt::(Node,Node) -> Grappa
stmt (n1,n2) = edge ("text", [n1,n2])
stmt (n1,n2) = edge ("cond", [n1,nno,nyes]) <*>
               stmts (nno,n2) <*>
               stmts (nyes,n2)
               where nno, nyes free
stmt (n1,n2) = edge ("cond", [n1,n2,nbody]) <*>
               stmts (nbody,n1)
               where nbody free
```

Fig. 2. A parser for flowgraphs

Parser combinators then can be defined in a fairly standard way (see [6]). The choice operator `<|>` combines two parsers and succeeds for a given graph `g`, if either the first or the second parser succeeds when applied to `g`:

```
(<|>)::Grappa -> Grappa -> Grappa
(p1 <|> p2) g = p1 g
(p1 <|> p2) g = p2 g
```

Two parsers can also be combined via `<*>`, the successive application. The second parser thereby starts with the input the first parser has left, i.e., `<*>` basically is function composition. We have to use a constraint (or `case`, alternatively) to ensure the intended order of evaluation:

```
(<*>)::Grappa -> Grappa -> Grappa
(p1 <*> p2) g | p1 g =:= g' = p2 g'
                 where g' free
```

In Fig. 2 the parser for flowgraphs is presented. The translation from the grammar is quite straightforward [1]. Note that we do not need to know the inner nodes of a production in advance. We simply define them as free variables, which can be instantiated according to the Curry narrowing semantics. Representing graph nodes as free variables actually is a functional-logic design pattern [7] here exploited in a novel way.

In contrast to string parsing the order of parsers in a successive composition via `<*>` is not that important as long as left recursion is avoided. Nevertheless, the chosen arrangement might have an impact on the performance. Usually, it is advisable to deal with the terminal edges first.

Parsing is not the only thing we can do with these functions. We can also apply them backwards to construct graphs of the language. That way we can

enumerate all graphs in the language up to a particular size. However, with the quite naive framework presented here a lot of list permuations are returned as redundant results (in the next section we show how to avoid this issue). We can further use the parser to perform a kind of auto-completion. Say, the edge `("text",[2,3])` in the graph given in Fig. 1a is missing, such that the flowgraph is not a member of the language anymore (it consists of two correct flowgraphs though). We can try inserting an edge `e` as a free variable and see how `e` is instantiated by the parser. One of the possible completions consuming the whole input is `("text",[2,3])`, the edge we deleted.

3 Suitability of Curry Compilers for Graph Parsing

In this section we discuss the benefits of the compilers MCC, PAKCS and KiCS with respect to graph parsing.

Table 1. Overview over compiler's features relevant to graph parsing

MCC	PAKCS	KiCS
native code generation	compiles to Prolog	compiles to Haskell
type classes [8]	function patterns [9]	narrowing on numbers [10]
disequality constraints	finite domain constraints	search control [4]

3.1 Disequality Constraints for Ensuring Correctness of Parsers

As it is, the flowgraph parser given in Fig. 2 accepts too many graphs. The so-called identification condition and the dangling edge condition have to be enforced to ensure correctness [5]. The former states that matches have to be injective, i.e., involved nodes have to be pairwise distinct; the latter states that there must not be other edges in the remaining graph visiting inner nodes of a match.

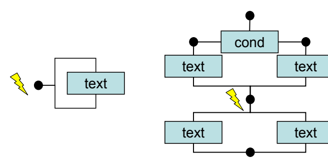

Fig. 3. Violation of identification and dangling edge condition

In fact, both conditions need to be ensured by additional checks. Therefore, the disequality constraints as provided by MCC (`=/=`) are very handy. Indeed disequality constraints are an actively discussed topic [11] in the Curry community. With our application we provide a practically relevant example (complementary to the sometimes quite artificial examples provided in research papers) demonstrating the need for disequality constraints as a language extension.

With our framework we can generate graphs where all nodes are just free variables. At the same time, we can ensure that these variables cannot coincide later by rising proper disequality constraints. Just using the operator `(/=)` is not sufficient since that way the computation would be suspended. Thus, using MCC a correct formulation of a sample body of `stmts` can be realized as follows:

```
stmts (n1,n2) | allDifferent [n1,n2,n]
              = stmt (n1,n) <*>
                stmts (n,n2) <*>
                noDanglEdge [n]
                where n free
```

Here, `allDifferent` generates a set of pairwise disequality constraints that have to be ensured in order to apply this rule. The primitive `noDanglEdge` ensures (also via generating disequality constraints) that the given nodes (the inner ones) are not visited by the edges of the remaining graph. Since that way a lot of disequality constraints (including many duplicates) are generated (in particular if graph nodes are free variables as it is the case for graph generation), it is important that the compiler can deal with them efficiently.

3.2 Performance

As described in [3] we have connected our framework with the diagram editor generator DiaGen [12]. Here, it is used as a back-end for the computation of diagram completions. Diagram editors are a highly interactive application, so that performance is crucially important for the acceptance of new functionality by users. In fact, graph parser combinators are an elegant and straightforward approach to graph parsing. However, due to their top-down nature with backtracking they are not efficient. (Graph) parsers based on dynamic programming techniques usually scale much better [13]. Nevertheless it is worth to see how Curry compilers deal with it.

The Figures 4, 5 and 6 provide some performance data for different kinds of flowgraphs. We have applied the parser given in Fig. 2 to such graphs of different sizes (=number of edges). To avoid the arrangement of edges in the graph representation to affect the performance we have searched for all possible derivations using `findall`.

The measurement has been performed on standard hardware (Intel Core2 Duo, 2GHz, 2GB RAM). The operating system has been Ubuntu Linux 7.10 running in a VMware virtual machine on Windows Vista. We have used MCC 0.9.11, PAKCS 1.9.1 (2) with SICStus 4.0.2[4], and KiCS 0.81893. We provide the programs we have used for this benchmark via the project website. We will also add to this site all suggestions for further, possibly compiler-specific improvements we will receive.

We see that MCC ist most efficient (it is a native code compiler after all). Second, we have PAKCS (with SICStus). The series PAKCS/FP in Fig. 4 will

[4] PAKCS also can be used with SWI-Prolog as a backend. However, applied to our example SICStus outperforms SWI-Prolog by a factor of 5.

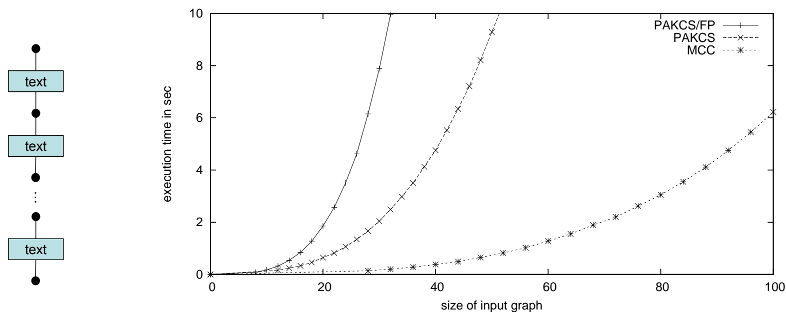

Fig. 4. Parsing of `text`-sequences

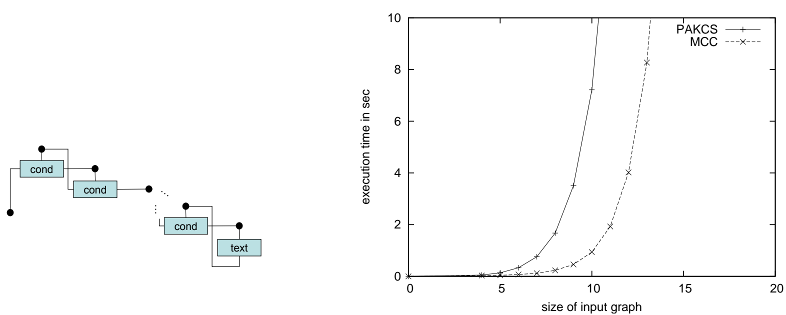

Fig. 5. Parsing of nested `cond`s

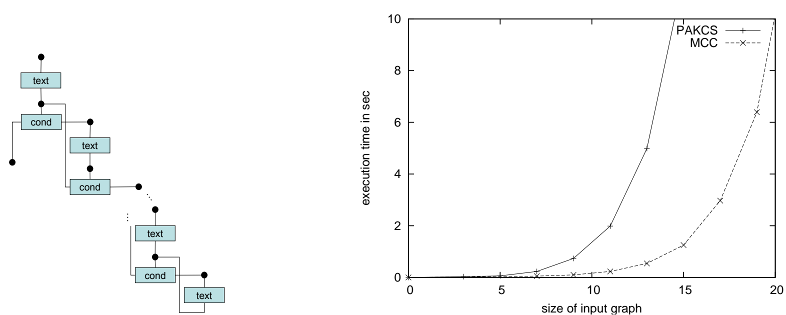

Fig. 6. Parsing of graphs consisting of `cond`s and `text`s

be discussed later. KiCS data is not shown in the figures. In fact, the biggest graph that we have successfully processed with KiCS has been of size 5 only. However, we have to consider that KiCS is still very much work in progress. Furthermore, our application makes heavy use of logic programming features. So it is no surprise that a Prolog backend is superior here. In contrast, KiCS outperforms PAKCS in computations without non-determinism [4].

3.3 Numbers

A flexible number type often is very useful. For instance, we can pass an additional parameter to our parsers that determines the number of edges that might be added while parsing. As a result not only the remaining graph is returned but also the edges that have been pretended:

```
type Grappa = Nat -> Graph -> (Graph,Graph)

edge::Edge -> Grappa
--consume edge
edge e 0 g | g=:=(g1++e:g2) = ([],g1++g2)
             where g1, g2 free
--pretend edge
edge e (_+1) g = ([e], g)

(<*>)::Grappa -> Grappa -> Grappa
(p1 <*> p2) errs g = case p1 errs g of
                     (gc1,g')->case p2 (errs-size gc1) g' of
                               (gc2,g'')->(gc1++gc2,g'')
```

However, both PAKCS and MCC are not so good at dealing with (i.e., guessing) numbers [10]. Therefore, we had to declare our own type `Nat` (Peano numbers). KiCS, in contrast, provides built-in support for narrowing on numbers, even floats [14].

3.4 Type classes

Our original framework [15] written in Haskell has heavily relied on type classes. In particular, monads have been used to handle context and state conveniently. Researchers have already started to investigate the integration of type classes in Curry, see e.g. [8]. Although type classes are very nice to have for a lot of reasons, we do not rely on their existence in our functional-logic framework. Instead we have shared results across parsers via logical variables. Admittedly, this is a quite obvious idea. However, we have found it so incredibly helpful that we actually suggest it as an additional design pattern [7].

Consider the grammar for Sierpinski triangles given in Fig. 7a. The thick lines here represent binary terminal hyperedges. Unfortunately, this grammar does also generate irregular triangles like the one shown in Fig. 7b. In contrast,

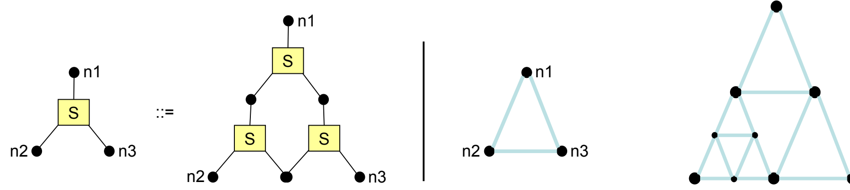

Fig. 7. a) Grammar of Sierpinski triangles and b) irregular triangle

the following parser recognizes and generates regular (i.e., equally deep unfolded) Sierpinski triangles only:

```
s 0 (n1,n2,n3) = edge ("line",[n1,n2]) <*>
                 edge ("line",[n2,n3]) <*>
                 edge ("line",[n3,n1])
s (depth+1) (n1,n2,n3) = s depth (n1,n4,n5) <*>
                         s depth (n4,n2,n6) <*>
                         s depth (n5,n6,n3)
                         where n4,n5,n6 free
```

Note that this parser, altough not suited for efficient parsing, can be directly used to generate huge Sierpinski triangles in a reasonably efficient way. Compared to other graph transformation tools [16] it seem to be even in the center-field (not to mention that a more readable implementation is hard to imagine). For instance, we have generated a Sierpinski triangle of generation 11 with nearly 200.000 edges with MCC in about a minute on standard hardware. Thereby a term with a very large number of free variables has been generated.

3.5 Function Patterns

Function patterns [9] are a language extension only provided by PAKCS. That way the primitive `edge` can be defined more elegantly as:

```
edge::Edge -> Grappa
edge e1 (g1++e2:g2) | e1=:=e2 = g1++g2
```

Note that we indeed have to introduce an additional variable `e2`, since a variable is allowed to occur in the left-hand side of a rule at most once [9]. Unfortunately, in our application this usage of a function pattern has a significantly negative impact on the performance even with function pattern optimization switched on (cf. Fig. 4, series PAKCS/FP). In general, the use of function patterns is recommended to improve the performance though [9].

3.6 Finite Domain Constraints

Finite domain constraints as provided by PAKCS (SICStus Prolog) have appeared very useful in order to carry over our approach from graphs to diagrams. Indeed, there are several diagram languages where the hypergraph model corresponds very closely to the actual concrete syntax. As an example consider Nassi-Shneiderman diagrams (NSD) where nodes of the hypergraph model represent corners of statements. In this case we can just use coordinates as node identifiers (cf. Fig. 8), i.e., we can define the type `Node` as a pair of two `Int`s. The representation of node identifiers does not affect the implementation of parsers at all (we only rely on a notion of equality).

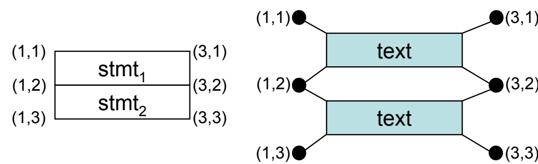

Fig. 8. Coordinates as node identifiers

Consider the diagram given at the left side of Fig. 9. If the parser is used to compute completions, the additional edges it returns can be used as diagram components directly. We even know their coordinates, since they are encoded in the particular node identifiers.

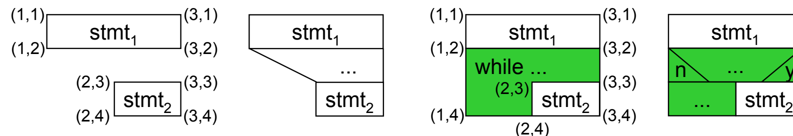

Fig. 9. Incomplete NSD, dubious fix and two meaningful completions

Unfortunately, it's not as simple as that. The second diagram in Fig. 9 would be one of our results – this is not exactly, what we have expected. Indeed, some information is still missing. We need to ensure additional constraints to get a result also meaningful at the diagram level. In particular we must state, that the nodes visited by `text` edges have to form a proper rectangle, i.e., they have to be unifiable with `[(x1,y1),(x2,y1),(x1,y2),(x2,y2)]`. Thereby, it must further hold, that `x1<x2` and `y1<y2`.

If we also introduce corresponding constraints to the other components we get a unique and meaningful completion of size one also shown in Fig. 9. Of course, we also can ask for larger completions. An example result is given at the right side of Fig. 9.

This approach is quite restricted though. In [17] constraint hypergraph grammars have been introduced that are better suited for this purpose. In particular, minimal changes are computed for constrained variables in order to find solutions. We have followed this more flexible approach in [3].

3.7 Search

KiCS provides support for the most fine-grained control and encapsulation of search [4]. Its breadth-first approach seems to be promising especially for the generation of graph languages. We have to study this aspect in greater depth in the future.

4 Conclusion

In this paper we have summarized our experiences with Curry compilers that we gained while implementing and testing our framework of graph parser combinators. Syntax analysis of graphs, in particular the computation of graph completions, is a practically relevant application of functional-logic languages [3]. None of the existing compilers supports all features/language extensions beneficial for graph parsing yet, most importantly:

- high performance,
- support for both finite domain and disequality constraints,
- narrowing on numbers,
- search control

Due to compiler's different strengths and weaknesses users will frequently switch the compiler. This procedure can and should be simplified by defining a common set of libraries as done e.g. in the Haskell report. The existence of different compilers is a very useful thing as long as potential users do not have to reimplement their program for each one anew.

Acknowledgement

We especially thank Bernd Braßel, Wolfgang Lux and Michael Hanus for their ongoing support.

References

1. Mazanek, S., Minas, M.: Functional-logic graph parser combinators. In: Proc. of the 19th Intl. Conference on Rewriting Techniques and Applications. LNCS, Springer (2008)
2. Hanus, M.: Multi-paradigm declarative languages. In: Proc. of the Intl. Conference on Logic Programming (ICLP 2007), Springer (2007) 45–75

3. Mazanek, S., Maier, S., Minas, M.: Auto-completion for diagram editors based on graph grammars. In: Proc. of the 2008 IEEE Symposium on Visual Languages and Human-Centric Computing, IEEE Computer Society Press (2008)
4. Huch, F., Brassel, B.: The Kiel Curry System KiCS. In: Proceedings of the International Conference on Applications of Declarative Programming and Knowledge Management (INAP 2007) and the Workshop on (Constraint) Logic Programming (WLP 2007). (2007)
5. Drewes, F., Habel, A., Kreowski, H.J.: Hyperedge replacement graph grammars. In Rozenberg, G., ed.: Handbook of Graph Grammars and Computing by Graph Transformation. Vol. I: Foundations. World Scientific (1997) 95–162
6. Caballero, R., López-Fraguas, F.J.: A functional-logic perspective on parsing. In: FLOPS '99: Proceedings of the 4th Fuji International Symposium on Functional and Logic Programming, London, UK, Springer-Verlag (1999) 85–99
7. Antoy, S., Hanus, M.: Functional logic design patterns. In: Proc. of the 6th Intl. Symposium on Functional and Logic Programming, Springer (2002) 67–87
8. Lux, W.: Adding Haskell-style overloading to Curry. In Hanus, M., Fischer, S., eds.: 25. Workshop der GI-Fachgruppe "Programmiersprachen und Rechenkonzepte". (2008)
9. Antoy, S., Hanus, M.: Declarative programming with function patterns. In: LOPSTR. (2005) 6–22
10. Brassel, B., Fischer, S., Huch, F.: Declaring numbers. In: Workshop on Functional and (Constraint) Logic Programming. (2007)
11. Arias, E.J.G., Carballo, J.M., Poza, J.M.R.: A proposal for disequality constraints in curry. Electron. Notes Theor. Comput. Sci. **177** (2007) 269–285
12. Minas, M.: Concepts and realization of a diagram editor generator based on hypergraph transformation. Science of Computer Programming **44**(2) (2002) 157–180
13. Mazanek, S., Maier, S., Minas, M.: An algorithm for hypergraph completion according to hyperedge replacement grammars. In: Proc. of the 4th Intl. Conference on Graph Transformation. LNCS, Springer (2008)
14. Christiansen, J.: Ach, wie gut, dass niemand weiß, dass ich dreizehnsiebtel heiß. In Hanus, M., Fischer, S., eds.: 25. Workshop der GI-Fachgruppe "Programmiersprachen und Rechenkonzepte". (2008)
15. Mazanek, S., Minas, M.: Graph parser combinators. In: Proc. of the 19th Intl. Symposium on the Implementation and Application of Functional Languages. LNCS, Springer (2008)
16. Gabriele Taentzer et al. : Generation of sierpinski triangles: A case study for graph transformation tools. In: Proc. of AGTIVE 2007. LNCS, Springer (2008)
17. Minas, M., Viehstaedt, G.: Specification of diagram editors providing layout adjustment with minimal change. Proc. 1993 IEEE Symposium on Visual Languages (1993) 324–329

Adding Haskell-style Overloading to Curry

Wolfgang Lux

University of Münster
wlux@uni-muenster.de

Abstract. The integrated declarative language Curry [Han06] combines features from functional, logic, and constraint programming languages. Its syntax and semantics are very similar to those of the lazy functional language Haskell, but currently it lacks some of Haskell's more advanced features. The most important of these is Haskell's systematic approach to overloading with type classes. Type classes have proven very useful in Haskell over more than a decade now and are one of Haskell's well recognized features, increasing the user's ability to write generic and more concise code. This feature has been missed more than once by users of Curry.
Our own experience with adding type classes to the Münster Curry compiler shows that it is mostly straightforward, since the theory behind and the implementation of type classes are well studied. However, there are some problematic areas, including overloading of numeric literals, possibly overloaded equality constraints, and soundness problems of rank-2 types in functional-logic languages. We present design options for these issues and motivate the design decisions taken for the Münster Curry compiler.

1 Introduction

The integrated declarative language Curry [Han06] combines features from functional, logic, and constraint programming languages. Its syntax and semantics are very similar to those of the lazy functional language Haskell [Pey03]. As a functional-logic language Curry also supports logical variables and non-deterministic functions, which allow representing partial data and search for solutions. While being a mostly conservative extension of Haskell in this respect, Curry lacks some of Haskell's advanced features, in particular its systematic approach to overloading with type classes.

Type classes are a tool for supporting overloading (ad-hoc polymorphism) in a Hindley-Milner type system with automatic type inference. This is achieved by declaring overloaded functions as members of type classes in class declarations and providing implementations of these functions for particular types in instance declarations. Fig. 1 shows (simplified) `Eq` and `Num` classes providing an overloaded equality test operator and overloaded arithmetic operations, respectively, together with sample instance declarations. In order to accommodate overloaded functions, types are extended with contexts which may restrict the types of arguments to instances of particular classes. For instance, the function

```
class Eq a where
  (==) :: a -> a -> Bool
instance Eq Int where
  (==) = primEqInt
instance Eq Float where
  (==) = primEqFloat
```

```
class Num a where
  fromInteger :: Integer -> a
  (+), (-), (*) :: a -> a -> a
instance Num Int where
  fromInterger =
    primIntFromInteger
  (+) = primAddInt
  (-) = primSubInt
  (*) = primMulInt
instance Num Float where
  ...
```

Fig. 1. Overloaded equality and arithmetic operations

```
lookup a [] = Nothing
lookup a ((x,y):xys) =
  if a == x then Just y else lookup a xys
```

has type `lookup :: Eq a => a -> [(a,b)] -> Maybe b`. The context `Eq a` indicates that the type of the first argument must be an instance of the `Eq` class, which is checked statically during type inference, thus ensuring that an implementation of the equality test `(==)` is available when `lookup` is evaluated.

The common approach for implementing type classes is based on a source-to-source transformation in the compiler [Aug93,PJ93]. This transformation introduces implicit dictionary arguments for all type class constraints occurring in the type of a function. Each type class declaration gives rise to a new dictionary data type declaration and a global function declaration for each of its methods extracting the respective method implementation from the dictionary. Instance declarations are transformed into functions which return the dictionary for their class and type. Fig. 2 gives a simple example for the transformation of the `Eq` class and the `lookup` function.

This dictionary transformation can be used to add support for type classes to Curry as well. Due to the presence of logical variables and the subtle differences between Haskell's and Curry's operational semantics, one faces a few semantic issues though. The most important of these issues are considered in this paper. In particular, the following sections address flexible vs. rigid evaluation in Curry (Sect. 2), possibly overloaded equality constraints (Sect. 3), overloaded numeric literals (Sect. 4), and rank-2 types (Sect. 5). The final sections of this paper present related work and conclude. We assume familiarity with the languages Haskell and Curry throughout this paper.

2 Flexible vs. Rigid Methods

Curry's operational semantics uses an optimal evaluation strategy that combines needed narrowing [AEH94] and residuation [ALN87]. User defined functions employ narrowing, i.e., free variables in demanded argument positions are

```
class Eq a where
  (==) :: a -> a -> Bool

instance Eq Int where
  (==) = primEqInt

lookup :: Eq a =>
  a -> [(a,b)] -> Maybe b
lookup a []   = Nothing
lookup a ((x,y):xys)
  | a == x    = Just y
  | otherwise = lookup a xys
```

```
data Dict_Eq a = Dict_Eq {
  (==) :: a -> a -> Bool }
(==) (Dict_Eq eq) = eq

dict_Eq,Int =
  Dict_Eq { (==) = primEqInt }

lookup :: Dict_Eq a ->
  a -> [(a,b)] -> Maybe b
lookup d_Eq a []  = Nothing
lookup d_Eq a ((x,y):xys)
  | (==) d_Eq a x = Just y
  | otherwise = lookup d_Eq a xys
```

Fig. 2. Sample dictionary transformation

instantiated non-deterministically during pattern matching. Such functions are called *flexible*. On the other hand, most primitive functions use residuation, i.e., free variables in demanded argument positions cause the current thread to be suspended until the variable is sufficiently instantiated by some other concurrent computation. Such functions are called *rigid*.

Promoting currently built-in rigid polymorphic primitives like `(==)` and `show` into type class methods has some important semantic implications, since code can no longer assume that these functions are rigid. This is of particular interest for the `(==)` equality operator, which is assumed to be a rigid function in many places including the Prelude's definition of the operator `($##)`, which is supposed to apply a (unary) function to an argument after evaluating the argument to a ground normal form: `f $## x | x==x = f x`.

In order to provide a rigid version of the `(==)` type class method we propose to introduce a new primitive `ensureGround :: a -> a`, which is a natural generalization of the `ensureFree` primitive and lazily evaluates its argument to a ground term. With this primitive, the operator `($##)` could be implemented as well as a rigid variant of the equality test.

```
f $## x = f $!! ensureGround x
x === y = ensureGround x == ensureGround y
```

3 Equality Constraints

Corresponding to the distinction between rigid and flexible functions, Curry provides two equality operations, the aforementioned boolean test for equality `(==) :: a -> a -> Bool` and the equality constraint operator `(=:=) :: a -> a -> Success`. The essential difference between the expressions e_1 `==` e_2 and e_1 `=:=` e_2 is that the former yields a result (either `True` or `False`) only for ground terms whereas the latter is satisfied if e_1 and e_2 can be reduced to equal

terms possibly instantiating unbound variables in one argument to a term at the same position in the other argument.

If type classes are available, it seems natural to make `(=:=)` an instance method of a new type class `Equal`:

```
class Equal a where
  (=:=) :: a -> a -> Success
instance Equal a => Equal [a] where
  []     =:= []     = success
  (x:xs) =:= (y:ys) = x=:=y & xs=:=ys
```

This approach nicely prevents a type soundness issue which is present for an overloaded polymorphic equality constraint that can be applied to expressions with a functional or existentially quantified type. While neither of this is part of the Curry standard, such constraints are supported by some implementations. The problem is that with this extension it is possible to define an unsound polymorphic type cast function[1], e.g.,

```
data T = forall a. T a      -- Existential quantification!
cast :: a -> b
cast x | T x =:= T y = y where y free
```

This definition would be prevented if `(=:=)` were a type class method because it is impossible to define a (non-trivial) `Equal` instance for type `T`. The definition

```
instance Equal T where
  T x =:= T y = x=:=y      -- Type error
```

is rejected because the pattern variables `x` and `y` have incompatible and non-unifyable types due to the existentially quantified type variable in `T`'s declaration.

However, the naive definition of `(=:=)` for the list type shown above has the major drawback that it can instantiate free variables unnecessarily and furthermore can cause an unintended non-deterministic evaluation of programs. For instance, consider the program

```
main = doSolve (x=:="IO") >> print x where x free
```

Given the above definition of `(=:=)`, this program would fail with a runtime error due to the non-deterministic evaluation of the constraint `x=:="IO"`, which is not encapsulated in an `IO` context.

In order to prevent unnecessary instantiation of logical variables and unintended non-deterministic choices, one could envision a more complicated instance definition, e.g.,

```
instance Equal a => Equal [a] where
  xs =:= ys =
```

[1] Instead of the existentially quantified data constructor `T` one could use an isomorphic partial application of the function `ignore x y = y`.

```
    if isVar xs then primInstVar xs ys
    else if isVar ys then primInstVar ys xs
    else case (xs,ys) of
      ([],[])      -> success
      (x:xs,y:ys) -> x=:=y & xs=:=ys
```

which uses two new primitives `isVar :: a -> Bool` and `primInstVar :: a -> a -> Success` that check whether the argument is a free variable and instantiate the free variable with another expression, respectively.[2] Since these are unsafe primitives and the user is not expected to write such boilerplate code all over the place, one better hides the use of these primitives in an overloaded Prelude function and provides only a method for comparing non-variable terms in class `Equal`.

```
class Equal a where
  equal :: a -> a -> Success

(=:=) :: Equal a => a -> a -> Success
x =:= y =
  if isVar x then primInstVar x y
  else if isVar y then primInstVar y x
  else equal x y

instance Equal a => Equal [a] where
  equal []     []     = success
  equal (x:xs) (y:ys) = x=:=y & xs=:=ys
```

4 Overloaded Numeric Literals

Overloading of arithmetic operations allows the user to define generic functions which can be used for any numeric type. For instance, the functions `sum` and `prod` in

```
sum, prod :: Num a => [a] -> a
sum  xs = foldr (+) 0 xs
prod xs = foldr (*) 1 xs
```

compute the sum and product, respectively, of a list of numbers of an arbitrary numeric type, i.e., a type which has a `Num` instance. Without type classes separate definitions with unique names would have to be defined for each numeric type, which soon becomes inconvenient if many numeric types exist. Recall that the Haskell'98 standard already provides fixed and arbitrary precision integer types, single and double precision floating-point types, rational numbers, and complex numbers.[3]

[2] The `primInstVar` primitive also has to perform the occurs check required by Curry's semantics and ensure that the second argument has a normal form.

[3] Actually, the rational and complex number types are overloaded themselves and can be used at any integral and floating-point type, respectively.

For the programmer's convenience, numeric literals in Haskell have overloaded types themselves. For instance, the constants `0` and `1` in the definition of `sum` and `prod` have type `Num a => a`. This is achieved by treating an (integral) numeric literal i in source code as an abbreviation for the expression `fromInteger` i (cf. the definition of the `Num` class in Fig. 1).

Overloading of numeric literals in Haskell carries over to patterns so that, e.g.,

```
zero 0 = True
zero _ = False
```

has type `zero :: Num a => a -> Bool`. This is made possible by Haskell's pattern matching semantics, which transforms case expressions with literal patterns into equivalent if-then-else expressions. For instance, the definition of `zero` is transformed into

```
zero x = if x == fromInteger 0 then True else False
```

This transformation could be used in Curry as well but it has the unfortunate effect of possibly making definitions more rigid than expected. For instance, the goal `zero x where x free; x::Int` would suspend since the equality instance for fixed precision integer numbers is based on a rigid primitive.

In order to avoid unintended suspensions, one might consider a different transformation which replaces numeric literals in patterns by equality constraints in guards.[4] Thus,

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

would be transformed

```
coin x | x=:=fromInteger 0 = success
coin x | x=:=fromInteger 1 = success
```

Yet, this transformation is problematic as well. Whereas the transformation using if-then-else expressions and `(==)` may unexpectedly suspend when applied to a free variable, the transformation using equality constraints may cause unintended non-deterministic search when applied to a concrete number. For instance, a runtime error would occur when executing the program

```
main = doSolve (x =:=0 &> coin x) >> print x where x free
```

It is also worth pointing out that this transformation requires either that `(=:=)` is a fully polymorphic primitive with type `a -> a -> Success`, or that the `Equal` class is made an additional superclass of the `Num` class besides `Eq` and `Show`.

Since neither of these transformations is perfect, we suggest to perform neither of them in the compiler. Thus, no overloading will be available for numeric literals in patterns. Note that this also means that the following program is rejected with a type error

[4] Function patterns [AH05] would yield an equivalent transformation.

```
data Nat = Z | S Nat deriving (Eq,Show)
instance Num Nat where ...
even 0         = success
even (S (S m)) = even m
```

If overloading is desired, the programmer has to make an explicit choice by rewriting the function in one or the other way herself. Also note that numeric literals in patterns are less useful in Curry than in Haskell due to the fact that all defining equations of a function are considered independently in Curry and therefore, given the definition of `zero` above, the goal `zero 0` has *two* solutions, namely `True` and `False`.

It might be useful though to use if-then-else cascades for transforming `case` expressions in Curry, since, similar to Haskell, only the first matching alternative is evaluated. Since case expressions evaluate their argument rigidly, the transformation must use the proposed top-level function `(===)` when comparing literals, i.e.,

```
zero x = case x of { 0 -> True; _ -> False }
```

would be transformed into

```
zero x = if x === fromInteger 0 then True else False
```

5 Rank-2 Types

It is well known that the dictionary transformation of Haskell source code requires rank-2 types in order to handle polymorphic methods correctly, These appear naturally with higher-order classes like `Functor` and `Monad`. For instance, the `Monad` class is defined as follows[5]

```
class Monad m where
  return :: a -> m a
  (>>=)  :: m a -> (a -> m b) -> m b
```

The corresponding dictionary

```
data Dict_Monad m = Dict_Monad {
  return :: forall a. a -> m a
  (>>=)  :: forall a b. m a -> (a -> m b) -> m b }
```

must use local universal quantifiers in the argument types of the $\texttt{Dict}_{Monad}$ constructor so that the monadic methods can be used at different types (though for the same monad) in a single function. For instance, consider the function

```
sequence (m:ms) =
  m >>= \x -> sequence ms >>= \xs -> return (x:xs)
```

[5] For the sake of the presentation we leave out the `(>>)` and `fail` methods.

where (>>=) is used at two different types, namely `Monad m => m a -> (a -> m [b]) -> m [b]` for the first and `Monad m => m [a] -> ([a] -> m [b]) -> m [b]` for the second occurrence.

Given that rank-2 types must be supported by an implementation anyway, it seems natural to make them available to the user. In fact, rank-2 types can be useful on their own. For instance, safe encapsulation of mutable state [LP95] can be implemented with operations of an abstract type `ST s a`, which describes state transformers with state type `s` and result type `a`.

```
data ST s a
instance Monad (ST s) where ...
runST :: (forall s. ST s a) -> a
```

The local universal quantification of the state parameter type `s` in `runST`'s type ensures that it cannot appear in the result type `a` of the state transformer and thus the state itself cannot escape. Note that the `IO` type is just a particular instance of the generic state transformer type: `type IO a = ST RealWorld a`.

Unfortunately, rank-2 types raise another type soundness issue for logic and functional-logic languages. Consider the type `data T = T (forall a. a)` and the function definition

```
f (T x) = x=:=0 & x=:="Hello"
```

This definition is accepted by the type checker, since `x` has type $\forall\alpha.\alpha$. In Haskell this definition is unproblematic because the only expressions that are sufficiently polymorphic to suit as argument of the `T` constructor are `undefined` and pattern variables matching a universally quantified argument of a data constructor. In Curry all expressions apart from such pattern variables have monomorphic types. On the other hand, the expression `let x free in f x` is accepted by the type checker and would cause `x` to be instantiated to a fresh term of the form `T y`, where `y` is another free variable.

In order to prevent such unsound instantiations of constructors with universally quantified type arguments, we propose to introduce a type class `Narrowable`

```
class Narrowable a where
  narrow :: a

instance Narrowable a => Narrowable [a] where
  narrow = []
  narrow = (x:xs) where x,xs free
```

and impose the additional restriction that the types of free variables must be instances of the `Narrowable` class. Thus, `let x free in x` will have type `Narrowable a => a`. The important point is that it is impossible to define a (non-trivial) instance for `T` due to Curry's monomorphism restriction.

In order to pass around `Narrowable` dictionaries at runtime without introducing `Narrowable` constraints all over the place they should be associated with the free variables themselves similar to constraints for existentially quantified type variables, e.g.

```
data LVar = forall a. Narrowable a => LVar a      -- Exist. Quant.
```

A downside of this approach is that additional type signatures may be required in order to avoid ambiguous type errors for the free variables of a definition and, in particular, a goal entered at the prompt of a Curry interpreter. Another problem is that in order for this approach to become effective the `narrow` method must be called when a free variable is detected during a flexible pattern matching. Due to the fact that the definition of `narrow` is almost always non-deterministic, this may cause unintended and unnecessary non-deterministic search in programs. Furthermore, solutions might be explored in the fixed order of the narrow method's declaration rather than the order of equations of the function where the free variable is matched.[6]

6 Related Work

Type classes initially were conceived only to provide overloading for types with kind $*$. This was extended by Mark P. Jones' work on qualified types to provide overloading for types with arbitrary kinds and thus paved the way for the ubiquitous `Functor` and `Monad` classes in Haskell. Type classes in Haskell'98 are restricted to only a single argument and furthermore types in instance declarations are restricted to the form $T\,x_1, \ldots, x_k$ where T is a type constructor with arity $n \geq k$ and $x_1, \ldots, x_k$ are type variables. These restrictions ensure that context reduction during type inference always terminates and instances are unique. Current Haskell implementations lift these restrictions and support multi-parameter type classes and ambiguous instances; [PJM97] explores the design space. Recent proposals for multi-parameter type classes include functional dependencies [Jon00] and associated types [CKP$^+$05].

An approach to combine type inference and overloading without type classes has been presented in [CF99].

An alternative to overloading are ML-style higher order modules, where the programmer would define functions using overloaded operations inside a higher order module that is parameterized with the signatures of modules providing these operations. The relation between type classes and higher order modules has been studied, among others, in [DHC07].

Type classes in Curry were first proposed in [MMdP$^+$96]. The first Curry implementation with type class support to our knowledge was the Zinc compiler[7]. Unfortunately, this experimental extension of the Münster Curry compiler does not address the issues described in this paper. In particular, it does not support polymorphic methods correctly and also does not support overloading of numeric literals.

[6] The order in which non-deterministic alternatives are evaluated is not specified by the Curry report. However, in order to explore the search space of a goal efficiently, some means to control the order of evaluation must be provided to the programmer and declaration order appears to be a natural choice for this.

[7] `http://sourceforge.net/projects/zinc-project`

7 Conclusion and Future Work

Type classes are a very useful extension to Curry that should be adopted. A prototypical implementation is available in the type classes branch[8] of the Münster Curry compiler. Support for most of the features discussed in the paper is present. In particular, it correctly supports polymorphic methods, which is the main advancement compared to the Zinc compiler.

Future work will concern adding support for multi-parameter type classes to Curry. In contrast to the problems presented in this paper, multi-parametric type classes are expected to introduce no new problems.

References

[AEH94] Sergio Antoy, Rachid Echahed, and Michael Hanus. A needed narrowing strategy. In *Proc. POPL'94*, pages 268–279. ACM Press, 1994.

[AH05] Sergio Antoy and Michael Hanus. Declarative programming with function patterns. In Patricia M. Hill, editor, *Proc. LOPSTR 2005*, LNCS 3901, pages 6–22. Springer, 2005.

[ALN87] Hassan Aït-Kaci, P. Lincoln, and R. Nasr. Le Fun: Logic, equations, and functions. In *Proc. 4th ILPS*, pages 17–23, 1987.

[Aug93] Lennart Augustsson. Implementing Haskell overloading. In *Proc. FPCA '93*, pages 65–73. ACM Press, 1993.

[CF99] Carlos Camarão and Lucília Figueiredo. Type inference for overloading without restrictions, declarations or annotations. In Aart Middeldorp and Taisuke Sato, editors, *Proc. FLOPS '99*, LNCS 1772, pages 37–52. Springer, 1999.

[CKP+05] Manuel Chakravarty, Gabrielle Keller, Simon L. Peyton Jones, , and Simon Marlow. Associated types with class. In Jens Palsberg and Martín Abadi, editors, *Proc. POPL'05*, pages 1–13. ACM, 2005.

[DHC07] Derek Dreyer, Robert Harper, and Manuel M.T. Chakravarty. Modular type classes. In *Proc. POPL '07*. ACM, 2007.

[Han06] Michael Hanus (ed.). Curry: An integrated functional logic language. (version 0.8.2). `http://www.informatik.uni-kiel.de/~mh/curry/report.html`, 2006.

[Jon00] Mark P. Jones. Type classes with functional dependencies. In Gert Smolka, editor, *Proc. ESOP 2000*, LNCS 1782, pages 230–244. Springer, 2000.

[LP95] John Launchbury and Simon L. Peyton Jones. State in haskell. *Lisp and Symbolic Computation*, 8(4):293–341, 1995.

[MMdP+96] Juan José Moreno-Navarro, Julio Mariño, Andrés del Pozo, Angel Herranz-Nieva, and Julio García-Martín. Adding type classes to functional logic languages. In *Proc. APPIA-GULP-ProDe'96*. Faculty of Informatics, San Sebastian, Spain, 1996.

[Pey03] Simon L. Peyton Jones, editor. *Haskell 98 Language and Libraries The Revised Report.* Cambridge University Press, 2003.

[PJ93] John Peterson and Mark P. Jones. Implementing type classes. In *Proc. PLDI'93*, SIGPLAN Notices 28(6), pages 227–236. ACM, 1993.

[PJM97] Simon Peyton Jones, Mark P. Jones, and Erik Meijer. Type classes: exploring the design space. In *Proc. of the Haskell Workshop*, 1997.

[8] `darcs get http://danae.uni-muenster.de/~lux/curry/darcs/type-classes`

Ach, wie gut, dass niemand weiß, dass ich dreizehnsiebtel heiß

Jan Christiansen

Institut für Informatik
Christian-Albrechts-Universität Kiel
jac@informatik.uni-kiel.de

In Programmiersprachen stellen Zahlen häufig einen ausgezeichneten Datentypen dar. In der Programmiersprache Curry äußert sich dies darin, dass Zahlen nicht geraten werden können. Das heißt es existieren keine freien Variablen der Datentypen `Int` und `Float`. In [1] haben Braßel, Fischer und Huch gezeigt wie man diese Sonderbehandlung im Falle der Ganzen Zahlen umgehen kann, indem man Ganze Zahlen durch einen algebraischen Datentypen darstellt.

Rationale Zahlen können als Kettenbruch dargestellt werden. Ein regulärer Kettenbruch hat zum Beispiel die folgende Form:

$$a_1 + \frac{1}{a_2 + \frac{1}{a_3 \ldots}}$$

Wir präsentieren eine eindeutige Darstellung Rationaler Zahlen mit Hilfe regulärer Kettenbrüche. Diese Darstellung kann genutzt werden um Rationale Zahlen als algebraischen Datentypen in Curry zu implementieren.

In einem Technischen Bericht des Massachusetts Institute of Technology [2] aus dem Jahre 1972 wird ein Algorithmus zur Addition, Subtraktion, Multiplikation und Division zweier regulärer Kettenbrüche präsentiert. Dieser Algorithmus soll genutzt werden um die entsprechenden Operationen für diesen Datentypen bereitzustellen.

Literatur

1. Brassel, B., Fischer, S., Huch, F.: Declaring numbers. In: Proc. of the 16th International Workshop on Functional and (Constraint) Logic Programming WFLP 2007. (2007)
2. Beeler, M., Gosper, R.W., Schroeppel, R.: Hakmem. Technical report, Cambridge, MA, USA (1972)

A Debugger for Functional Logic Languages*

Bernd Braßel

Institute of Computer Science, University of Kiel
Olshausenstr. 40, 24098 Kiel, Germany
Email: bbr@informatik.uni-kiel.de

Abstract. This paper is based on a recently developed technique to build debugging tools for lazy functional programming languages. With this technique it is possible to replay the execution of a lazy program with a strict semantics by recording information of unevaluated expressions. The recorded information is called an *oracle* and is very compact. Oracles contain the number of strict steps between discarding unevaluated expressions. The technique has already been successfully employed to construct a debugger for lazy functional languages.
This paper extends the technique to include also lazy functional *logic* languages. A debugging tool built with the technique can be downloaded at www-ps.informatik.uni-kiel.de/~bbr.

1 Introduction

It has often been remarked that the advanced features of modern functional (logic) languages pose an obstacle when trying to find errors, cf. for instance [14, 17]. Therefore in recent years, sophisticated techniques have been developed to support the programmer with useful tools to find bugs in his programs. The most influential technique is often called "Algorithmic Debugging" (and sometimes "Declarative Debugging") and was originally developed in the context of logic programming [16]. It has also been adopted to functional programming, e.g. in [15], and also to functional logic programming, see [9] for the most recent work. It has, however, also been argued that declarative debugging is not always the tool of choice, and that other tools provide complementary views, cf. [10]. Consequently, the most flexible tool for functional languages, HAT [18], provides several views among which the user can switch arbitrarily. The drawback of such flexibility is paid with a severe overhead in the usage of resources. Every HAT session records megabytes of data about the execution of a given program, often in the hundreds. Much of the work invested into HAT was concerned with making this huge amount of data manageable with acceptable response times. The resulting system was highly optimized for the case of functional programming and is, in consequence, not easy to port to broader settings like functional logic languages. An according attempt developed in [6, 5, 3] did not lead to the implementation of tools with satisfying performance.

* This work has been partially supported by the German Research Council (DFG) under grant Ha 2457/5-2.

Instead we have developed a different technique in [4] which enabled us to built a fast and stable debugging system within a short time. This technique has first been developed to build a debugger for lazy functional languages [8]. This work describes the extension to the broader setting of functional logic languages.

Before presenting the extension in Section 2, we will first describe the basic idea in Section 1.1. Section 3 contains the description of a debugging tool for the functional logic language Curry based on the technique. Section 4 concludes.

1.1 Leftmost Innermost Evaluation with Oracle

A simple observation was made during the development of [3]: all of the more sophisticated approaches to support debugging in lazy programming languages try to present information about the program's execution *as if* the semantics was eager. In other words, the job of the debugging tools was to wind back the aspects of a complex evaluation strategy and map it to a simple one. Now the reasoning was like this: If this mapping of lazy to eager evaluation is the core of successful debugging techniques, all of these approaches could be derived from mapping a given lazy derivation to an eager one. The information for such a mapping, however, is smaller by magnitudes than that needed by tools like HAT or by that developed in [6]. It can be compressed to counting left-most innermost steps between "discarding steps", i.e., such steps which discard expressions not needed for the whole evaluation. For example, evaluating the expression `(head (tail (from 0)))` in the context of the following program

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

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

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

head (tail (from 0)) => head (tail (0:from (0+1)))
                     => head (from (0+1))
                     => head (0+1:from ((0+1)+1) => 0+1 => 1
```

can be described as: "Do three steps innermost then discard the next two leftmost innermost expressions and do two more eager steps." In short the information can be comprised to the list of eager steps `[3,0,2]`. The first number is decreased and a leading zero means a discard step. The example derivation can then be mapped to the eager evaluation:

```
                        [3,0,2]
head (tail (from 0)) =>[2,0,2] head (tail (0:from (0+1)))
                     =>[1,0,2] head (tail (0:from 1))
                     =>[0,0,2] head (tail (0:1:from (1+1)))
```

```
=>[0,2]   head (tail (0:1:from _))
=>[2]     head (tail (0:1:_))
=>[1]     head (1:_)
=>[0]     1
```

In [4] we have formalized a technique to automatically record and replay such step information. Apart from showing the soundness of the approach we were able to prove interesting properties about the magnitude of resources needed to compute the oracle information. In [8] we have then proposed a tool for debugging lazy functional programs with the oracle approach. This paper is concerned with the extension of the approach to functional logic languages.

2 The Extension to Functional Logic Programming

The main topic of this paper is how to transfer the oracle technique described above to the more general setting of functional logic programming. We can identify three main topics of the extension:

1. operations defined by non-deterministic branching
2. free variables in conjunction with narrowing
3. free variables in conjunction with unification

From these three topics we will discuss only the two first ones, unification is left for future work.

Before we discuss the details of our solution we first give two well known examples for the two topics. First we will introduce the non-deterministic operation `(?)` on base of which all following non-deterministic operations will be defined. `(?)` takes two arguments and non-deterministically returns one of them.

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

Using `(?)`, we can define an operation which inserts a given argument anywhere into a given list.

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

Note that `(?)` has a very low precedence (the lowest possible in Curry actually). Based on `insert` there is an expressive way to define permutations and permutation sort, cf. [12].

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

```
permSort :: [Int] -> [Int]
permSort l | sorted permutation = permutation
  where permutation = permute l

sorted :: [Int] -> Bool
sorted []  = True
sorted [x] = True
sorted (x:y:ys) | x<=y = sorted (y:ys)
```

The above declaration of `permSort` indeed defines a sorting operation, e.g., the call `(permSort [2,1,3])` evaluates to `[1,2,3]` and nothing else.

As an example for the use of free variables together with narrowing, we will use the standard definition of a function on lists `head` and a Boolean function `guard`. Both functions are only partially defined such that there is a "narrowing" effect when applying them.

```
guard :: Bool -> a -> a
guard True x = x
```

The running example for narrowing in the following will be `(let x free in guard (head x) 1)` which evaluates to `1` with `x` bound to `(True:y)` for a fresh variable `y`.

After introducing the two running examples we now turn to examine two general concepts in functional logic programming, namely *generators* and computations on *search trees*. Applying these concepts approaches will then lead to the extension of the oracle technique.

2.1 Generators

There are two recent developments in the theory of functional logic languages that simplify the task to extend the oracle technique to this setting considerably. One is the observation that free variables coincide with so called "generator functions" as discovered independently in [2] and [11]. The second is that deterministic and non-deterministic aspects of a functional logic program can be neatly divided into a deterministic evaluation on one hand and a projection of the result according to a set of choices on the other hand. This second idea was described in detail in [7]. Before applying both ideas to the case at hand we illustrate them with regard to the running examples.

A generator for a given type is a function that non-deterministically evaluates to all possible values of that type. For example the generator for Boolean values is straightforward:

```
genBool :: Bool
genBool = True ? False
```

With regard to types with an infinite set of values we have to be more specific about the structure of a generator. Each alternative on the right-hand side should

introduce exactly one of the types' constructors. The constructors' arguments can then be again calls to other generator functions of an appropriate type. According to [2, Definition 3] and in compliance with [11, Definition 3.3] the generator for lists of Boolean values is:

```
genBoolList :: [Bool]
genBoolList = [] ? genBool : genBoolList
```

The resulting function has several interesting properties. Apart from the fact that eventually each possible value is generated (compare to [2, Lemma 1] and [11, Lemma 3.4]), the so defined generator is *productive* in the sense that each alternative produces a head normal form of arbitrary depth in a finite number of steps while introducing only a *finite* number of non-deterministic branches.

In order to be more true to Curry's type system, which allows polymorphic functions and constructors, we change the above definitions slightly. It is easy to see that the claims from [2, 11] still hold.

```
type Generator a :: () -> a
genBool :: Generator Bool
genBool _ = True ? False
genList :: Generator a -> Generator [a]
genList gen _ = [] ? gen () : genList gen ()
```

The extension with the artificial argument `()` is necessary because of call-time choice. Without it, `genList` would produce lists that either contain only `True` or only `False` but not, e.g., `[True,False]`.

The connection between generators and free variables can be illustrated with the narrowing example. Evaluating the expression replacing the variable with a generator, i.e., `(guard (head (genList genBool ())) 1)` also yields `1` with the expression being evaluated only as far as needed, that is to the expression `(True:genList genBool ())`. Not only with regards to semantics there is a close correspondence between generators and free variables, e.g., the result is `1`. But also operationally the correspondence is tight as could be shown in [7]. Wherever we have a free variable in the substitution part of a narrowing derivation, we find a non-evaluated generator in the expression for the derivation with generators.

2.2 Search Trees

The second insight important for the presented work was introduced in [7]. There the operational machinery of functional logic programming is separated into a deterministic part computing on so called search trees and a logic part projecting values out of that tree. The general idea is that all types τ are extended by two new constructors `Fail ::` τ which presents a failure and `Or :: Ref ->` τ `->` τ `->` τ which represents a non-deterministic branching. The reference of type `Ref` is used by the projection to identify identical choices with respect to call-time choice as explained below. The new `Or` constructors are all introduced by

the `(?)` operation, which now – in contrast to the first version above – looks like this:

```
(?) :: a -> a -> a
x ? y = Or r x y where r fresh
```

The generation of fresh references utilized for `(?)` is the only non-deterministic feature needed in order to define the complete operational behavior of functional logic languages as detailed in [7]. In the following we assume that references in `Or` constructors are simple integers although any type with a well-defined identity would suffice. In order to gain the full expressiveness of functional logic programming each pattern matching in the program is extended by a case for the `Or` constructor. For example, the declaration of `insert` is completed by the following rules:

```
insert x (Or r y z) = Or r (insert x y) (insert x z)
insert _ Fail = Fail
```

The definition of `permute` and the other operations introduced above are completed likewise. The only definition for which completion is a bit more complicated is `sorted`, as described below. To get an idea how the completed operations behave consider the following evaluation of `permute [3,1,2]`:

```
permute [3,1,2] => insert 3 (permute [1,2])
                => insert 3 (insert 1 (insert 2 []))
                => insert 3 (insert 1 [2])
                => insert 3 (Or 1 [1,2] [2,1])
{-new rule!-}   => Or 1 (insert 3 [1,2]) (insert 3 [2,1])
                => Or 1 (Or 2 [3,1,2] (1:insert 3 [2]))
                        (Or 3 [3,2,1] (2:insert 3 [1]))
                => Or 1 (Or 2 [3,1,2] (1:Or 4 [3,2] [2,3]))
                        (Or 3 [3,2,1] (2:Or 5 [3,1] [1,3]))
```

To project the search tree to a value we need a set of choices. Such a set defines for which reference which alternative to take. For example the one choice could be `(1,1)` representing that for reference `1` we take the first alternative whereas `(2,2)` would represent the choice to take the second alternative for reference 2. Projecting the resulting tree of the above example by the set of choices `[(1,1),(2,2),(4,2)]` we would obtain the list `[1,2,3]`. A search now boils down to the systematic construction of sets of choices. In order to illustrate this last point we also complete the definition of operation `sorted` defined above. As `sorted` matches on more than one constructor we have to introduce an auxiliary function. In general, in the resulting program each function matches at most one constructor. This makes it possible to continue the computation in the arguments of an `Or` node. The completed declaration of `sorted` is accordingly:

```
sorted :: [Int] -> Bool
sorted []  = True
```

```
sorted (x:xs) = sorted2 x xs
sorted (Or r x y) = Or r (sorted x) (sorted y)
sorted Fail = Fail

sorted2 x [] = True
sorted2 x (y:ys) | x<=y = sorted (y:ys)
sorted2 x (Or r y z) = Or r (sorted2 x y) (sorted2 x z)
sorted2 _ Fail = Fail
```

The guard "`| x<=y = sorted (y:ys)`" can be considered as syntactic sugar for `(guard (x<=y) (sorted (y:ys)))` giving us the opportunity to illustrate the last part of declaration completion, which is that constructors missing in the matching of the original definition will be mapped to `Fail`:

```
guard :: Bool -> a -> a
guard True       e = e
guard False      _ = Fail
guard (Or r x y) e = Or r (guard x e) (guard y e)
guard Fail       _ = Fail
```

With this it is easy to verify that `sorted (permute [3,1,2])` evaluates to:

```
sorted (permute [3,1,2]) = Or 1 (Or 2 Fail (Or 4 Fail True))
                                (Or 3 Fail (Or 5 Fail Fail))
```

The importance of the references in the `Or` constructors can now be seen when considering the evaluation of calls to `permSort`. Considering its definition the operation `permSort` inserts its argument in those places in the tree where there is a `True`. Therefore a total evaluation of the expression `(permSort [3,1,2])` is:

```
permSort [3,1,2] =
  Or 1 (Or 2 Fail (Or 4 Fail (Or 1 (Or 2 [3,1,2] (1:Or 4 [3,2]
                                                          [2,3]))
                                   (Or 3 [3,2,1] (2:Or 5 [3,1]
                                                          [1,3])))))
       (Or 3 Fail (Or 5 Fail Fail))
```

The important point is that for any projection from that tree to a value this result is equal to

```
permSort [3,1,2] = Or 1 (Or 2 Fail (Or 4 Fail [1,2,3])) Fail
```

The reason is that any path to the sub tree corresponding to `(permute [3,1,2])` leads over the choices `[(1,1),(2,2),(4,2)]`. Therefore these choices are also applied for this sub tree leading to the one possibility `[1,2,3]` only. In addition any choice including `(1,2)` can only yield a failure.

Just as a remark, irrelevant alternatives like those in the sub tree for `(permute [3,1,2])` are often discarded before they are evaluated at all because of laziness.

In those cases where they are evaluated they were needed by a former part of the evaluation as in the above example. Nevertheless, it is a good idea to cut away such irrelevant alternatives as soon as possible to free the memory and prevent unnecessary lookup of choices. However, possible optimization techniques are beyond the scope of this paper and we concentrate on the main advantage of the described technique for building debugging tools. This advantage is that because the main computation is now clearly separated in a deterministic derivation and a projection, the oracle technique developed to replay functional programs can be extended for functional logic programs in a straightforward way.

2.3 A Functional Logic Oracle

There are two main ideas we can directly use to extend the oracle technique to functional logic languages:

- non-deterministic choices can be treated like constructors
- free variables can be replaced by non-deterministic operations

These two ideas can be put together to translate functional logic derivations to strict derivations with oracle. For example, consider the lazy evaluation of the expression `head (insert 3 [1,2])`:

```
head (insert 3 [1,2]) => head (Or 1 [3,1,2] (1:insert 3 [2]))
                      => Or 1 (head [3,1,2]) (head (1:insert 3 [2]))
                      => Or 1 3 (head (1:insert 3 [2]))
                      => Or 1 3 1
```

In order to describe the innermost derivation with oracle, all we need to do is *describe the derivation as if it was a purely functional.* This means, we state that we do the first leftmost innermost step because the corresponding redex `(insert 3 [1,2])` was unfolded and in the result which is `(Or 1 [3,1,2] (1:insert 3 [2]))` the next leftmost innermost redex which is `(insert 3 [2])` is not evaluated and after that all remaining redexes are unfolded. The resulting oracle is therefore `[1,3]` and we can replay the derivation as:

```
                        [1,3]
head (insert 3 [1,2]) =>[0,3] head (Or 1 [3,1,2] (1:insert 3 [2]))
                      =>[3]   head (Or 1 [3,1,2] (1:_))
                      =>[2]   Or 1 (head [3,1,2]) (head (1:_))
                      =>[1]   Or 1 3 (head (1:_))
                      =>[0]   Or 1 3 1
```

As detailed in [7] the search tree approach is well suited to implement search strategies as traversals on the tree structure. This is also compatible with the presented approach. If, for instance the programmer would have been interested in a first solution only with, e.g., a depth first strategy, the above expression would have been evaluated to `(Or 1 3 (head (1:insert 3 [2])))` only and the corresponding oracle would have been `[1,2,0]`.

But not only can we describe the evaluation of non-deterministic operations with the same oracle but also the narrowing of free variables. This can be done by describing how the corresponding generator is evaluated. For example the narrowing derivation

```
guard (head x) 1 ={x/(a:z)} guard a 1
                 ={a/True}  1
```

can be described by the oracle `[2,7,0]`. The first three decisions describe the binding of the variable as the evaluation of the corresponding generator:

```
   [2,7,0] genList genBool ()
 =>[1,7,0] Or 1 [] (genBool () : genList genBool ())
 =>[0,7,0] Or 1 [] (Or 2 True False : genList genBool ())
 =>[7,0]   Or 1 [] (Or 2 True False : _)
```

The next three steps describe the application of `head` to the generated result:

```
   [7,0] head (Or 1 [] (Or 2 T F:_))
 =>[6,0] Or 1 (head []) (head (Or 2 T F:_))
 =>[5,0] Or 1 Fail (head (Or 2 T F:_))
 =>[4,0] Or 1 Fail (Or 2 T F)
```

And finally `[4,0]` describes the application of `guard` assuming that the programmer was searching for the first solution only, as above.

```
   [4,0] guard (Or 1 Fail (Or 2 T F)) 1
 =>[3,0] Or 1 (guard Fail 1) (guard (Or 2 T F) 1)
 =>[2,0] Or 1 Fail (guard (Or 2 T F) 1)
 =>[1,0] Or 1 Fail (Or 2 (guard T 1) (guard F 1))
 =>[0,0] Or 1 Fail (Or 2 1 (guard F 1))
 =>[0]   Or 1 Fail (Or 2 1 _)
```

The main result of the consideration is that in order to extend the oracle approach to functional logic programming, the definition of an oracle does not need any change. The main requirement is that the events are recorded in compliance with the operational semantics described in [7].

3 The debugging Tool

We have implemented the ideas introduced in the last section into a debugging tool for the language Curry. The tool is an extension of the one presented in [8] for the functional subset of Curry. In this section we will describe the basic ideas of how to present Curry derivations to the user.

3.1 Representing Non-Determinism

The derivations with `Or` constructors and their references as well as the more technical details of the completed reductions are not suited for the programmer who is looking for a bug in his Curry program. Therefore several conventions help to get a more simple view on derivations.

- unfolding of generator functions is never seen (trusted functions)
- the references of `Or` constructors are hidden; a value like `(Or 2 True False)`, for instance is represented as `(True ? False)`
- irrelevant parts of search trees (recall the `permSort` example from above) are always omitted
- when an `Or` node has only a single valid alternative; this alternative is shown rather than any failures
- calls to auxiliary functions like `sorted2` in the above example are omitted

Accordingly, the following output is generated by a step by step examination for `permSort [2,1]` in our debugging tool:[1]

```
permSort [2,1]
  permute [2,1]
    permute [1]
      permute [] => []
      insert 1 [] => [1]
    permute [1] => [1]
    insert 2 [1]
      insert 2 [] => [2]
    insert 2 [1] => [2,1] ? [1,2]
  permute [2,1] => [2,1] ? [1,2]
  sorted ([2,1] ? [1,2])
    2 < 1 => False
    1 < 2 => True
    sorted [2]
    sorted [2] => True
  sorted ([2,1] ? [1,2]) => True
permSort [2,1] => [1,2]
```

As described in [8], the tool also features a declarative debugging mode. In this mode the user can state correctness or faultiness of sub derivations to isolate erroneous rules.

3.2 Bubbling

In order to omit the representation of references in `Or` constructors without loosing semantically important information, we have adopted *bubbling* as first presented in [1]. Bubbling is related to the approach presented in Section 2.2 in

[1] For this presentation, we have deleted some redundant lines and added the spacing.

the sense that non-deterministic branching is treated (almost) like a constructor. The `Or` constructors in Section 2.2 are "lifted up" one step at a time by the rules added for completion like `head (Or r x y) = Or r (head x) (head y)`. In bubbling, in contrast, when a `(?)` is at a needed position it is moved up in the term structure until a "proper dominator" has been copied, i.e., a symbol which is above all references to that `(?)`. The exact definition of bubbling [1, Definition 5] is rather technical but the idea is quite intuitive and we will use the style of [13]. Consider the following example:

```
let l=insert 1 [2] in (head l,l) =>
let l=[1,2] ? [2,1] in (head l,l)
```

In the approach described in Section 2.2 the end result of this derivation would be $(1\ ?_1\ 2,[1,2]\ ?_1\ [2,1])$. The references (here denoted as subscript to (?)) are then needed to reconstruct the fact that both parts of the tuple share the same choice, i.e., that `(1,[2,1])` is not a valid projection of the result. In bubbling, in contrast, the next step is to copy the whole `let` expression. (If there was an outer context of this expression, that context would not be copied.) In the notions of [1], the tuple constructor is the dominator.

```
let l=[1,2] ? [2,1] in (head l,l) =>
let l=[1,2] in (head l,l) ? let l=[2,1] in (head l,l) =>
(1,[1,2]) ? (2,[2,1])
```

The advantage of this technique is that `?` is never duplicated and, thus, no references are needed. This is why we use the technique to omit the references when presenting values. In our tool, the derivation is presented as

```
main
  insert 1 [2]
    insert 1 [] => [1]
  insert 1 [2] => [1,2] ? [2,1]
head ([1,2] ? [2,1]) => 1 ? 2
main => (1,[1,2]) ? (2,[2,1])
```

As you can see, `head` is applied to the non-deterministic argument `([1,2] ? [2,1])` but the presentation at the end is the result of a bubbling procedure in the pretty printer.

3.3 Representation of Free Variables

As discussed in Section 2.3, the oracle approach maps free variables to the evaluation of the corresponding generator. As a consequence, in the strict evaluation all bindings of a given variable are already computed before any operation is applied to that variable. (Compare to the evaluation of `(guard (head x) 1 where x free)` above.) So far, the debugging tool building on this technique can therefore only access all of the bindings for that variable which will occur in the whole derivation. This can be unexpected for the user and the aim of this section is

to explain how a special representation for free variables can be supported. To achieve this, we have to make some adjustments to the search tree mechanism as described in 2.2. The first change is that we need to be able to distinguish between `Or` nodes originating from generator functions and those stemming from a call to `(?)`. To keep most of the described mechanism as similar as possible, we introduce this distinction to the type `OrRef`:

```
data OrRef = Generator Int | NoGenerator Int
```

Accordingly, we change the definitions of `(?)` and the generator operations as follows:

```
(?) :: a -> a -> a
x ? y = Or (NoGenerator r) x y where r fresh
genBool :: Generator Bool
genBool _ = Or (Generator r) True False where r fresh
```

Each narrowing step has to change the `Or` reference such that the corresponding result is not treated as a free variable anymore. For this we use the auxiliary function `narrow`:

```
narrow :: OrRef -> OrRef
narrow (Generator x)   = NoGenerator x
narrow (NoGenerator x) = NoGenerator x
```

Function `narrow` is called in those rules which were added to each function to treat the `Or` case. For example, function `head` is now completed with the following rule (instead of the one used in Section 2.2):

```
head (Or r x y) = Or (narrow r) (head x) (head y)
```

With these changes we can now give a special treatment to free variables and can show the user the derivation of `tuple x where x free)` where `tuple x = (x,not x)` as follows:.

```
main
tuple A
  not A => True ? False
tuple A => (False,True) ? (True,False)
main => (False,True) ? (True,False)
```

3.4 Unification

One of the main open problems with generator functions is how to reclaim the power of the unification operator `(=:=)`. This operator introduces a new quality to functional logic programming. Where *narrowing* only binds free variables to non-variable constructor terms, `(=:=)` can also bind free variables to other variables. In this section we can only sketch the main ideas to solve this problem for the oracle approach.

In order to include unification in the presented technique, we need a further extension of the information contained in `Or` references. In addition to the possibility to tell generator branches from ordinary ones introduced in Section 3.3, we need information about the `Or` references of the children of a generator. As a simple example, to establish the equality between to generated boolean lists

```
genList genBool ()=:=genList genBool () =>
Or (Generator 1) [] (genBool ():genList genBool ()) =:=
Or (Generator 2) [] (genBool ():genList genBool ())
```

we need not only to establish a connection between the references `1` and `2` but also between the references of the respective arguments of the `(:)`. The according change to the declaration of `OrRef` is:

```
data OrRef = Generator Int [Int]
           | Narrowed  Int [Int]
           | NoGenerator Int
```

Each generator has to include the references for its direct children in the `Or` reference of the parent. The dummy parameter now becomes a proper argument.

```
type Generator a = Int -> a
genBool i = Or (Generator i []) True False
genList gen i = Or (Generator i [j,k]) [] (gen j:genList genBool k)
  where j,k fresh
```

A free variable, e.g. `x::[Bool]`, is now introduced by `genList genBool r where r fresh`.

The next step would be to design a type of constraints. For unification we need only a single kind of constraints but, in principle, other kinds of constraints can be treated in the same way.

```
data Constraint = Eq OrRef OrRef | ...
```

We need an additional constructor `Constraint :: Constraint -> a -> a`. Constraints have to be lifted just like `Or` constructors or `Fail`, e.g.:

```
head (Constraint c x) = Constraint c (head x)
```

Finally, the projection from search trees to values described in Section 2.2 has to be extended to a constraint solver. Due to lack of space we cannot describe the implementation of such a solver and leave this part for future work.

The described extension is implemented in the debugging tool such that the evaluation of `(x=:=True:y &> (x,y) where x,y free)` can be shown as

```
main
A =:= (True : B) => Success
Success &> (True : B,B) => (True : B,B)
main => (True : B,B)
```

4 Related Work and Conclusion

We have presented a recently developed technique to record compact data about programs written in a lazy functional language. We have shown how this technique can be extended to include the advanced features of lazy functional *logic* languages, especially with respect to narrowing and non-deterministic operations. The extension to unification has only been sketched and a more thorough treatment is part of future work.

With respect to related work, the presented approach is complementary, as also described in Section 1. There are many debugging tools for lazy functional languages, cf. [10] for a survey but also for functional logic languages, cf. [9] for the most recent paper. The presented approach is about a technique to record less data about programs and how this technique can be employed to create efficient debugging tools. With the ideas developed in [3] we think that this technique can be applied to integrate all of the related approaches. The transfer of the framework described in [3] and its application to build different debugging views is therefore the next step of the development. The debugging tool built on the presented technique can be downloaded at `www-ps.informatik.uni-kiel.de/~bbr` and includes apart from a step/skip mode the possibility of declarative debugging and virtual I/O as described in [8].

References

1. S. Antoy, D.W. Brown, and S.-H. Chiang. On the correctness of bubbling. In *Proc. RTA'06*. To appear in Springer LNCS, 2006.
2. S. Antoy and M. Hanus. Overlapping rules and logic variables in functional logic programs. In *Proceedings of the 22nd International Conference on Logic Programming (ICLP 2006)*, pages 87–101. Springer LNCS 4079, 2006.
3. B. Braßel. A framework for interpreting traces of functional logic computations. *Electronic Notes in Theoretical Computer Science*, 177:91–106, 2007.
4. B. Braßel, S. Fischer, M. Hanus, F. Huch, and G. Vidal. Lazy call-by-value evaluation. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming (ICFP'07)*, pages 265 – 276, 2007.
5. B. Braßel, S. Fischer, and F. Huch. A program transformation for tracing functional logic computations. In *Pre-Proceedings of the International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'06)*, pages 141–157. Technical Report CS-2006-5, Università ca' Foscari di Venezia, 2006.
6. B. Braßel, M. Hanus, F. Huch, and G. Vidal. A semantics for tracing declarative multi-paradigm programs. In *Proceedings of the 6th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'04)*, pages 179–190. ACM Press, 2004.
7. Bernd Braßel and Frank Huch. On a tighter integration of functional and logic programming. In Zhong Shao, editor, *APLAS*, volume 4807 of *Lecture Notes in Computer Science*, pages 122–138. Springer, 2007.
8. Bernd Braßel and Holger Siegel. Debugging Lazy Functional Programs by Asking the Oracle. In Olaf Chitil, editor, *Proc. Implementation of Functional Languages (IFL 2007)*, Lecture Notes in Computer Science. Springer, 2008. To appear.

9. Rafael Caballero, Mario Rodríguez-Artalejo, and Rafael del Vado Vírseda. Declarative Diagnosis of Missing Answers in Constraint Functional-Logic Programming. In Jacques Garrigue and Manuel V. Hermenegildo, editors, *FLOPS*, volume 4989 of *LNCS*, pages 305–321. Springer, 2008.
10. O. Chitil, C. Runciman, and M. Wallace. Freja, Hat and Hood – a comparative evaluation of three systems for tracing and debugging lazy functional programs. In *Proc. of the 12th International Workshop on Implementation of Functional Languages (IFL 2000)*, pages 176–193. Springer LNCS 2011, 2001.
11. Javier de Dios Castro and Francisco J. López-Fraguas. Extra variables can be eliminated from functional logic programs. *Electron. Notes Theor. Comput. Sci.*, 188:3–19, 2007.
12. J.C. González-Moreno, M.T. Hortalá-González, F.J. López-Fraguas, and M. Rodríguez-Artalejo. An approach to declarative programming based on a rewriting logic. *Journal of Logic Programming*, 40:47–87, 1999.
13. F.J. López-Fraguas, J. Rodríguez-Hortalá, and J. 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)*, pages 197–208. ACM Press, 2007.
14. H. Nilsson and P. Fritzson. Algorithmic debugging for lazy functional languages. *Journal of Functional Programming*, 4(3):337–370, 1994.
15. H. Nilsson and J. Sparud. The Evaluation Dependence Tree as a Basis for Lazy Functional Debugging. *Automated Software Engineering*, 4(2):121–150, 1997.
16. E. Shapiro. *Algorithmic Program Debugging.* MIT Press, Cambridge, Massachusetts, 1983.
17. J. Sparud and C. Runciman. Tracing Lazy Functional Computations Using Redex Trails. In *Proc. of the 9th Int'l Symp. on Programming Languages, Implementations, Logics and Programs (PLILP'97)*, pages 291–308. Springer LNCS 1292, 1997.
18. M. Wallace, O. Chitil, T. Brehm, and C. Runciman. Multiple-View Tracing for Haskell: a New Hat. In *Proc. of the 2001 ACM SIGPLAN Haskell Workshop.* Universiteit Utrecht UU-CS-2001-23, 2001.

WCET Annotation Languages Reconsidered: The Annotation Language Challenge [1]

Albrecht Kadlec, Raimund Kirner, Jens Knoop, Adrian Prantl, Markus Schordan and Ingomar Wenzel

Institute of Computer Engineering, Institute of Computer Languages
Vienna University of Technology, Austria
email: {albrecht,ingomar,raimund}@vmars.tuwien.ac.at
{adrian,markus,knoop}@complang.tuwien.ac.at

Abstract Worst-case execution time (WCET) analysis is a prerequisite for successfully designing and developing systems, which have to satisfy hard real-time constraints. Of key importance for the precision and performance of algorithms and tools for WCET analysis are the expressiveness and usability of annotation languages, which are routinely used by developers for providing WCET algorithms and tools with hints for separating feasible from infeasible program paths.

Reconsidering and assessing the strengths and limitations of current annotation languages, we believe that contributions towards further enhancing their power and towards a commonly accepted uniform annotation language will be essential for the next major step of advancing the field of WCET analysis. To foster this development we have recently proposed the *WCET annotation language challenge*. This challenge complements the already earlier successfully launched *WCET tool challenge*. In this paper we summarize the essential features of current annotation languages and recall the WCET annotation language challenge derived from their assessment.

1 Motivation

The precision and performance of *worst-case execution time* (*WCET*) analysis depends crucially on the identification and separation of feasible and infeasible

[1] An extended version of this paper has been published in the Proceedings of the *7th International Workshop on Worst-Case Execution Time Analysis* (*WCET'07*).
This work has been partially supported by the Austrian Science Fund (Fonds zur Förderung der wissenschaftlichen Forschung) under contract No P18925-N13, *Compiler Support for Timing Analysis*, `http://costa.tuwien.ac.at/`,
the ARTIST2 Network of Excellence, `http://www.artist-embedded.org/`
and research project "Integrating European Timing Analysis Technology" (ALL-TIMES) under contract No 215068 funded by the 7th EU R&D Framework Programme.

program paths. This information can automatically be computed by appropriate tools or manually be provided by the application programmer. In both cases some dedicated language is necessary for annotating this information and making it available for a subsequent WCET analysis. Languages used for this purpose are commonly known as *annotation languages*. Over the past 15 years, an array of conceptually quite diverse proposals of annotation languages has been presented. Many of them have been used for the implementation of a WCET tool. A comprehensive survey of WCET tools and methods has been given by Wilhelm et al. [27]. Until recently, however, there was no approach towards a systematic comparison of the various approaches proposed on annotation languages for WCET analysis [14].

The goal of our approach of [14] was three-fold: (1) To identify an array of important universally valid criteria, in which the usefulness of annotation languages for WCET analysis becomes manifest. (2) To investigate and classify a selection of prototypical representatives of annotation languages used in practice along these criteria in order to shed light on the relative strengths and limitations of the different annotation concepts. (3) Based on these findings to extend the invitation to researchers working in this field to contribute to the challenge of designing novel and superior annotation languages, which will support the development of enhanced WCET algorithms and tools which will outperform their current counterparts for WCET analysis: The so-called *WCET annotation language challenge*.

As pointed out in [14], we believe that mastering the WCET annotation language challenge will be the key for further advancing the field of WCET analysis. Moreover, we believe that it will also be essential in order to enable the recently successfully launched *WCET tool challenge*, which has attracted the attention of many WCET tool developers [6,26], to unfold its strength and impact in full.

In this paper we summarize the essential findings of the comparison of an array of prototypical annotation languages presented in [14] and the conclusions drawn from this comparison.

2 Assessment Criteria

The criteria we use in order to assess the strengths and limitations of WCET annotation languages can be divided into two groups of *language design* and *usability* criteria. While the characteristics of the language design criteria are essentially under control when designing the language, the characteristics of the usability criteria are essentially an outcome of the characteristics of the language design criteria. In addition we consider a singleton third criterion, which is orthogonal to the other criteria. This is the existence of a *tool* using the annotation language. It is worth noting that the availability of a tool need not directly be related to a specific property or feature of an annotation language. In fact, there may be manyfold reasons why a tool has been developed, and vice versa, why not. Actually, these reasons need not necessarily be related to the

language at all. Nonetheless, we consider the availability of a tool an important indicator of the general usefulness and usability of an annotation language. We thus report the existence of tools, however, it is beyond the scope of this paper to assess the quality of any such tool. Readers interested in this might refer to the article by Wilhelm et al. [27].

Here, we proceed with an overview of the assessment criteria of annotation languages we use and which we discuss in more detail subsequently.

1. Language Design
   (a) Expressiveness
   (b) Annotation placement and abstraction level
   (c) Programming language
2. Usability
3. Tool Support

Expressiveness. We consider expressiveness of an annotation language the most important criterion at all. Intuitively, expressiveness refers to the capability of an annotation language to describe control-flow paths. Key for the expressiveness of an annotation language is the type of flow information it allows to describe. We call an annotation language *complete*, if it allows to precisely describe all feasible paths of arbitrary terminating programs. The capability of an annotation language to cope with inter-procedural program flow or selected iteration ranges of loops are other important aspects of expressiveness. Important setscrews a language designer can use to control the expressiveness of an annotation language are the means and their capabilities to deal with *loop bounds*, *triangle loops*, and, more generally, the *context sensitivity* of loop iterations and procedure or function incarnations, and the *execution order* of statements.

Annotation placement and abstraction level. The question of where to place annotations and at which level of abstraction has a strong impact on the usability of an annotation language because it directly affects the demands on a programmer when using a language.

First, it has to be decided if annotations shall be placed at the location of the source code statements they describe, or in a separate file? None of the two options is always superior over its counterpart. As a rule of thumb we have: If annotations are provided manually, it is usually more convenient to directly annotate the code. If annotations are computed automatically, it is often pragmatically advantageous to provide annotations in separate files.

Second, it has to be decided if the source code or the object code shall be annotated. Taking a (human-centered) usability perspective, annotating the source code appears generally preferable. This might be obvious, if code annotations are manually provided. However, it also holds, if flow information is automatically computed because it is often obligatory or at least desirable to verify automatically computed annotations manually, e.g., to verify that the correct execution context has been taken into account.

Closely related to this is the issue of establishing a mapping between source code and object code: If an object code-based annotation language is used to express the behavior of constructs of the original programming language it is necessary to establish a correspondence between the object code and the source code. This can be achieved e.g. by defining a set of so-called *anchors*, special language constructs, which can be recognized after compilation, such as loops or procedure calls.

Programming language. Restricting a programming language to a well chosen sublanguage and tailoring an annotation language towards this sublanguage is an important means to control the expressiveness, precision, and efficiency of a WCET analysis using this language. For example, an annotation language can be limited to *reducible* code. Also the WCET calculation methods which are compatible with an annotation language can constrain the features of a programming languages, which can meaningfully be handled. Another source, which can impose restrictions on the programming language, are the techniques for the automatic calculation of flow information. For example, a technique might not support floating point operations.

It is also an important feature of an annotation language if it supports path analysis of the object code. This is crucial because compared to path analysis at the source code level this imposes additional challenges at the object code level. For example, source code typically makes use of high-level control-flow statements which simplifies the construction of the control-flow graph (CFG) of a program. For object code, a precise (re-) construction of the CFG requires usually additional annotations.

Usability. The usability of an annotation language is possibly best reflected by the skills and the amount and the complexity of work it demands from a programmer when using it. It is also reflected by the knowledge which is required beyond the annotation language itself, e.g. about the WCET analysis expected to make use of it, maybe even of the implementation specifics of this technique as it might affect its performance. Similarly, this holds for the amount of work required to update a program annotation in response to an update of the program. Another issue referred to concerns the ability to cope with annotations that are automatically provided by a tool.

In principle, there are two potential classes of users that provide code annotations: Programmers writing manual code annotations, and tools automatically computing annotations by means of some code analysis.

For code annotations which are to be provided manually it is most important that the program behavior can be described concisely and compactly. As an extreme case, the size of an annotation describing a specific program property, may grow exponentially with the program size. For code annotations which are automatically computed, it is important that the underlying techniques are able to deliver their information in a format which is supported by the annotation language.

It is also an important issue if a WCET calculation method which is compatible with an annotation language can provide the user with adequate information explaining its results. For example, *Integer Linear Programming* (*ILP*) with flow constraints as very often used in practice can only provide information about the execution frequency of statements, but not on their execution order.

All this shows that usability is the outcome of the interplay of several factors, in particular, of the complex interaction of an annotation language and the possible support for applying this language which is provided by the (tool) environment it is used in. Assessing the usability of an annotation language thus implicitly amounts to an assessment of its usability with respect to a specific global environment, which might even change over time. This, however, is beyond the scope of this paper. In addition to usability, we thus introduce a second more specific term, which we call the *intricacy* of an annotation language. We refer to this term in order to assess the language-inherent conceptual and technical complexity of an annotation language, detached from any environment or tool support of using it.

Tool Support. As mentioned above, the availability of a tool using a specific annotation language can be considered an indicator of the general usefulness and usability of this language. We therefore report the availability of tools but we do not aim at assessing their quality.

3 WCET Fundamentals

We consider the general typology of current WCET calculation methods and the types of flow information they rely on as WCET fundamentals which we recall next.

Types of flow information. Intuitively, flow information provides a WCET calculation method with information about the dynamic behavior of a program. Typically, the (interprocedural) control-flow graph of a program is used to provide this information. The various kinds of flow information can roughly be classified as follows:

1. Explicit execution frequency
2. Explicit execution order
3. Context-sensitive flow information
   (a) Loop-context sensitive flow information
   (b) Call-context sensitive flow information

Explicit execution frequency information describes the execution count of nodes or edges of the control-flow graph. In principle, this information can be given as absolute execution count of a code location or as a relation between the execution count of one code location and another one. In practice, this kind of information is usually provided in terms of linear equations between the execution count of different code locations.

Explicit execution order information describes patterns of execution order of nodes or edges of the control-flow graph of a program. This information allows WCET calculation methods to cope with the intricacies of advanced modern processors, where the execution time of an instruction can depend on the execution history.

Context-sensitive flow information is relevant for reliably capturing the effect of instructions which may be executed multiple times within a program execution. In principle, two major sources of context-sensitive flow information can be distinguished: Instructions executed within a loop and instructions executed within a possibly recursive function or procedure which is called multiple times.

Two examples of concrete flow information are *loop bounds* and *recursion bounds*. Such bounds information is mandatory for any WCET calculation method. It can thus be considered the minimal flow information necessary for WCET analysis.

WCET calculation methods. WCET calculation methods can roughly be divided into dynamic and static techniques. Intuitively, dynamic methods are measurement-based and run the program to figure out the worst case execution time, whereas static methods are analysis-based and compute a bound for the worst case execution time of a program without running it. In this paper, we concentrate on static methods. The static methods can roughly be classified as follows:

1. Timing Schema Approaches
2. Path-based Approaches
3. Implicit Path Enumeration Technique (IPET) Approaches

Timing schema approaches operate on the *abstract syntax tree* (*AST*) of a program. Intuitively, each leaf of the tree representing elementary operations is assigned an execution time, each inner node an operation allowing to compute its execution time as a function on the execution time of its successor nodes. This directly induces a hierarchical approach for computing the worst case exection time of a program. Historistically, timing schema based approaches were among the first WCET calculation methods used in practice [22,24,20]. Refinements of these approaches e.g. towards an improved handling of nested loops have been proposed more recently [3]. The popularity of the timing schema approaches is in part due to their conceptual simplicity, which simplifies their implementation.

Unlike timing schema approaches, *path-based* approaches decompose a program into fragments. For each of the fragments they determine a program path with maximum execution time [7,25]. These times are then combined to the worst case execution time of the program. Path-based approaches have been developed for capturing the effects of pipelines, however, they are less appropriate for taking global timing effects into account, like cache behavior.

Implicit path enumeration technique (*IPET*) approaches perform an implicit search for the longest path of a program without enumerating paths explicitly [16,23]. This distinguishes them from path-based approaches. Intuitively, IPET

approaches model the control flow of a program by constraints. Typically, only linear constraints are used in order to reduce the complexity of solving the resulting constraint problem. This leads to an *integer linear program* (*ILP*), which can be solved by off-the-shelf open source or commercial ILP solvers.

4 WCET Annotation Languages

In this section we recall the essential features of the seven annotation languages, which we selected as prototypical representatives for our conceptual comparison of WCET annotation languages.

1. The Timing Analysis Language TAL
2. The Path Language PL and Information Description Language IDL
3. Linear Flow Constraints
4. SPARK Ada
5. Symbolic Annotations
6. The Annotation Language of Bound-T
7. The Annotation Language of aiT

In the following, we focus on the most relevant key facts concerning these languages. A more detailed description of these languages and the calculation methods and tools using them can be found in [14].

The *Timing Analysis Language* (*TAL*) has been developed by Mok et al. [18]. It is a timing schema approach. The TAL language is an integral part of the timing analysis system developed at the University of Texas and is used by the tool *timetool* [2].

The *Path Language* (*PL*) has been developed by Park and Shaw [24,20,21,19]. It is a path-based approach, which describes feasible and infeasible paths of a program by means of regular expressions. Later on Park developed a more high-level variant of PL called *Information Description Language* (*IDL*) [19], which is easier to use than the more low-level PL.

Linear Flow Constraints are typically used by IPET approaches [4,13] as already discussed in the previous section. We thus proceed with *SPARK Ada*. This is a subset of Ada83 which is extended by a special kind of comments which are used for both program proof and timing analysis. Spark Ada programs can be analyzed by the *Spark Proof and Timing System* (*SPATS*), which is based on symbolic execution.

Symbolic Annotations is a term which we coined to denote an approach proposed by Blieberger [1]. This approach combines aspects of a pure annotation language with those of a programming language extension. The clue of this approach is the invention of so-called *discrete loops*. These can be considered a generalized and more flexible kind of for-loops. Exploiting the structural properties of discrete loops, however, loop bounds can often automatically be computed by simple mathematical reasoning.

Bound-T is a commercial WCET tool originally distributed by Space Systems Finland Ltd. It has been developed by Holsti et al. [10,11,9] and is currently

marketed by Tidorum Ltd. A specialty of the annotation language of Bound-T is that it is designed to be usable both within high-level languages programs and assembler programs.

The *Annotation Language of aiT*, finally, is used by the ait WCET tool, a commercial tool developed by AbsInt Angewandte Informatik GmbH, Germany. This tool targets different hardware architectures including ARM7, Motorola Star12/HCS12, and PowerPC 555 [5,8]. A specialty of this tool and its annotation language is to start from binary files as input to be analyzed.

5 Main Results

Table 1 summarizes the major findings of our comparison of the seven languages we selected for assessing and highlighting the strengths and limitations of current WCET annotation languages.

Most of the criteria listed in the leftmost column of Table 1 are self-explaining or have been discussed before, except of triangle loops and the various kinds of context-related information.

Intuitively, *triangle loops* are nested loops which meet a triangular pattern in the iteration space (i, j). The two IPET-based methods in our comparison, linear flow constraints and Bound-T, allow a precise description of the behaviour of triangle loops by allowing the use of equalities and inequalities in the specification of constraints.

Often the timing behaviour of the first iteration of a loop is different from that of subsequent iterations, e.g. because of cache effects. *Loop context-sensitive* annotations allow to make such differences explicit. Similarly, the bounds of loops inside of procedures and functions depend often on the values of their input parameters. *Context-sensitive* annotations of the calling context allow to differentiate between the various calling scenarios and thus to obtain more precise analysis results.

Application context-sensitive annotations, finally, are a specialty used in SPARK Ada. It refers to a feature called *modes* which allow to describe multiple annotations for a function depending on different input parameters. This resembles the scenario of calling context-sensitivity without being exactly the same. We thus introduced the term *application context sensitivity* for this feature of SPARK Ada.

Table 1 illustrates that none of the seven prototypical annotation languages selected for our comparison uniformly outperforms its competitors. They all have their own individual strengths and limitations. This became the more apparent, if we were to take further criteria into account, e.g., the possibility and ease of reconstructing the control-flow graph on the object-code level such that it precisely reflects its counterpart on the source-code level [15] or the consideration of application domains of annotation languages which go beyond pure WCET analysis as e.g. recently proposed by Lisper [17].

Criteria	Annotation Language							
	TAL	*PL and IDL*	*Linear Flow Constraints*	*Bound-T*	*aiT*	*SPARK Ada*	*Symbolic Annotations*	*Challenge*
Expressiveness	Timing schema	Regular expressions	Constraint-based	Constraint-based	Constraint-based	Loop-bounds	Loop-annotations	
Loop-bounds	yes	yes	yes	yes	yes	yes	yes	yes
Triangle-loops	yes	no	yes	some	yes	no	yes	yes
Calling context	yes	no	possible	implicit	no	explicit	no	yes
Loop context	no	no	possible	no	no	no	no	yes
Appl. context	no	no	no	no	yes	yes	no	yes
Execution order	no	yes	no	no	no	no	no	yes
Intricacy of Annotations	high	medium to high	medium	medium	medium	low	low to medium	as low as possible
Annot. placement	External TAL-script	Ideally inside the source code	Ideally inside the source code	External file	External file; partially inside source code	Source code comments	Integral part of the source language	– Design Decisions –
Abstraction level								
Source code	no	yes	yes	no	yes	yes	yes	
Object code	yes	no	yes	yes	yes	no	no	
Program. language								
Implementation	Asm/C	C	-	C, Ada	Asm/C	Ada	Ada	
General Scope	-	Any structured language	Any structured language	Any structured language	Any structured language	-	Any structured language	
Tool available	yes	no	yes	commercial	commercial	yes	prototype	yes

Table 1. Assessment summary

6 Conclusions

The summary of Table 1 demonstrates that the languages proposed and used so far for WCET analysis all have their own specific profile of strengths and limitations. The demand for an annotation language, which combines the individual strengths of the known annotation languages, while simultaneously avoiding their limitations, is thus apparent. In Table 1 this demand is reflected by the right-most column denoted by "*Annotation Language Challenge*." It grasps the summarized strengths of the different annotation concepts. Developing a language (or an annotation concept), which enjoys this profile is the central challenge, which we derive from our investigation, and which we would like to present to the research community.

This challenge, however, is not the only challenge, which is suggested by the findings of our investigation. It is obvious that an annotation language and a methodology for computing the WCET of a program based on annotations given in this language are highly intertwined. Expressiveness delivered by an annotation language, which cannot be exploited by a WCET computation methodology, is in vain. Vice versa, the power of a WCET computation methodology cannot be evolved if the annotation language is too weak to express the needed information. This mutual dependence of annotation languages and WCET computation methodologies suggests two further challenges. Which annotation language serves a given WCET computation methodology best? And vice versa: Which WCET computation methodology makes the best use of a given annotation language?

Of course, the meaning of "best" must be made more precise in order to be practically useful. We argue that the underlying notion of the relation "bet-

ter" has several dimensions, each of these leading to possibly different solutions. Besides parameters like ease of use, we consider the parameters of power and performance and the trade-off between the two most important.

Summing up, this results in the following *challenges*:

1. Finding an annotation language, which enjoys the individual strengths of the known annotation languages while avoiding their limitations.
2. Finding an annotation language, which serves a given WCET computation methodology best.
3. Finding a WCET computation methodology, which makes the best use of a given annotation language.

It is worth noting that these challenges can be considered on various levels of refinement, depending for example on the notion of the relation "better" as discussed above. The challenges above thus represent a full array of more fine-grained challenges rather than exactly three individual challenges.

In a companion paper published in the present proceedings [12], too, we make a proposal towards such a new annotation language by highlighting ingredients, which we consider essential for an annotation language that can serve as a commonly accepted uniform WCET annotation language in the future.

References

1. Johann Blieberger. Discrete loops and worst case performance. *Computer Languages*, 20(3):193–212, 1994.
2. Moyer Chen. *A Timing Analysis Language - (TAL).* Dept. of Computer Science, University of Texas, Austin, TX, USA, 1987. Programmer's Manual.
3. Antoine Colin and Isabelle Puaut. A modular and retargetable framework for tree-based wcet analysis. In *Proc. 13th Euromicro Conference on Real-Time Systems*, pages 37–44, Delft, Netherland, June 2001. Technical University of Delft.
4. Jakob Engblom and Andreas Ermedahl. Modeling complex flows for worst-case execution time analysis. In *Proc. 21st IEEE Real-Time Systems Symposium (RTSS)*, Orlando, Florida, USA, Dec. 2000.
5. Christian Ferdinand, Reinhold Heckmann, and Henrik Theiling. Convenient user annotations for a WCET tool. In *Proc. 3rd International Workshop on Worst-Case Execution Time Analysis*, pages 17–20, Porto, Portugal, July 2003.
6. Jan Gustafson. The WCET tool challenge 2006. In *Preliminary Proc. 2nd Int. IEEE Symposium on Leveraging Applications of Formal Methods, Verification and Validation*, pages 248 – 249, Paphos, Cyprus, November 2006.
7. Christopher A. Healy, Robert D. Arnold, Frank Mueller, David Whalley, and Marion G. Harmon. Bounding pipeline and instruction cache performance. *IEEE Transactions on Computers*, 48(1), Jan. 1999.
8. Reinhold Heckmann and Christian Ferdinand. Combining automatic analysis and user annotations for successful worst-case execution time prediction. In *Embedded World 2005 Conference*, Nürnberg, Germany, Feb. 2005.
9. Niklas Holsti. Bound-t assertion language: Planned extensions. Technical report, 2005.
10. Niklas Holsti, Thomas Långbacka, and Sami Saarinen. Worst-case execution time analysis for digital signal processors. In *European Signal Processing Conference 2000 (EUSIPCO 2000)*, 2000.

11. Niklas Holsti, Thomas Långbacka, and Sami Saarinen. *Bound-T timing analysis tool User Manual*. Tidorum Ltd, 2005.
12. Albrecht Kadlec, Raimund Kirner, Peter Puschner, Adrian Prantl, Markus Schordan, and Jens Knoop. Towards a common WCET annotation language: Essential ingredients. In *Proceedings of the 25th Annual Workshop of the GI-FG 2.1.4 "Programmiersprachen und Rechenkonzepte", Bad Honnef, Germany*, 2008.
13. Raimund Kirner. The programming language WCETC. Technical report, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-1, 1040 Vienna, Austria, 2002.
14. Raimund Kirner, Jens Knoop, Adrian Prantl, Markus Schordan, and Ingomar Wenzel. WCET Analysis: The Annotation Language Challenge. In *Post-Workshop Proceedings of the 7th International Workshop on Worst-Case Execution Time (WCET 2007) Analysis*, pages 83 – 99, 2007.
15. Raimund Kirner and Peter Puschner. Classification of code annotations and discussion of compiler-support for worst-case execution time analysis. In *Proc. 5th International Workshop on Worst-Case Execution Time Analysis*, Palma, Spain, July 2005.
16. Yau-Tsun Steven Li and Sharad Malik. Performance analysis of embedded software using implicit path enumeration. In *Proc. 32nd ACM/IEEE Design Automation Conference*, pages 456–461, June 1995.
17. Björn Lisper. Ideas for annotation language(s). Technical Report Oct. 25, Department of Computer Science and Engineering, University of Mälardalen, 2005.
18. Aloysius K. Mok, Prasanna Amerasinghe, Moyer Chen, and Kamtorn Tantisirivat. Evaluating tight execution time bounds of programs by annotations. In *Proc. 6th IEEE Worksop on Real-Time Operating Systems and Software*, pages 74–80, Pittsburgh, PA, USA, May 1989.
19. Chang Y. Park. Predicting program execution times by analyzing static and dynamic program paths. *Real-Time Systems*, 5(1):31–62, 1993.
20. Chang Y. Park and Alan C. Shaw. Experiments with a program timing tool based on a source-level timing schema. *Computer*, 24(5):48–57, May 1991.
21. Chang Yun Park. *Predicting Deterministic Execution Times of Real-Time Programs*. PhD thesis, University of Washington, Seattle, USA, 1992. TR 92-08-02.
22. Peter Puschner and Christian Koza. Calculating the maximum execution time of real-time programs. *The Journal of Real-Time Systems*, 1:159–176, 1989.
23. Peter Puschner and Anton V. Schedl. Computing maximum task execution times – a graph-based approach. *Journal of Real-Time Systems*, 13:67–91, 1997.
24. Alan C. Shaw. Reasoning about time in higher level language software. *IEEE Transactions on Software Engineering*, 15(7):875–889, July 1989.
25. Friedhelm Stappert and Peter Altenbernd. Complete worst-case execution time analysis of straight-line hard real-time programs. *Journal of Systems Architecture*, 46(4):339–355, 2000.
26. Lili Tan and Klaus Echtle. The WCET tool challenge 2006: External evaluation – draft report. In *Handout at the 2nd Int. IEEE Symposium on Leveraging Applications of Formal Methods, Verification and Validation*, Paphos, Cyprus, November 2006. 13 pages.
27. Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckman, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The worst-case execution time problem - overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(3):36:1–36:53, April 2008.

Towards a Common WCET Annotation Language: Essential Ingredients[1]

Albrecht Kadlec, Raimund Kirner and Peter Puschner+
Adrian Prantl, Markus Schordan and Jens Knoop*

Vienna University of Technology,
+Institute of Computer Engineering, Treitlstraße 3/E182.1, Wien, Austria,
{albrecht, raimund, peter}@vmars.tuwien.ac.at
*Institut of Computer Languages, Argentinierstraße 8/E185.1, Wien, Austria,
{adrian, markus, knoop}@complang.tuwien.ac.at

Abstract. Within the last years, ambitions towards the definition of common interfaces and the development of open frameworks have increased the efficiency of research on WCET analysis. The *Annotation Language Challenge* for WCET analysis has been proposed in line with these ambitions in order to push the development of common interfaces also to the level of annotation languages, which are crucial for the power of WCET analysis tools.
In this paper we present a list of essential ingredients for a common WCET annotation language. The selected ingredients comprise a number of features available in different WCET analysis tools and add several new concepts we consider important. The annotation concepts are described in an abstract format that can be instantiated at different representation levels.

Keywords: Worst-case execution time (WCET) analysis, annotation languages, WCET annotation language challenge.

1 Why a Common WCET Annotation Language?

The situation for WCET analysis is very heterogeneous. Within the real-time community it is a well known fact that manual annotations are needed to assist non-perfect analyses. Various tools exist providing different levels of sophistication [17]. However, as the *WCET Tool Challenge* [6] has shown, few tools share the same target hardware, analysis method or annotation language.

[1] An extended version of this paper has been published in the Preliminary Proceedings of the *8th International Workshop on Worst-Case Execution Time Analysis* (*WCET08*).
This work has been partially supported by the Austrian Science Fund (Fonds zur Förderung der wissenschaftlichen Forschung) within the research project "Compiler-Support for Timing Analysis" (CoSTA) under contract No P18925-N13, by the ARTIST2 Network of Excellence (http://www.artist-embedded.org/), and the research project "Integrating European Timing Analysis Technology" (ALL-TIMES) under contract No 215068 funded by the 7th EU R&D Framework Programme.

While a multitude of targets is beneficial and a diversity in tools and methods is favorable, a common annotation language is *required* for an accepted set of benchmarks in order to evaluate the various tools and methods. Still, as a direct consequence of the first WCET Tool Challenge a set of accepted benchmarks has already been collected, without such annotation support.

To enable common annotations within these benchmarks, the *WCET Annotation Language Challenge* [11] has formulated the need for a common annotation language. This language is a means of specifying the *problem-inherent information* in a tool- and methodology-*independent* way, supporting, e.g., static analysis equally well as measurement-based methods, thus allowing the combination of their results. It must also be expressive enough to master the difficult task of providing annotations at the *source* level, which is the natural specification level, as well as supporting the annotation of binary or object code, if the source code is not available, e.g., for closed sources like operating systems or libraries.

Therefore, a common language may allow the tool developers to concentrate on their analysis methods, creating interchangeable building blocks within the timing analysis framework, as intended by ARTIST2 [9]. By using this common annotation format as a common interface, tools can evaluate the same set of sources for a fair comparison of performance and may exchange analysis results to synergetically supplement each other. The steps of manual annotation, automatic annotation and timing analysis can be repeated, thus iteratively refining the analysis results.

All this should foster common established practices and may, eventually, lead to standardization, resulting in a broader dissemination of WCET analysis throughout research and industry.

2 Basic Concepts

2.1 Definitions

Flow Constraints: We define *flow constraints* to be any information about the control or data flow of a program code. Data flow, however, is not only meant in the sense of *def-use chains*, but, for example, variable-value ranges at program locations. Typical examples of flow constraints are loop bounds or descriptions of (in)feasible paths.

Timing Constraints: We define *timing constraints* to be any information that is introduced in order to describe the search space of the WCET analysis.

Because control and data flow represent the basis for the WCET analysis, the *flow constraints* of a program are always part of the *timing constraints*. An example of a *timing constraint* not being a *flow constraint* is the specification of access times of different memory areas.

Constraints versus Annotations: We distinguish between the *timing constraints* and the *timing annotation* of program code. The timing constraints are the information per se and the timing annotation is the linkage of the timing constraints with the program code.

There are different possibilities of how to annotate the program code with timing constraints. One possibility to annotate the program is to write the timing constraints directly into the source code, either as native statements of the programming language or as special comments. It is also possible to place timing constraints in a separate file, if the source code may not be changed.

If a programmer has to annotate the program modules at different representation levels a common syntax for the different representation levels would be especially beneficial and useful.

2.2 Layers

The WCET of a program cannot be determined precisely without knowing information about the target-computer platform on which the program will be used. The computer platform of a program includes, for example, the development tools, the operating system, the hardware, and the application environment. Naturally, the computer platform is sliced into layers to benefit from the independence of different parts that constitute the computer platform. For example, the operating system is an optional layer that may be placed on top of the hardware layer, and again, the layer of the development tool chain may be on top of the operating system.

These layers are the key to the *reuse* of timing annotations in case a layer is changed. For example, if we change the processor type (hardware layer) but still use exactly the same code binary, any timing constraints describing the behavior of the *build-and-run layer* can still be reused, if it does not specify explicit times.

A prerequisite for the smooth replacement of layers is that each annotation has a layer specified in its definition. A layer is replaced by disabling the current instance of the layer and enabling another one as input for the analysis.

Note that the layers are not fixed, but rather open for extensions. For example, if an operating system delivered in binary form has different absolute times specified for different processor types, it does make sense to specify them in a combined OS/HW layer besides the other OS and HW layers.

2.3 Validity of Timing Constraints (Timing Invariants vs. Fictions)

The goal of WCET analysis is to calculate a precise WCET bound. However, the developer might also be interested in experimenting with the timing constraints to analyze changes of the program behavior, e.g., to tune the system. For example, the developer might specify a fictive loop bound to determine the influence of the loop on the overall timing. As another example, the developer might want to test an absolute time bound for a code section independently of the real execution time. In both scenarios, timing constraints are not necessarily used to describe a superset of the real program behavior.

In WCET analysis research, program annotations are typically assumed to describe a superset of the possible system behavior, i.e., system invariants. We extend this annotation concept to information that does not have to be a superset of the system behavior. We call all timing constraints that describe a superset of the possible system behavior *timing invariants*. In contrast, we introduce *timing fictions* as arbitrary timing constraints the user might want to use for experimenting with the timing behavior of the system. We add a flag to each timing annotation to mark it either as a *timing invariant* or a *timing fiction*.

The intention of introducing *timing fictions* is not to foster its use for WCET analysis, because *timing fictions* may cause an underestimation of the WCET. But in case that a developer wants to experiment with the sensitivity of the timing behavior, then it is an additional safety feature if the user is able to explicitly mark such timing constraints as *timing fictions* and has to enable them explicitly to be included in the analysis.

Definition 1. *(Timing Invariant): A timing constraint C is a* timing invariant *at its associated annotation layer L, iff for all possible systems that use annotation layer L, it holds that for all possible initial system states the system execution fulfills the timing constraint C. If a timing constraint is associated with more than one layer, then the condition has to hold for all possible systems that use all of its associated layers.*

Definition 2. *(Timing Fiction): If a timing constraint C is not a* timing invariant *at its associated annotation layer, then it is a* timing fiction.

In the case that *timing invariants* and *timing fictions* are in conflict, the semantics of *timing fictions* is to override conflicting *timing invariants*. Whenever a *timing invariant* is overridden due to a *timing fiction*, the WCET analysis tool should give a log entry to the user.

The following provides examples of *timing invariants* and *timing fictions*:

```
void f (int a, char[] b)
{
  int i;
  a = a % 20;
  for (i=0; i<a; i++) //loop1
  {
    if (i%2 == 0)
      b[i] = a; //m1
    else
      b[i] = 0; //m2
  }
}
```

Timing Invariant:
Expressing as linear flow constraint that the `then`-path is executed at least as often as the `else`-path: $m1 \geq m2$ (see annotation **C2.3**)

Timing Fiction:
Specifying a lower and upper loop bound of 40: $LB(loop1) = 40 \ldots 40$ (see annotation **C2.1**)

In the *timing fiction* example with loop bound $LB(loop1) = 40 \ldots 40$, an IPET-based WCET analysis tool typically transforms the program structure into flow equations and the fictive loop bound is transformed into a flow constraint. In this case, the *timing fiction* redefines the execution count of control-flow edges in the final WCET calculation.

2.4 Checking of Invariants

Manual annotations are potentially error-prone and may yield incorrect WCET estimates. In the case that timing constraints originate from the operation environment it is, however, possible to "lift" operation environment information to the *program layer*, e.g., by inserting range checks and similar assertions wherever appropriate.

```
int count = read_from_sensor();
while (count ≥ 0) {
  count--;
  ...
```

If we assume that the environment dictates that the return value of read_from_sensor() is in the interval [0,47], an upper loop bound of 48 would be an *invariant at the operation layer* and a *fiction at the program layer*.

```
int count = read_from_sensor();
if(count < 48) {
  while (count ≥ 0) {
    count--;
    ...
  } else {
    error();
  }
```

However, if we specialize the program, the loop bound of 48 becomes an invariant at the program layer.

As a result of lifting annotations to the program layer, the resulting program becomes a specialized instance of the original program. Because the assertions allow the compiler to perform additional optimizations, the specialized program can also have better performance than the original program. These kinds of assertions can easily be generated by an automatic tool and could be valuable for diagnosis and testing of annotations. An example of using runtime checks with special support by the compiler is Modula/R: the Modula/R compiler optionally generates for each source-code location that is referenced by a timing constraint a separate counter variable where an exception is raised at runtime if their specified bound is exceeded [16].

3 Ingredients of the WCET Annotation Language

In the following we describe essential ingredients for a WCET annotation language. The different timing constraints are described at a conceptual level without focusing on the concrete syntax of an annotation language. We use ANSI C code examples to illustrate the usefulness of the different timing constraints. The definition of a concrete syntax is beyond the scope of this paper. We propose the following categories of ingredients, which are detailed in the rest of this section:
C1 Annotation Categorization
C2 Program-Specific Annotations
C3 Addressable Units
C4 Control Flow Information
C5 Hardware-Specific Annotations

C1 Annotation Categorization
We define attributes for timing constraints to categorize and group them. These categorization attributes help to organize, check, and maintain timing annotations. Supporting the maintenance of timing annotations is a very important aspect to improve the correctness of timing constraints. For example, if a user writes an annotation with speculative constraints just for testing the influence on the timing behavior of the system, there is the potential danger that he/she forgets to remove such an annotation from the program later on. Further, whenever code is reused or parts of the computer platform are changed, it is necessary to identify those annotations that have to be checked or adapted. The categorizations **C1.1**, **C1.2**, and **C1.3** are orthogonal categorizations, but their joint use is intended.
C1.1 Annotation Layer
Each timing constraint has associated an *annotation layer* to describe its validity. As described in Section 2.2, the WCET of a program depends on its computer platform. The computer platform is typically divided into several layers, allowing the customization of the system at each layer. As shown in Figure 1 we propose to support the specification of at least the following three *annotation layers*:
Program Layer: If an annotation belongs solely to the program layer, the timing constraint is assumed to be platform-independent. Here it is important to note that in programming languages like C or C++ the functional behavior is not fully platform-independent, i.e., some timing constraints about the control flow may already belong to the *computer-platform layer*.

Computer-Platform Layer: The computer platform of a program includes everything necessary to execute the program. If a finer granularity is needed, the platform may be divided into different layers, like, for example, the build and run environment, the operating system, any middleware, and also the hardware (as shown in Figure 1.a). For example, the cache geometry and the cache miss penalty may be specified at the hardware layer. As another example, knowing the attached flash memory device, one may specify the time needed for the completion of a write access.
Figure 1 also shows the difference between the orthogonal *layers* and the interface, a *platform* presents to a stack of layers. In Figure 1.a we see the different annotation layers, including the *computer-platform layers*, each of them clearly separated from the others. Please note the difference between a *computer-platform layer* (a name of an annotation layer) and a *platform* (as described in the MDA [13] of the OMG). In contrast to an annotation layer, a platform subsumes all the annotation layers below it. The platform can also be seen as an interface that comprises the information belonging to all annotation layers below it. Thus, as shown in Figure 1.b, the system behavior influenced by each interface contains the behavior of all annotation layers below it.

Operation Layer: The *operation layer* describes the usage of the computer system, i.e., how the environment of the system is configured and how the environment behaves. For example, timing constraints at the application layer may describe that the computer system is connected to three sensors, implying that a loop in the software to poll these sensors will iterate exactly three times.

The *program-*, *computer-platform-* and *operation-layers* are examples, only. Based on the specific system architecture, the user may refine the layering to further annotation layers. It can also happen that a timing constraint is associated to multiple annotation layers. However, whenever possible, it is advised to split such constraints into multiple constraints where each constraint belongs only to a single annotation layer. Note that the layer stack suggested by Figure 1.a is not mandatory; layers may be also placed horizontally. But the important point is that the different layers should be orthogonal, so that it is relatively easy in the system to exchange a layer and its specific timing annotations.

For timing constraints that refer to annotation layers other than the program layer, or timing constraints that represent *fictions*, more care has to be taken to ensure their intended use. For example, a loop bound may be tighter using information from the operation layer, as opposed to using only information from the program layer. Constraints refined with information from the operation layer are associated naturally also to the operation layer.

C1.2 Annotation Class

The annotation class is an attribute to describe the validity of timing constraints. As described in Section 2.3, besides the *timing invariants* we also introduce *timing fictions* as additional class of timing constraints. Each timing constraint should therefore contain a flag that indicates its class.

Invariants: *Invariants* are used to explicitly annotate information which is assumed to be valid with respect to the concrete semantics of the associated *annotation layer*.

Fictions: *Timing fictions* are used to provide fictive timing constraints to experiment with the sensitivity of a system's timing behavior.

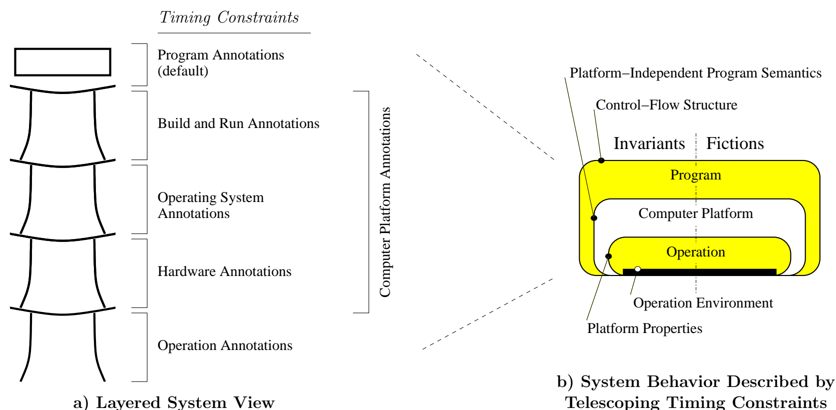

Fig. 1. Layered Timing Constraints

The criterion of whether a timing constraint is an *invariant* (and not a *fiction*) is not only whether it holds for each possible input data on the program code. This is because, as shown in Figure 1.b, the system can be annotated at different layers (layers are described by the timing-constraint attributes **C1.1**).
For example, if a timing constraint describes properties of the *computer-platform layer*, we have to look at the concrete computer platform to decide whether this timing constraint is a *timing invariant* or a *timing fiction*.

C1.3 Annotation Group

The grouping mechanism allows for different WCET evaluations. For each annotation group a separate WCET calculation with its own set of timing constraints can be conducted.
There are several reasons why one might use different sets of timing constraints. For example, one might want to use and annotate different scenarios at the *operation layer*, or different tool chains at the *computer-platform layer*, etc. *Timing fictions* can be organized in groups as well to ensure their selective and correct use.
The grouping mechanism allows us to give each timing constraint membership to multiple groups. A group is a symbolic name together with a description field. There is no special semantics behind the groups: their intended meaning has to be described in their description fields. With the grouping mechanism one can specify which timing constraints will be used together for WCET analysis. Hierarchical definitions of groups are supported by specification of an optional list of nested groups.
Timing constraints that are *invariants* at the *program layer* are relatively easy to maintain. They can be checked directly against the source code and they only have to be changed if the program code changes. They remain valid if the computer platform changes.

C2 Program-Specific Annotations

We define program-specific annotations as timing constraints that directly describe the control and data flow of a program.

C2.1 Loop Bounds

Loop bounds comprise the minimal timing constraints at the *program layer* that are necessary to estimate the WCET of a program. For this reason, they were the first type of annotation that was introduced in the short history of WCET annotation languages [11].
Although loop bounds can always be expressed through linear flow constraints, there are practical reasons to allow loop bounds to be specified in a specialized and more compact notation. To maintain a tight execution count estimate after certain loop optimizations, it is desirable to specify lower loop bounds directly.

```
int i;
for (i = 0; i < n; ++i) {
  process(g[n]);
}
```

Here, the loop bound depends on the value of variable n. Static interprocedural program analysis over the whole program may find that the possible value of n at the beginning of the loop is 3...10, resulting in a lower loop bound of 3 and an upper loop bound of 10.

C2.2 Recursion Bounds

When a recursion is bounded, time and stack size requirements are also bounded using this recursion depth. If such conditions cannot be established by analysis, user annotations can supply the required data. In analogy to the earlier work on loop-bounds [1], Blieberger and Lieger established the conditions necessary for establishing upper bounds for stack space and time requirements of directly recursive functions. They also generalize the approach to indirectly recursive functions [2]. Recursion depth annotations are also used by Ferdinand et al. [4].

`unsigned fac(unsigned n) {`
`  if (n == 0) return 1;`
`  else return n*fac(n-1);`
`}`
	The most precise recursion bound of procedure *fac* is the maximum value of input variable n. If a static program analysis finds *fac* always to be called with $n \leq 10$, then 10 is the most precise recursion bound.

C2.3 Linear Flow Constraints

Linear flow constraints are the basis for IPET-based WCET calculation methods. In the course of the calculation, all other program-specific constraints and control-flow constraints will eventually get translated into linear flow constraints. While flow constraints have a very high expressiveness, they are not necessarily as easy to write as, e.g., loop bounds, which is one of the reasons for allowing multiple ways of annotating the same flow constraint.

Linear flow constraints are used to express a relationship between certain reference points in the control flow graph (CFG) of a program. From the perspective of the source language this necessitates the introduction of auxiliary annotations like *markers* (to obtain a reference point) and *scopes* (to restrict the lexical validity of a constraint). The constraints themselves are usually called *restrictions*.

`for (i = 0; i < n; ++i) {`
`  for (j = i; j ≥ 0; --j) {`
`    stmt1;`
`  }`
`}`
	We assume that the execution count of the entry of the outer loop is labeled as m_0 and the execution count of the inner loop's body is labeled as m_1. Then, the linear flow constraint "$m_1 \leq n \cdot (n-1)/2 \cdot m_0$" can be used to provide refined information about the execution count of the loop nest.

C2.4 Variable-Value Restrictions

Variable-value restrictions describe data-flow and are thus not a direct control-flow restriction. Variable-value restrictions can be transformed into an explicit control-flow restriction by a program analysis tool.

`if (i < 72) {`
`  stmt1;`
`  ...`
	Directly before *stmt1* the value of i is confined by $i_{min} \leq i < 72$, where i_{min} is the smallest possible value of the data type of i.

C2.5 Summaries of External Functions

Often, software libraries are distributed as binaries and without any source code. In these cases, the library manufacturer could provide summaries of the library functions that contain the missing information that is necessary to analyze programs that use the library. A summary of a function may contain side effects (list of modified items) or value ranges of the returned values. A function summary may still be useful, even when the source code is available, e.g., for hard-to-analyze facts.

`int signum(int x);`	The subroutine *signum* is assumed to be pure and returns $-1, 0$ or $+1$. Thus we can annotate that the set of objects modified by this subroutine is empty, and the value returned by the subroutine is always from within $[-1, 1]$.

C3 Addressable Units

Addressable units of an annotation language are those that can be associated with timing constraints. The more language constructs and levels of abstraction can be addressed, the more fine-grained timing constraints can be specified. Examples of how to address different units of the program layer are given in [8]. In this section we list all language constructs that we consider relevant for being annotated with timing constraints.

C3.1 Control-Flow Addressable Units

Conceptually, WCET annotations typically express relationships between nodes, edges

and paths of the CFG. If the paths between functions are included in the graph as well then we call this graph an interprocedural control flow graph (ICFG) [15]. Although the ICFG is implicitly defined by the program structure, it is not generally visible and will be generated ad hoc by the compiler. The annotation language therefore faces the problem to address entities inside a graph that have no standardized explicit representation.

We thus propose the following addressable units of the ICFG based on the program source code:

C3.1a Basic Blocks

A basic block is a code sequence with single entry and single exit point. For timing analysis it is relevant that execution passes a basic block's entry point as often as its exit point. Thus, instead of annotating the basic block, any location within the basic block can hold the block annotation.

C3.1b Edges

Edges in the CFG, however, do not necessarily have a direct counterpart in the program because they are implicitly defined by the semantics of the respective language construct.

To circumvent this problem we introduce a set of reserved edge-names for each control flow construct of the source language. For example, considering some constructs of the C language, such names could include $TrueEdge_{if}$, $FalseEdge_{if}$ and $BackEdge_{while}$. Such names allow a user to associate timing constraints with specific edges of the respective CFG for a given language construct.

C3.1c Subgraphs

Subgraphs of the ICFG can be addressed and thus annotated. For example, an annotation can be associated with an entire function, or with a statement containing several function calls, or some nested loops.

To handle control flow inside expressions, such as function calls and short-circuit evaluation, it is necessary to normalize the program first. In this step short-circuit evaluation will be lowered into nested if-statements and function calls are extracted from expressions. For the addressing of subexpressions, a mapping between the normalized code and the original code must be maintained.

C3.2 Loop Contexts

For all kinds of loops it may be of interest to annotate specific iterations separately, or to exclude specific iterations, i.e. annotate all but these specific iterations. The most prominent example is that the first (few) iteration(s) may be very different from the following ones due to cache effects.

Code	Description

`for (int i = 0; i < n; ++i)`
`  for (int j = 0; j < d; ++j)`
`    a[i][j] *= v[j];`
	Due to the "warming-up" of the cache, the first iteration could show a different behavior than the subsequent iterations.

C3.3 Call Contexts

As different call sites are bound to present different preconditions for a function e.g. input values, separate annotation of these different call contexts must be possible.

```
void g() { f(50); }

int f(int i) {
  while (--i ≥ 0) {
    ...
  }
```

The loop bound in function f depends on the value of input variable i. Thus, as a context-dependent flow constraint we can write that the upper loop bound is 50 when *f()* is directly called by *g()*.

C3.4 Values of Input Variables

If a function behaves significantly different depending on the possible values of an input

parameter, it can be useful to provide different sets of annotations for each case. This kind of annotation was first introduced with SPARK Ada [14] and was called "modes".

```
int f(struct data *x) {
  if (x == NULL)
    return NULL;
  ...
}
```

The function may behave completely different depending on whether the input variable x is $NULL$ or not: e.g. whenever $x == NULL$, the function returns immediately.

C3.5 Explicit Enumeration of (In)feasible Paths

In path-based approaches [3,**?**,**?**,**?**], explicit knowledge of the feasibility of paths can be incorporated into the analysis process.

```
void worker() {
  init();
  while (cond) process();
}
void process() {
  if (!initialized)
    init();
  ...
```

In this example, function *init()* is never called from function *process()*, if *process()* itself is called from function *worker()*. We can thus annotate that there is no path *worker*→*process*→*init*.

C3.6 The Goto Statement

The `goto` statement allows to introduce edges of non-structured control flow. If the target of a `goto` statement is statically known, it is not necessary to introduce any special annotations to specifically address a goto statement in the CFG; the containing basic block can be used equivalently. If the target address of a goto is not statically known, it makes sense to annotate possible jump targets as described in paragraph **C4.3**. The `break`, `continue` and `return` statements are specialized (better-behaved) instances of the `goto` statement in that their branch target is further restricted from function scope to the current control scope. This can be exploited by better analysis, but from the annotation standpoint there is not much difference to the `goto` except that there is less need for an annotation, when the analysis is easier.

C4 Control-Flow Constraints

The CFG is a valuable abstraction level that can be refined in various ways to improve the precision of the analysis. This is to aid the automatic CFG generation within the tools by additional information that is not available within the program itself.

C4.1 Unreachable Code

This is a program-specific annotation, which has been used by Heckmann and Ferdinand [7]. Unreachable code could as well be specified by linear flow constraints. Having a specific mechanism however makes the intention of the user explicit.

C4.2 Predicate Evaluation

Closely related to the above case, annotations of predicate evaluations were also introduced by Heckmann and Ferdinand [7]. These kind of annotations describe for conditions/decisions whether they will always evaluate to True or False.

C4.3 Control-Flow Reconstruction

Introduced by Ferdinand et al. [5], and further elaborated by Kirner and Puschner [12], the CFG Reconstruction Annotations are used as guidelines for the analysis tool to construct the control flow graph (CFG) of a program. Without these annotations it may not be possible to construct the CFG from the binary or object code of a program. On one hand, annotations are used for the construction of syntactical hierarchies within the CFG, i.e., to identify certain control-flow structures like loops or function calls. For example, a compiler might emit ordinary branch instructions instead of specific instructions for function calls or returns. In such cases it might be required to annotate a branch instruction whether it is a call or return instruction.

The high-level programming language features that can lead to code that is difficult to analyze locally are: function-pointer calls, virtual-method calls, and returns as well as

indirect conditional control-flow transfer like computed goto or switch statements or transformation results obtained from combining conditional control flow with ordinary or indirect calls or returns.

```
void process((void)(int*) func,
             int *data) {
  (*func)(data);
}
```

In this code, it might be known that the target of function pointer *func* points either to (void)reset(int*) or to (void)iterate(int*).

A work-around that sometimes helps avoiding code annotations is to match code patterns generated by a specific version of a compiler. However, such a "hack" cannot cover all situations and may also have the risk of incorrect classifications, for example, if a different version of the compiler is used.
On the other hand, annotations may be needed for the construction of the CFG itself. This may be the case for branch instructions where the address of the branch target is calculated dynamically. Of course, static program analysis may identify a precise set of potential branch targets for those cases where the branch target is calculated locally. In contrast, if the static program analysis completely fails to bind the branch target, it has to be assumed that the branch potentially branches to each instruction in the code, which obviously is too pessimistic in order to compute a useful WCET bound. In such a case, code annotations are required that describe the possible set of branch targets.

The following list summarizes examples of code annotations derived from aiT [5,7]:

- instruction <addr> calls <target-list>;
- instruction <addr> branches to <target-list>;
- instruction <addr> is a return;
- snippet <addr> is never executed;
- instruction <addr> is entered with <state>;

Note that these annotations must not be linked to a specific instruction type, since an optimizing compiler may combine or change instructions, but the annotation needs to stay.

C5 Hardware-Specific Annotations

For a realistic modeling of the execution behavior of a program, an annotation language also needs mechanisms to describe the behavior of the underlying hardware. Many of these annotations are supported by industrial timing analyzers like aiT [7].
Since some hardware-specific annotations are associated to the *hardware layer* only, they are independent from the program layer and can thus be easily reused for multiple programs running on the same embedded platform. It can thus make sense *not* to annotate this information to program code, but rather gather it in a common location so that it can be combined with the annotations of more than one program.
Examples of such basic hardware data to be kept separate from the program annotations are:

Instruction timing: The general timing information of instructions has to be maintained separate from the program.

Clock rate: The analysis must be able to convert clock ticks to absolute times when computing the WCET, and vice versa for absolute-time specification annotations.

Access times for ROM, internal and external RAM: It would be tedious and cumbersome to specify these times at each of the various read and write operations.

Memory map: As the memory map binds memory access times to a multitude of memory access operations, the information that is available to the linker can, when supplied to the timing analysis, largely reduce the annotation effort for the program.

Hardware implementation details that hold on the program as a whole, and cannot be tied to a single specific program location, also need to be specified separately. Caches or jump prediction details are examples.
It is not always obvious where to draw the borderline between hardware-specific annotations and information that is better managed by the analysis tool. The following items are examples of timing constraints that are reasonably expressed as timing annotations.

C5.1 Memory and Memory Accesses
The temporal behavior of memory accesses depends on the characteristics of the memory. Embedded systems typically use different types of memory depending on the access frequency and access pattern. It is thus necessary to specify the following characteristics:

- address range of read operations
- address range of write operations
- writeable memory area (RAM) and read-only memory area (ROM)
- data and code regions
- access time of specific memory regions (in cycles or ms)

C5.2 Absolute Time Bounds
Providing a means for absolute time bounds allows to specify the maximum and minimum execution time of a fraction of code. Such a feature can be found in wcetC [10], for example.

```
char poll() {
  volatile char io_port;
  while (io_port ≠ 0)
    /* wait */ ;
}
```

It could be an invariant of the hardware platform that the execution time of the subroutine *poll()* (busy waiting) is always between 30 and $100\mu s$.

4 Conclusion

The lack of common interfaces and open analysis frameworks is an impediment for the research in WCET analysis. Activities have been started within the ARTIST2 Network of Excellence to define such a common WCET analysis platform. As part of this, *The Annotation Language Challenge* for WCET analysis has been proposed [11]. This paper is aimed to be a first step towards a *common WCET annotation language*. It describes essential ingredients such an annotation language should include. The timing constraints are described conceptually to allow instantiation for different representation levels and tools.

We analyzed existing timing-annotation constructs and described them in a conceptual way. We identified the potential need for further mechanisms and developed some new ingredients for annotation languages. Among the new contributions are the categorization techniques of timing constraints by the separation between *timing invariants* and *timing fictions*, the introduction of *annotation layers*, *annotation groups*. Further, we gave a discussion of *addressable units* to be used for annotating the program.

We consider the proposed list of essential ingredients for a WCET annotation language as complete for procedural languages. Therefore we want to encourage professionals and researchers to provide their feedback as a basis for the refinements of this list.

Acknowledgments
We would like to thank Niklas Holsti from Tidorum Ltd for his valuable comments on this paper.

References

1. Johann Blieberger. Discrete loops and worst case performance. *Computer Languages*, 20(3):193–212, 1994.
2. Johann Blieberger. Real-time properties of indirect recursive procedures. *Inf. Comput.*, 171(2):156–182, 2001.
3. Roderick Chapman, Alan Burns, and Andy Wellings. Combining static worst-case timing analysis and program proof. *Real-Time Systems*, 11(2):145–171, 1996.
4. Christian Ferdinand, Reinhold Heckmann, Marc Langenbach, Florian Martin, Michael Schmidt, Henrik Theiling, Stephan Thesing, and Reinhard Wilhelm. Reliable and precise WCET determination for a real-life processor. In *Proc. of the 1st Int'l Workshop on Embedded Software (EMSOFT 2001)*, Tahoe City, CA, USA, Oct. 2001.
5. Christian Ferdinand, Reinhold Heckmann, and Henrik Theiling. Convenient user annotations for a WCET tool. In *Proc. 3rd International Workshop on Worst-Case Execution Time Analysis*, pages 17–20, Porto, Portugal, July 2003.
6. Jan Gustafsson. The WCET tool challenge 2006. In *Preliminary Proc. 2nd Int. IEEE Symposium on Leveraging Applications of Formal Methods, Verification and Validation*, pages 248 – 249, Paphos, Cyprus, November 2006.
7. Reinhold Heckmann and Christian Ferdinand. Combining automatic analysis and user annotations for successful worst-case execution time prediction. In *Embedded World 2005 Conference*, Nürnberg, Germany, Feb. 2005.
8. Niklas Holsti. *Bound-T Assertion Language*. Tidorum Ltd, Helsinki, Finland, 6.2 edition, Feb. 2008. online available at: `http://www.tidorum.fi/bound-t/assertion-lang.pdf`.
9. IST-004527. The ARTIST2 Network of Excellence on Embedded Systems Design. `http://www.artist-embedded.org/`, September 1st 2004 - August 31st 2008. ARTIST2 is funded by the European Commission within FP6.
10. Raimund Kirner. The programming language WCETC. Technical report, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-1, 1040 Vienna, Austria, 2002.
11. Raimund Kirner, Jens Knoop, Adrian Prantl, Markus Schordan, and Ingomar Wenzel. WCET analysis: The annotation language challenge. In *Proc. 7th International Workshop on Worst-Case Execution Time Analysis*, Pisa, Italy, July 2007.
12. Raimund Kirner and Peter Puschner. Classification of code annotations and discussion of compiler-support for worst-case execution time analysis. In *Proc. 5th International Workshop on Worst-Case Execution Time Analysis*, Palma, Spain, July 2005.
13. Object Management Group. *MDA Guide*, version 1.0.1 edition, June 2003. document number: omg/2003-06-01.
14. Chang Y. Park. Predicting program execution times by analyzing static and dynamic program paths. *Real-Time Systems*, 5(1):31–62, 1993.
15. Micha Sharir and Amir Pnueli. Two approaches to inter-procedural data-flow analysis. In Steven S. Muchnik and Neil D. Jones, editors, *Program Flow Analysis: Theory and Applications*. Prentice-Hall, 1981. ISBN:0137296819.
16. Alexander Vrchoticky. Modula/R - Language Definition. Technical report, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-1, 1040 Vienna, Austria, Mar. 1992.
17. Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckman, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenstrom. The worst-case execution time problem - overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(3), Apr. 2008.

TuBound – A Conceptually New Tool for Worst-Case Execution Time Analysis [1]

Adrian Prantl, Markus Schordan and Jens Knoop

Institute of Computer Languages,
Vienna University of Technology, Austria
email: {adrian,markus,knoop}@complang.tuwien.ac.at

Abstract TuBound is a conceptually new tool for the worst-case execution time (WCET) analysis of programs. A distinctive feature of TuBound is the seamless integration of a WCET analysis component and of a compiler in a uniform tool. TuBound enables the programmer to provide hints improving the precision of the WCET computation on the high-level program source code, while preserving the advantages of using an optimizing compiler and the accuracy of a WCET analysis performed on the low-level machine code. This way, TuBound ideally serves the needs of both the programmer and the WCET analysis by providing them the interface on the very abstraction level that is most appropriate and convenient to them.
In this paper we present the system architecture of TuBound, discuss the internal work-flow of the tool, and report on first measurements using benchmarks from Mälardalen University. TuBound has also been entered to the WCET Tool Challenge 2008.

1 Motivation

Static WCET analysis is typically implemented by the implicit path enumeration technique (IPET) [13,16] which works by searching for the longest path in the *interprocedural control flow graph (ICFG)*. This search space is described by a set of *flow constraints* (also called flow facts), which include e.g. upper bounds for loops and relative frequencies of branches. Flow constraints can generally be determined by statically analyzing the program. However, there are many cases

[1] This paper has been published in the Preliminary Proceedings of the *8th International Workshop on Worst-Case Execution Time Analysis* (*WCET'08*).
This work has been partially supported by the Austrian Science Fund (Fonds zur Förderung der wissenschaftlichen Forschung) under contract No P18925-N13, *Compiler Support for Timing Analysis*, `http://costa.tuwien.ac.at/`,
the ARTIST2 Network of Excellence, `http://www.artist-embedded.org/`
and research project "Integrating European Timing Analysis Technology" (ALL-TIMES) under contract No 215068 funded by the 7th EU R&D Framework Programme.

where a tool has to rely on *annotations* that are provided by the programmer, because of the imprecision of the analyses. Current WCET analysis tools, as they are used by the industry, therefore allow the user to annotate the machine code with flow constraints.

The central idea behind TuBound is to close the gap between source code annotations and machine-specific WCET analysis. We argue that providing high-level annotation support at the source code has several benefits:

- *Convenience and Ease*: For the user, annotating the source code is generally easier and less demanding as annotating the assembler output of the compiler.
- *Reuse and Portability*: Source code annotations, which specify hardware-independent behaviour, can directly be reused when the program is ported to another target hardware.
- *Feedback and Tuning*: Source code annotations can be used to present the results of static analyses to the programmer for inspection and further manual refinement.

In spite of these benefits gained from source code annotations, the actual longest-path search of the WCET calculation must be performed on the machine code that will be executed on the target hardware.

Compiler *optimizations*, however, represent an obstacle for using source code annotations, as they can change the control flow of the program and hence invalidate annotations. In TuBound, this is taken care of by transforming flow constraints according to the performed optimizations. Technically, this is achieved by a special component, called FlowTrans, which is a core component of TuBound and described in Section 3.2. FlowTrans performs source-to-source transformations. Therefore, our overall approach is retargetable to other WCET tools; currently we are using CalcWCET_{167}.

From the tool developer's point of view, this source-based approach offers the advantage that analyses can use high-level information that is present in the source code, but would be lost during the lowering to an intermediate representation. A typical example for such information is the differentiation between bounded array accesses and unbounded pointer dereference operations. Since the output of a source-based analysis is again annotated source code, it is also possible to create a feedback loop where the user can run the static analysis and fill in the annotations where the analysis failed to produce satisfying results. Afterwards, the analysis could be rerun with the enriched annotations to produce even tighter estimates.

TuBound is based on earlier work by Kirner [12] who formulates the correct flow constraint updates for common compiler transformations. TuBound goes beyond his approach by extending it to source-to-source transformations and by adding interprocedural analysis. Optimization traces for flow constraint transformations are also used by Engblom et al. [7]. With FlowTrans, we are taking this concept to a higher level, by performing control-flow altering transformations already at the source level. The integration of static flow analysis with

low-level WCET analysis is also implemented in the context of SWEET, which uses a technique called abstract execution to analyse loop bounds [8,9]. Again, our approach uses a higher level of abstraction by performing static analyses directly at the source code level. The interaction of compiler optimizations and the WCET of a program has been covered by Zhao et al. [20], where feedback from a WCET analysis was used to optimize the worst-case paths of a program.

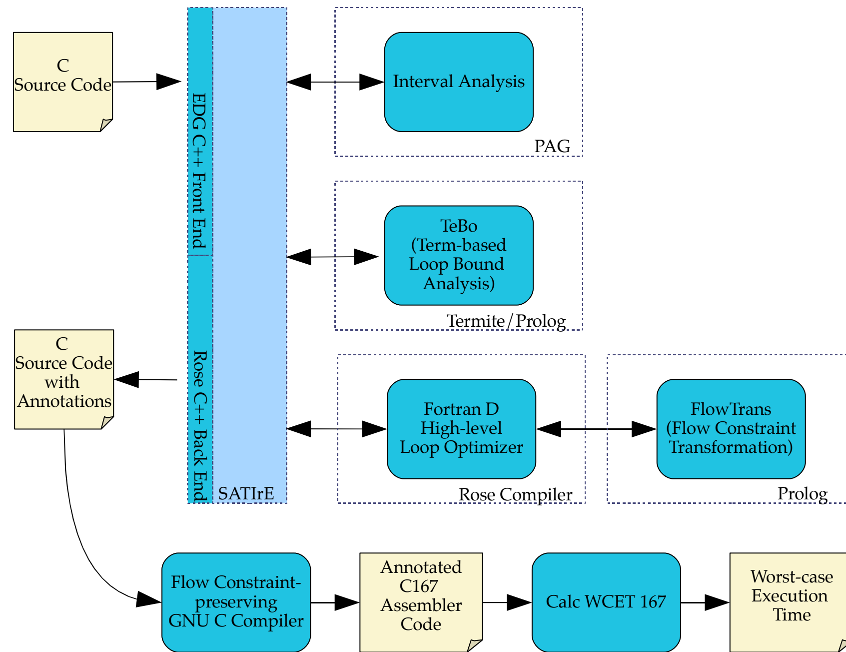

Figure 1. The collaboration of TuBound's components

2 The Architecture of TuBound

TuBound is created by integrating several components that were developed independently of each other. The majority of the components are designed to operate on the source code. This decision was motivated by gains in flexibility for both tool developer and users.

The architecture and work flow of TuBound is summarized in Figure 1. The connecting glue between the components is the Static Analysis Tool Integration Engine (SATIrE) [17,3]. SATIrE enables using data flow analyzers specified with the Program Analyzer Generator (PAG) together with the C++ infrastructure of the Rose compiler. SATIrE internally transforms programs into its own intermediate representation, which is based on an abstract syntax tree (AST). An external term representation of the AST can be exported and read by SATIrE.

This term representation is generated by a traversal of the AST and contains all information that is necessary to correctly unparse the program. This information is very fine-grained and including in particular line and column information of the respective expressions. The terms are also annotated with the results of any preceding static analysis. The key feature, however, is the syntax of the term representation. It was designed carefully to match the syntax of Prolog terms. This allows it to be manipulated as a Prolog program very easily. A similar approach of using Prolog terms to represent the AST of a program is used in the JTransformer framework for the Java language [1].

```
7  for (i = 0; i < 100; i++) {
```

```
for_statement(
  for_init_statement( [ expr_statement( assign_op(
        var_ref_exp(
          var_ref_exp_annotation(type_int,"i",0,
                               null,analysis_result(null,null)),
          file_info("triang.c",7,10)),
        int_val(null,value_annotation(0,analysis_result(null,null)),
                file_info("triang.c", 7, 12)),
        ... ], default_annotation(null, analysis_result(null,null)),
                          file_info("triang.c", 7, 3)),
    expr_statement( less_than_op(
        var_ref_exp(var_ref_exp_annotation(type_int,"i",0,null,
           ...
```

Figure 2. The external AST term representation of SATIrE

The Rose compiler [5] is a source-to-source transformation framework that includes the EDG C++ front end, a loop optimizer and a C++ unparser [18]. The loop optimizer was ported from the Fortran D compiler. In TuBound we are using the front end and the high-level loop optimizer that is part of Rose. The Program Analyzer Generator (PAG) [2] by AbsInt Angewandte Informatik GmbH allows the specification of data flow analyses in a specialised functional language. Using PAG, we implemented a variable interval analysis for TuBound. CalcWCET_{167} [4] is a tool that performs WCET analysis for the Infineon C167 micro-controller. CalcWCET_{167} expects annotated C167 assembler code as input. The tool is complemented by a customized version of the GNU C compiler that translates annotated C sources into annotated assembler code for the C167 micro-controller.

3 The Work Flow of TuBound

Conceptually, the work flow of analysing a program with TuBound comprises three stages:

3.1 Start-up and Annotation

Parsing. In the first phase, the source code of the program is parsed by the EDG C++ front end that is integrated into the Rose compiler. Rose then creates a C++ data structure of the AST and performs consistency checks to verify its integrity. The Rose loop optimizer performs analysis and transformations based on the AST data structure.

Interval Analysis. The AST is traversed by SATIrE to generate the interprocedural control flow graph (ICFG), an amalgam of call graph and *intra*procedural CFG [19]. This data structure is the interface for the PAG-based interval analysis that calculates the possible variable value ranges at all program locations. The context-sensitive interval analysis operates on a normalized representation of the source code that is generated during the creation of the ICFG. The interval analysis is formulated as an interprocedural data-flow problem and is a pre-process of the loop bounding algorithm, which is otherwise unable to analyze iteration counts that depend on variable values that stem from different calling contexts. Once the interval analysis converges to a fixed point, the results are mapped back to the AST.

Loop Bound Analysis. The next step is the loop bound analysis. This analysis operates on the external term representation of SATIrE. We exploit this fact with our term-based loop bounder (TeBo) which was written entirely in Prolog. Our loop bounding algorithm exploits several features of Prolog: To calculate loop bounds, a symbolic equation is constructed, which is then solved by a set of rules. It is thus possible for identical variables with unknown, but constant values to cancel each other out. For example, in the code `for (p = buf; p < buf+8; p++)`, the symbolic equation would be $lb = (buf + 8 - buf)/1$. The right-hand side expression can then be reduced by TeBo's term rewriting rules. The loop bounding algorithm also ensures that the iteration variable is not modified inside the loop body. This is implemented with a control flow-insensitive analysis [14] that ensures that the iteration variable does not occur at the left-hand side of an expression inside the loop body and its address is never referenced within its scope.

Original program

```
int main()
{
  int i,j;
  for (i = 0; i < 100; i++) {
    for (j = 0; j < i; j++) {
      // body
    }
  }
}
```

Annotations generated by TuBound

```
int main() {
#pragma wcet_marker(m1)
  int i;
  int j;
  for (i = 0; i < 100; i++) {
#pragma wcet_constraint(m2=<m1*100)
#pragma wcet_marker(m2)
#pragma wcet_loopbound(100)
    for (j = 0; j < i; j++) {
#pragma wcet_constraint(m3=<m_1*4950)
#pragma wcet_marker(m3)
#pragma wcet_loopbound(99)
      // body
    }
  }
  return 0;
}
```

Figure 3. Finding flow constraints with constraint logic programming

In the case of nested loops with non-quadratic iteration spaces, loop bounds alone would lead to an unnecessary overestimation of the WCET. In TeBo, we are using constraint logic programming to yield generalized flow constraints that describe the iteration space more accurately. An example is shown in Figure 3.

The nested loop in the example has a triangular iteration space, where the innermost basic block is executed $n * \frac{n-1}{2}$ times. Our analyzer finds the following equation system for this loop nest:

$$
\begin{array}{lll}
m3 & = \sum_{n=0}^{99} m3_n(\{i := n\}) & (1) \\
m3_n(env) & = n = i & (2) \\
m2 & = m1 * 100 & (3)
\end{array}
$$

The equations are constructed with the help of an *environment* that consists of the assignments of variables at the current iteration. The variable $m1$ stands for the execution count of the `main()` function, $m2$ for the count of the outer loop and $m3$ for the count of the innermost loop. Equation 1 describes the fact that the values of i as well as the iteration counts for the individual runs of the inner loop are 0..99, respectively. Equation 2 describes the generic behaviour of the inner loop, stating that its iteration count is equal to the value of n in the current environment. The last equation describes the behaviour of the outer loop. The use of constraint logic programming allows for a lightweight implementation that does not rely on additional tools. In earlier work, Healy et al. [10] are using analysis data to feed an external symbolic algebra system that solves the equation systems for loop bounds.

Eventually, the results of the loop bound analysis are inserted into the term representation as annotations of the source code. We are using the `#pragma` directive to attach annotations to basic blocks. The annotations consist of markers, scopes, loop bounds and generic constraints. Markers are used to provide unique names for each basic block, which can then be referred to by constraints. Constraints are inequalities that express relationships between the execution frequencies of basic blocks. Loop bounds are declared within a loop body and denote an upper bound for the execution count of the loop relative to the loop entry. Scopes are a mechanism to limit the area of validity of markers such that it is possible to express relationships that are local to a sub-graph of the ICFG.

3.2 Program Optimization and WCET Annotation Transformation

The FlowTrans phase is concerned with program sources that are already annotated with WCET constraints, stemming from either an earlier analysis pass, or from a human. WCET constraints describe the control flow of the program in order to reduce the search space for feasible paths. During the compilation, however, optimizations are performed that modify the control flow. Examples of addressed optimizations are loop unrolling, loop fusion and inlining, whereas constant folding and strength reduction do not affect the control flow graph. To guarantee the correctness of the annotations of the program sources, we either have to disable these unsafe optimizations and sacrifice performance or transform the annotations accordingly. To achieve the latter, we implemented FlowTrans, a transformation framework for flow constraints.

A large number of CFG-altering optimizations are loop transformations. For this reason, we based our implementation on the Fortran D loop optimizer that is part of Rose. Keeping optimizations of interest separate from the compiler, our transformation framework is very flexible and also portable to other

optimizers. The input of FlowTrans is an optimization trace (consisting of a list of all transformations the optimizer applied to the program) and a set of rules that describe the correct constraint update for each optimization. The concept of using an optimization trace can be applied to any existing compiler. The rules need to be written only once per optimization. The rules, as well as the transformation of the flow constraints are written in Prolog and operate on the term representation of the AST. As a matter of fact, the syntax used to express the flow constraints is identical to that of Prolog terms, too, thus rendering the manipulation of flow constraints very easy. Figure 4 gives an example of such a transformation. We currently implemented rules for loop blocking, loop fusion and loop unrolling. With all support predicates, the definitions of the rules range from 2 (loop fusion) to 25 (loop unrolling) lines of Prolog [15].

Original annotated program

```
int* f(int* a)
{
        int i;
#pragma wcet_marker(m_func)
        for (i = 0; i < 48; i += 1) {
#pragma wcet_loopbound(48)
#pragma wcet_marker(m_for)
          if (test(a[i])) {
#pragma wcet_marker(m_if)
            // Domain-specific knowledge
#pragma wcet_restriction(m_if =< m_for/4)

            a[i]++;
          }
        }
        return a;
}
```

After loop unrolling by factor 2

```
int *f(int *a)
{
  int i;
  for (i = 0; i <= 47; i += 2) {
#pragma wcet_marker(m_f_1_1)
#pragma wcet_loopbound(24)
    if ((test(a[i]))) {
#pragma wcet_marker(m_f_1_1_1)
#pragma wcet_restriction(
  m_f_1_1_1+m_f_1_1_2=<m_f_1_1/2)
      a[i]++;
    }
    if ((test(a[1 + i]))) {
#pragma wcet_marker(m_f_1_1_2)
#pragma wcet_restriction(
  m_f_1_1_1+m_f_1_1_2=<m_f_1_1/2)
      a[1 + i]++;
    }
  }
  return a;
}
```

Figure 4. Prolog terms everywhere: WCET constraints before and after loop unrolling

3.3 Compilation and WCET Calculation

Compilation to Assembler Code. The annotated source code resulting from the previous stage is now converted into the slightly different syntax of the WcetC-language that is expected by the compiler [11]. This compiler is a customized version of GCC 2.7.2 which can parse WcetC and guarantees the preservation of all flow constraints at the C167 machine language level. The output of the GCC is annotated assembler code.

WCET Calculation. CalcWCET_{167} reads the annotated assembler code that is produced by the GCC and generates the control flow graph of every function. CalcWCET_{167} implements the IPET method and contains timing tables for the instruction set and memory of the supported hardware configurations which are used to construct a system of inequalities describing the weighted control flow graph of each function. The weights of the edges correspond to the execution time of each basic block. This system of inequalities is then used as input for

an integer linear programming (ILP) solver that searches for the longest path through the weighted CFG. The resulting information can then be mapped back to the assembler code and can also be associated with the original source code.

4 Measurements

To demonstrate the practicality of our approach, we use a selection of benchmarks that were collected by the Real-Time Research Center at Mälardalen University [6]. For our experiments we selected those benchmarks that can be analysed by TuBound without annotating the sources manually. Figure 5 shows the time spent in the different phases of TuBound and the estimated WCET for a subset of benchmarks. It must be noted that a large part (about 45% for the `ns` benchmark) of the time spent in TeBo is currently used to parse the term representation from one and write it to another file. This bottleneck can be eliminated by directly generating the data structure via the foreign function interface of the Prolog interpreter process and thus eliminating the expensive parsing and disk I/O. In Figure 6 the influence of compiler optimizations on the WCET of the benchmarks can be seen, where the different bars per benchmark denote the analyzed WCET of the unoptimized program vs. the program with high-level and/or low-level optimizations turned on. Note that the y-axis uses a logarithmic scale. From the results, three different groups can be observed:

Group 1: cnt, crc, lcdnum, qurt

Group 2: bsort100, cover, expint, fibcall, recursion, sqrt, st, whet

Group 3: fdct, jfdctint, matmult, ns

In the first group, the calculated WCET is always lower for the loop-optimized code. In the second group, the WCET is the same, regardless of loop optimizations. In the third group, the WCET of the loop-optimized program is better than that of the unoptimized program, however, if both kinds of optimizations are enabled, they interfere and less well performing code is generated, which is reflected by the higher WCET. One reason for this is extra spill code that is generated due to higher register pressure.

5 Conclusion

TuBound is a WCET analysis tool which is unique for combining the advantage of low level WCET analysis with optimizing compilation and high level source code annotations. The flow constraint transformation framework FlowTrans ensures that annotations are transformed according to the optimization trace as provided by the high-level optimizer. This approach allows us to close the gap between source code annotations and machine-specific WCET analysis. TuBound has also been entered to the WCET Tool Challenge 2008 [6].

Acknowledgements. We would like to thank Raimund Kirner for his support in integrating his tool CalcWCET$_{167}$ and Albrecht Kadlec for many related discussions.

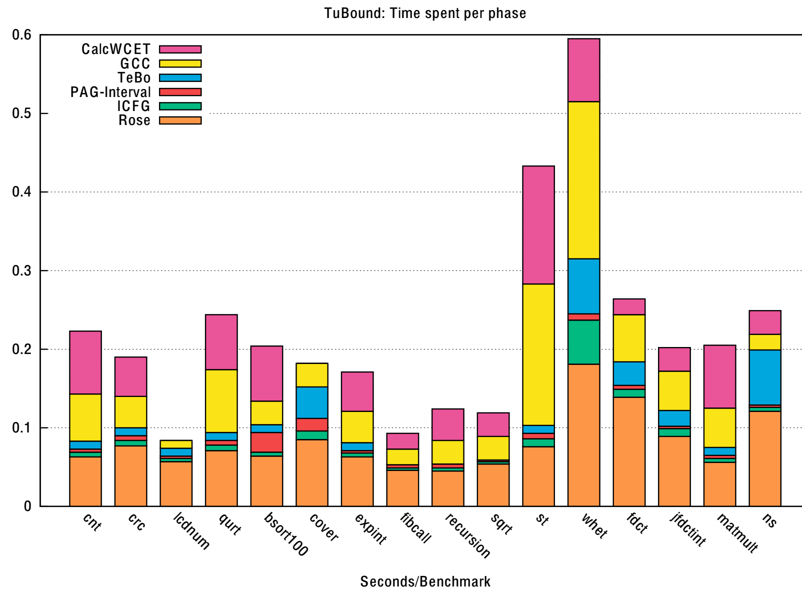

Figure 5. Analysis runtime of the selected benchmarks

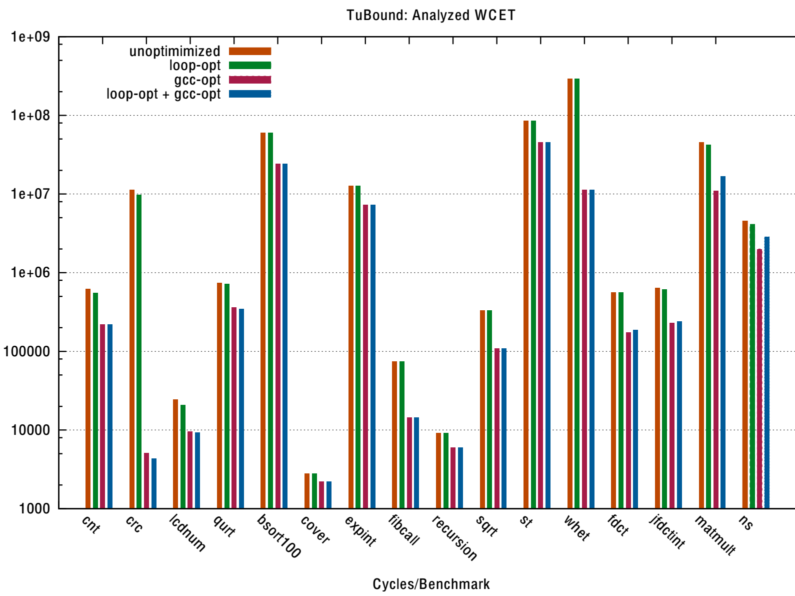

Figure 6. Analyzed WCET of the selected benchmarks

References

1. The JTransformer framework. `http://roots.iai.uni-bonn.de/research/jtransformer/`.
2. The program analyzer generator PAG. `http://www.absint.com/pag/`.
3. The SATIrE framework. `http://www.complang.tuwien.ac.at/markus/satire/`.
4. The CalcWCET$_{167}$ tool. `http://www.vmars.tuwien.ac.at/~raimund/calc_wcet/`.
5. The Rose Compiler. `http://www.rosecompiler.org/`.
6. The WCET tool challenge 2008. `http://www.mrtc.mdh.se/projects/WCC08/`.
7. J. Engblom, P. Altenbernd, and A. Ermedahl. Facilitating worst-case execution time analysis for optimized code. In *Proceedings of the 10th EuroMicro Workshop on Real-Time Systems, Berlin, Germany*, June 1998.
8. Jan Gustafsson, Andreas Ermedahl, and Björn Lisper. Towards a flow analysis for embedded system C programs. In *10th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2005)*, pages 287–297, 2005.
9. Jan Gustafsson, Andreas Ermedahl, Christer Sandberg, and Björn Lisper. Automatic derivation of loop bounds and infeasible paths for WCET analysis using abstract execution. In *27th IEEE Real-Time Systems Symposium (RTSS'06)*, 2006.
10. Christopher A. Healy, Mikael Sjodin, Viresh Rustagi, David B. Whalley, and Robert van Engelen. Supporting timing analysis by automatic bounding of loop iterations. *Real-Time Systems*, 18(2/3):129–156, 2000.
11. Raimund Kirner. The programming language wcetC. Technical report, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-1, 1040 Vienna, Austria, 2002.
12. Raimund Kirner. *Extending Optimising Compilation to Support Worst-Case Execution Time Analysis*. PhD thesis, Technische Universität Wien, Treitlstr. 3/3/182-1, 1040 Vienna, Austria, May 2003.
13. Yau-Tsun Steven Li and Sharad Malik. Performance analysis of embedded software using implicit path enumeration. In *Proceedings 32nd ACM/IEEE Design Automation Conference*, pages 456–461, June 1995.
14. Steven S. Muchnik. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.
15. Adrian Prantl. The CoSTA Transformer: Integrating Optimizing Compilation and WCET Flow Facts Transformation. In Proceedings 14. Kolloquium „Programmiersprachen und Grundlagen der Programmierung (KPS'07)". Technical Report, Universität zu Lübeck, 2007.
16. Peter Puschner and Anton V. Schedl. Computing maximum task execution times – a graph-based approach. *Journal of Real-Time Systems*, 13:67–91, 1997.
17. Markus Schordan. Combining tools and languages for static analysis and optimization of high-level abstractions. Proceedings 24th Workshop of „GI-Fachgruppe Programmiersprachen und Rechenkonzepte". Technical Report, Christian-Albrechts-Universität zu Kiel, 2007.
18. Markus Schordan and Daniel J. Quinlan. A source-to-source architecture for user-defined optimizations. In László Böszörményi and Peter Schojer, editors, *JMLC*, volume 2789 of *Lecture Notes in Computer Science*, pages 214–223. Springer, 2003.
19. Micha Sharir and Amir Pnueli. Two approaches to inter-procedural data-flow analysis. In Steven S. Muchnik and Neil D. Jones, editors, *Program Flow Analysis: Theory and Applications*. Prentice-Hall, 1981.
20. Wankang Zhao, William C. Kreahling, David B. Whalley, Christopher A. Healy, and Frank Mueller. Improving WCET by Optimizing Worst-Case Paths. In *IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 138–147, 2005.

Applying the Component Paradigm to AUTOSAR Basic Software

Dietmar Schreiner

Vienna University of Technology
Institute of Computer Languages, Compilers and Languages Group
Argentinierstrasse 8/185-1, A-1040 Vienna, Austria
{schreiner}@complang.tuwien.ac.at

1 Introduction

Current trends in embedded systems software for the automotive domain aim at an increase of reusability, exchangeability and maintainability, and thus at a significant reduction of time- and costs-to-market. One way to reach these goals is the adaption of Component Based Software Engineering (CBSE) for resource constrained embedded systems. The Automotive Open System Architecture (AUTOSAR) [1, 2], an upcoming industry standard within the automotive domain, reflects this fact by constituting CBSE as development paradigm for automotive applications: Application concerns are covered by software components, while infrastructural ones are handled within layered component middleware—the AUTOSAR Runtime Environment (RTE)[3] and the Basic Software (BSW) [4].

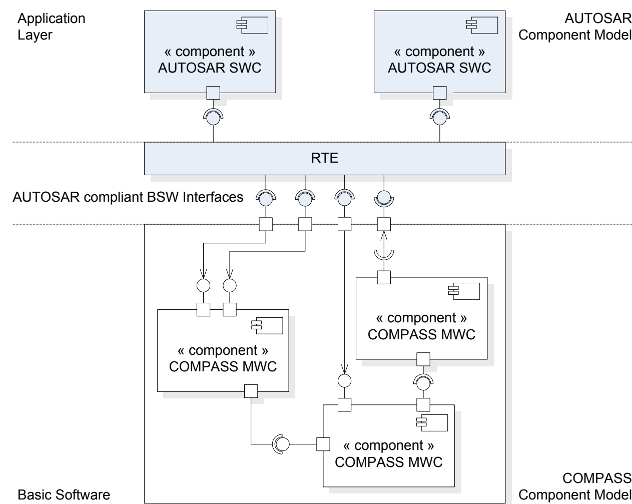

Fig. 1. Component based Basic Software

However, the AUTOSAR *Basic Software* itself is specified as layered architecture that is only customizable on a coarse-grained level, and thus tends to be heavy-weight and less flexible. Therefore, this paper contributes by applying the component paradigm to AUTOSAR *Basic Software* , to improve the capabilities of AUTOSAR compliant software systems, as conceptually depicted in Fig. 1: The redesigned BSW externally provides all interfaces to the RTE prescribed by the AUTOSAR standard, whereas the BSW's internal architecture is fully component based.

2 AUTOSAR Basic Software Components

In CBSE, the most common entity is called component. Unfortunately, literature shows many, very often contradictory [5], definitions of that term. So the first thing to do when dealing with CBSE, is to clarify the semantic meaning of this appellation, and to provide a clear vocabulary as basis for the remainder of this paper.

In accordance to the work of [6–8] we define components as follows:

> Components are trusted architectural elements of execution that interact only by their well defined interfaces, according to contractual guarantees, and strictly contain no other external dependencies. Components conform to a component model, so they adhere to a composition and interaction standard, and can be independently deployed and composed without modification. As a result, components are well suited for reuse and third-party composition. A set of well composed components is referred to as component architecture, while the term component model denotes the framework and standards, a component has to adhere to.

When building component based middleware for AUTOSAR, two things have to be taken into consideration:

- Middleware components cannot rely on component middleware. Therefore, the AUTOSAR component model [9] cannot be applied at *Basic Software* level.
- Middleware components have to provide standardized AUTOSAR functionality. Hence, the component model for middleware components has to be designed in line with the AUTOSAR standard, especially when it comes to the type system, to allow a seamless integration of component based middleware into the AUTOSAR environment.

A detailed specification of a component model, designed to meet these prerequisites within the AUTOSAR context, is provided in [10] and is used for the remainder of this paper. The so called COMPASS component model is compatible to the type system of AUTOSAR's C-language binding. It defines middleware components to be encapsulated units of execution, that interact via function calls within one global address space, and via shared memory access. The COMPASS component model specifies the *Basic Software* component classes, defines their minimal interfaces, and prescribes all means of composition and interaction for BSW components.

3 Component Recognition for AUTOSAR BSW

As our primary goal was to redesign AUTOSAR *Basic Software* in a component based way, we analyzed an exiting, layered BSW implementation to identify basic groups of functionality that could serve as base-line for BSW components: All functions, specified by AUTOSAR, which are either coupled via function calls, or which are coupled via shared memory accesses, have been marked as candidates for the same BSW component class. In addition, functions that are semantically related to each other, but are not directly coupled, have manually been assigned to the appropriate BSW component classes by domain experts. In that way we identified a set of BSW components that completely resemble the functionality and all external interfaces of the standard's layered BSW. Using the COMPASS component model and the identified *Basic Software* building blocks, it is now possible to build a component based AUTOSAR BSW, that on the one hand provides a fine-grained, function-based partitioning, enabling the creation of custom-tailored BSW, and that on the other hand highly supports reuse and exchangebility of BSW components.

3.1 Coupling within Software Components

One of the important properties of software components is that of encapsulation and separation. A well designed component contains a set of semantically related operations and data holders, that in total provide specific functionality exposed by the component's interfaces.

When trying to find those related operations and data holders within a global set of functions and data structures contained within monolithic or coarse-grained layered software, the coupling between all functions and all data structures has to be examined. For the task of component recognition two types of coupling are of interest:

Coupling via Control-Flow. Control-flow refers to the path of execution of a program. Two distinct functions within a program are strongly coupled via control-flow, if at least one of them passes control over to the other one. This is typically done by invoking the other function via a function call.

Coupling via Data-Flow. Data-flow refers to the flow of information during the execution of a program. As information is typically stored within data holders like memory cells, coupling via data-flow can be observed by examining access to data holders. Two distinct functions are strongly coupled via data-flow if both access the same data-holder, no matter if the type of access is read or write.

When taking these two types of coupling into account, two rules have to hold when performing automated component recognition:

1. Two distinct operations must not be coupled if they are contained within distinct components (horizontal coupling).
2. Two distinct operations may be coupled if they are contained within one component (vertical coupling).

3.2 Recognition Algorithm

When developing an algorithm based on static analysis, various options regarding complexity and thus execution time exist. Our algorithm was developed with respect to scalability, hence its complexity is kept linear to the size of the analyzed source code. In addition, our algorithm incorporates configuration data, especially data on late bound function pointers and on domain specific properties, to provide linear complexity and to find sufficient solutions quickly.

1. **Calculate the call graph.** A call graph, $G_C = (N_C, E_C)$, is computed from P's AST where functions of the program are represented by nodes, N_C, and calls are represented by edges, E_C, between the calling and the called function.

2. **Calculate usage graph.** A usage graph, $G_U = (N_U, E_U)$, is computed where functions and accessed data fields are represented by nodes, N_U, and the usage of a specific data field in the respective function is represented by an edge, E_U, between the function node and the data field node. To identify field accesses, the occurrence of arrow- and dot-expressions within the AST is analyzed. Our algorithm traverses the program's AST and finds all occurrences of field accesses in user defined data structures.

3. **Calculate component graph.** A component graph, $G_P = (N_P, E_P)$, can be calculated by creating a set union of the call graph and the data usage graph. It unites all gathered information on coupling via control-flow and on coupling via structure type based data field usage.

4. **Extract components from component-graph.** The algorithm's final step is the extraction of all disjoint, connected sub-graphs, the components, from the component graph. Our algorithm performs the extraction via a reachability calculation. The algorithm's output is a set of sets of nodes where each set of nodes represents one component. A domain expert can further group subsets of the automatically computed components into single components if desired.

4 Results

To prove our algorithm, it was applied to a full fledged implementation of an AUTOSAR communication stack, which is a subset of AUTOSAR *Basic Software* . The same implementation was manually decomposed as described in [11], which allows a reliable validation of the algorithm's results.

The analyzed source code is full-fledged C-code that implements the *FlexRay Interface- and Driver-Layer* as specified by the AUTOSAR standard. The source code can be characterized as shown in Table 1 and in Table 2. The generated AST contained 291794 nodes, which were traversed only once by our algorithm. The full execution of

FlexRay Interface			
	# of files	LOC	kB
Header	4	1620	59
Implementation	15	4192	135
FlexRay Driver			
Header	13	1660	88
Implementation	27	7142	222

Table 1. Source Code Characteristics

Function Definitions	107
Function Call Expr.	431
AddressOf Op.	12
Ptr Deref. Exp.	241
Arrow and dot Op.	457
Cast Expr.	4924

Table 2. Program Characteristics

our component recognition algorithm took less than 5 seconds and used approx. 80MB of memory on a 64-bit Intel PC.

The manual decomposition identified 8 components for the FlexRay Interface- and the FlexRay Driver-Layer. They were called *Base*, *Transmitter*, *Receiver*, *Time Services*, *Status*, *MTS*, *WUP*, and *TransceiverDrv*.

To perform our automatic component recognition, a set of 87 functions has been marked as relevant. 12 data structures respectively their data fields have been marked as irrelevant. On that basis our algorithm was able to recognize not only one but a set of valid decompositions. This is due to the fact, that some of the manually defined components consist of multiple, uncoupled sub-components that may be combined randomly, as they do not interfere. However, the manually created decomposition was contained within the calculated set, proving our algorithm valid in terms of imposed requirements.

5 Acknowledgments

This work has been partially funded by the European Community under the FP7 IST project All-Times, contract 215068.

References

1. Heinecke, H.: AUTomotive Open System ARchitecture An Industry-Wide Initiative to Manage the Complexity of Emerging Automotive E/E-Architectures. In: Proceedings of the Convergence International Congress & Exposition On Transportation Electronics, Detroit, MI, USA (2004)
2. AUTOSAR: Automotive Open System Architecture. `http://www.autosar.org/`.

3. AUTOSAR GbR: Specification of RTE Software 1.0.1. http://www.autosar.org/download/AUTOSAR_SWS_RTE.pdf.
4. AUTOSAR GbR: Layered Software Architecture 2.0.0. http://www.autosar.org/download/AUTOSAR_LayeredSoftwareArchitecture.pdf.
5. Broy, M.: A uniform mathematical concept of a component (appendix to M. broy et al: "What characterizes a (software) component?". Software - Concepts and Tools **19**(1) (1998) 57–59
6. Szyperski, C.: Component Software: Beyond Object-Oriented Programming. Addison-Wesley (January 1998)
7. Heineman, G.T., Councill, W.T., eds.: Component-Based Software Engineering. Addison Wesley (2001)
8. Meyer, B.: The grand challenge of trusted components. In: ICSE. (2003) 660–667
9. AUTOSAR GbR: Software Component Template 2.0.1. http://www.autosar.org/download/AUTOSAR_SoftwareComponentTemplate.pdf.
10. Schreiner, D., Göschka, K.M.: A component model for the AUTOSAR Virtual Function Bus. In: COMPSAC '07: Proceedings of the 31st Annual International Computer Software and Applications Conference. Volume 2., IEEE (2007) 635–641
11. Galla, T.M., Schreiner, D., Forster, W., Kutschera, C., Göschka, K.M., Horauer, M.: Refactoring an automotive embedded software stack using the component-based paradigm. In: Proceedings of the IEEE Second International Symposium on Industrial Embedded Systems (SIES 2007). IEEE, IEEE (Jan 2007) 200–208

An Example of Source-To-Source Analysis with SATIrE

Markus Schordan

Vienna University of Technology, Austria
markus@complang.tuwien.ac.at

Abstract. The design of the *Static Analysis Tool Integration Engine* (SATIrE [5]) allows to map source code annotations to its intermediate program representation as well as generating source code annotations from analysis results that are attached to the intermediate representation. This enables numerous applications such as automatic annotation of interfaces, testing of analyses by checking the results of an analysis against provided annotations, domain aware analysis by utilizing domain-specific program annotations, and making analysis results persistent as annotations in source code.
This concept is supported by a plug-in mechanism which allows to add user-defined analyses. Based on the annotation mechanism, users can view the results of their analyses as annotations in a given program, can test the analyzer by providing expected analysis results as annotations in source code and have them checked by SATIrE, or combine different analyses by accessing analysis information computed by a previous analyzer run. In this paper we present the approach of source-to-source analysis and show in a detailed example analysis how we support this approach in SATIrE.
The technical challenges are the design of the analysis information annotation language, the bidirectional propagation of the analysis information through different phases of the internal translation processes, and the combination of the different analyses through the plug-in mechanism. In its current version SATIrE targets C/C++ programs.

1 Introduction

Source-To-Source analysis allows to analyze programs and generate the analysis results as annotations. Furthermore analysis results computed in a previous run of the analyzer can be read in and are mapped such that they can be reused in subsequent analysis phases.
A well-known example is the analysis of a library where we determine for each function whether it has side effects. This information is then made available as annotation of the interface where the function is accessible. Source-to-source analysis starts to show its full potential if we not only consider properties at the function level, but also flow-sensitive and context-sensitive information. In order to annotate programs at this level of detail, we must reason about associating information with a certain position in a program dependent on the flow and

context of the program. This is complicated by the fact that we need to map information obtained from the annotated source code through multiple levels of intermediate representations. The multiple levels of intermediate representation become necessary with the separation of different kinds of analysis information, such as control flow and data flow information, as well as calling contexts for inter-procedural analysis.
For source-to-source analysis we demand the following properties

- The analysis information is associated with locations in the original code.
- Modular analysis must be possible with the generated analysis information.
- No internal information of an analyzer is exposed in the analysis results.

That the analysis information is associated with the original source code is the basis for presenting analysis information such that developers can use the results with appropriate tools for better understanding, verifying, and debugging their programs. The analysis of separate modules allows to make analysis results persistent and to reuse analysis results at a later analysis stage. One example application is the analysis of a library and reusing this information when analyzing an application that uses the library. This supports whole-program analysis. The third requirement, to not expose internal information of the analyzer is a requirement for tool interoperability and an essential property of source-to-source analysis. Otherwise it would mean to simply dump the internal structure and the computed analysis information - instead we want to ensure that the computed analysis information is made consistent with the original source code and analysis results are presented as annotations of the original program. This problem becomes challenging, when the analysis information contains references to program locations. Here we need to deal with mappings between the internal representation, necessary normalizations of the program code, and the mapping of analysis information such that it can be associated with the original input source code - independent of a possibly normalized internal program representation on which the analysis was performed.
A source-to-source analyzer has in common with a compiler that it normalizes or lowers program code before performing a program analysis. The reason is that the reduced number of different language constructs allows for a more compact analysis specification because only a reasonable small number of different language constructs must be explicitly addressed in the analysis. Compilers usually continue to generate machine code either for a concrete hardware platform or for a virtual machine. In contrast, a source-to-source analyzer propagates analysis information back to the original source code representation through all intermediate levels and generates the analysis information associated with the original source code.

2 Normalization

The normalization steps performed for source-to-source analysis can all be expressed as transformations with a one-to-one mapping of transformed language

constructs such that the inverse operation is possible. This allows to guarantee that information can be propagated back to the original source code. Normalization can mean to lower the code to a low-level code similar to quadruple code for machine generation, i.e. programs with the only control constructs if and goto. Expressions can be normalized to have at most one operator of each side of an assignment. However, if an analysis is designed to operate on specific patterns in expressions, this can complicate the specification instead of simplifying it. Therefore, to achieve a good trade-off between normalization level and the problems needed to be addressed in the analysis specification, we consider an analysis specification being specific to some normalization level.

Normalizing the program and specifying the analysis for the selected normalization level is straightforward. The interesting question is how the analysis information needs to be transformed to be consistent at each normalization level. In particular, since we not only generate analysis information, but also read it in we need to be able to map it in both directions through different normalization levels. If this is achieved, we can perform source-to-source analysis.

3 Intermediate Levels in SATIrE

In SATIrE we presently have two distinct different forms of intermediate representations (IR), a Tree-IR and a Graph-IR. The Tree-IR is a decorated abstract syntax tree. The Graph-IR is an inter procedural control flow graph where each node is a specific Graph-IR node. The Tree-IR is generated by the Front End, whereas the Graph-IR is generated from the Tree-IR to make control flow explicit. Graph-IR nodes may hold references to Tree-IR nodes if they are representing statements. An analyzer can operate on either one of the two representations, where for flow-insensitive analyses or metrics it is often sufficient to operate on the Tree-IR, while flow-sensitive fixpoint analyses operate on the Graph-IR.

Normalization levels are defined as source-to-source transformations accompanied with a mapping function that defines how analysis information associated with program locations is mapped between these levels.

3.1 Mapping Analysis Information

After an analysis the analysis information is propagated back through the intermediate levels to the source code level. This requires to map any entity of the analysis information that is related to the lower intermediate level to an entity of the higher level. For example, if reaching definitions are computed on an intermediate level the locations of the variable definitions of the lower intermediate level must be mapped to labels of the higher level, and ultimately to labels associated with locations in the source code. The example shown in Section 5 provides details on how this mapping is currently performed in SATIrE.

4 Analysis Information Annotation Language

The annotation format for exchanging analysis information between different analyses is a general format with only a few basic data types. The language is general enough to represent analysis information specified in PAG's [4] SETLA language. Since PAG is integrated into SATIrE this is a requirement for reusing any analysis information computed with a PAG generated analyzer. It consists of sets, lists, tuples, maps, and some primitive data types. Additionally each analysis information has an Analysis Identifier. This identifier allows to associate semantics with the analysis information.

An analysis information annotation consists of

Analysis Identifier. Allows to identify the analysis information and associate the data with semantics.

Location. Each analysis information is associated with a program location and its corresponding program fragment by a label. The labeling is computed on the Tree-IR as well as for the Graph-IR and a mapping between the two is maintained.

Data Specifier. A Data Specifier allows to define whether an analysis information is a pre or post information in a flow-sensitive information. For context-sensitive analyses it can also specify the context (e.g. the call string).

Data Element. The data element represents the analysis information computed at that point.

A data element can consist of a set, list, tuple, map, or a primitive type such as string or a number. Two special symbols exist for the top and bottom element of lattices. Sets, lists, and maps can only contain data elements of the same type, whereas tuples can contain elements of different type. In a data element the types of data can be arbitrarily nested and combined.

SATIrE offers some special categories of program information which is precomputed for the entire program: VariableIds, ExpressionIds, and Labels representing locations in a program. If VariableId is used in an analysis then a map from VariableIds to Variables is provided. The map contains the VariableId, a scope specifier, and the label of the respective variable declaration. VariableIds are a subset of ExpressionIds. In SATIrE arbitrary expressions are supported as well, but the integration of arbitrary expression mappings in the annotation format is ongoing work.

A label is defined by a mapping function that generates the labels as part of the annotations. A label associates an analysis information with some program fragment in the annotations, but can be used in the analysis information as well. This allows to refer to certain positions in a program, such as reaching definitions where labels correspond to statements in the program.

5 Example

The generation of annotations for a given program can be done in two different forms. Either the annotations are added to a given program or in a separate file. Here we show an example with annotations added to the input program.
In Fig. 2 the result of a context-insensitive reaching definition analysis is shown. The labels represent locations in the source program, which are identified by the label field in the annotations. Pre/post denotes the pre and post information computed for the respective statement. The Graph-IR for one line of the source code, the statement b=inc(b)+inc(b), is shown in Fig. 1. The analysis information that must be changed when mapping between the source-level and the IR-level is marked by being underlined. In this example the information of $(b, 15)$, computed at the Graph-IR node with label 15, must be mapped to $(b, 20)$ at the source level to be consistent with other information at the source level. The same holds for $(x, 4)$, computed at the IR-Graph level, which must be mapped to $(x, 3)$ at the source level due to normalization of function parameter binding.
In Fig. 2 some items are crossed out. These items are not included in the analysis information at the source level, but are members of the analysis information that is mapped from the respective positions in the Graph-IR. Since this information is computed for variables introduced in the Graph-IR only, there is no variable that corresponds to these variables in the Tree-IR and at the source level. Therefore this information is not included in the analysis information at the source level.
If an annotated source program is read in, the annotations need to be processed and mapped to the corresponding locations in the IR. The internal mapping mechanism provides information for processing the annotation information. The annotation information is mapped to the IR such that it can be used in a subsequent analysis as if the analysis had been performed on the IR.
Some items of the analysis information may need to be remapped to be consistent with the other information at the IR level. Currently we invalidate such items and rerun the analysis to recompute those invalidated elements. All other information can be reused and does not need to be recomputed. For the reaching definition analysis discussed here, this requires that information containing the label 20 is invalidated because the statement with level 20 contains a function call, which generates information specific to locations in the normalized IR. Thus item $(b, 20)$ must be invalidated - effectivly meaning that the item is removed from the analysis information set. Then the analyzer is invoked on the Graph-IR where each node is initialized with the processed analysis information extracted from the annotations. Thus, the analyzer is reinvoked, but only recomputes information for nodes which are generated by the normalization step. The overhead is small, as information is already available at the statement level and only needs to be recomputed for nodes that have been added by the normalization step. The recomputation of the invalidated items can have higher impact and further experience with the presented analysis method and mapping techniques is necessary to evaluate the performance impact on large programs. The goal is to

```
pre  :20 {(b,15),(a,25),(a,14),(b,24),(c,-1),(c,11)}
label:20 ArgumentAssignment(VarRefExp("$arg_0"), VarRefExp("b"))
post :20 {(b,15),(a,25),(a,14),($arg_0,20),(b,24),(c,-1),(c,11)}
pre  :22 {(b,15),(a,25),(a,14),($arg_0,20),(b,24),(c,-1),(c,11)}
label:22 FunctionCall("inc", [VariableSymbol("$arg_0")])
post :22 local_edge: bottom
post :22 call_edge : {(b,15),(a,25),(a,14),($arg_0,20),(b,24),(c,-1),(c,11)}
pre  :23 {(a,14),(a,25),(b,15),($retvar,5),(b,24),(c,-1),(c,11)}
label:23 FunctionReturn("inc", [VariableSymbol("$arg_0")])
post :23 {(a,14),(a,25),(b,15),($retvar,5),(b,24),(c,-1),(c,11)}
pre  :21 {(a,14),(a,25),(b,15),($retvar,5),(b,24),(c,-1),(c,11)}
label:21 ReturnAssignment(VariableSymbol("$inc$return_1"),
                          VariableSymbol("$retvar"))
post :21 {(a,14),(a,25),(b,15),(b,24),($inc$return_1,21),(c,-1),(c,11)}
pre  :16 {(a,14),(a,25),(b,15),(b,24),($inc$return_1,21),(c,-1),(c,11)}
label:16 ArgumentAssignment(VarRefExp("$arg_0"), VarRefExp("b"))
post :16 {(b,15),(a,25),(a,14),(b,24),($arg_0,16),($inc$return_1,21),(c,-1),(c,11)}
pre  :18 {(b,15),(a,25),(a,14),(b,24),($arg_0,16),($inc$return_1,21),(c,-1),(c,11)}
label:18 FunctionCall("inc", [VariableSymbol("$arg_0")])
post :18 local_edge: bottom
post :18 call_edge: {(b,15),(a,25),(a,14),(b,24),($arg_0,16),($inc$return_1,21),
                    (c,-1),(c,11)}
pre  :19 {(a,14),(a,25),(b,15),(b,24),($inc$return_1,21),($retvar,5),(c,-1),(c,11)}
label:19 FunctionReturn("inc", [VariableSymbol("$arg_0")])
post :19 {(a,14),(a,25),(b,15),(b,24),($inc$return_1,21),($retvar,5),(c,-1),(c,11)}
pre  :17 {(a,14),(a,25),(b,15),(b,24),($inc$return_1,21),($retvar,5),(c,-1),(c,11)}
label:17 ReturnAssignment(VariableSymbol("$inc$return_0"),
                          VariableSymbol("$retvar"))
post :17 {(b,15),(a,25),(a,14),(b,24),($inc$return_1,21), ($inc$return_0,17),(c,-1),
         (c,11)}
pre  :15 {(b,15),(a,25),(a,14),(b,24),($inc$return_1,21), ($inc$return_0,17),(c,-1),
         (c,11)}
label:15 ExprStatement(AssignOp(VarRefExp("b"),
                                AddOp(VarRefExp("$inc$return_0"),
                                      VarRefExp("$inc$return_1"))))
post :15 {(b,15),(a,14),(a,25),(c,-1),(c,11)}
```

Fig. 1. Example: Graph-IR fragment representing the statement b=inc(b)+inc(b) in normalized form. The underlined items are transformed when mapping the information to the source code level. The boxes show which information is mapped to source code locations (see Fig. 2). Variables are represented by identifiers in the Graph-IR, but for better readability the names are shown in this figure. Pre/post denotes the flow-sensitive analysis information.

```
#pragma RD 3 pre info : {(a,14),(a,25),(b,15),(b,24),(c,-1),(c,11),($inc$return_1,21),
                         ($arg_0,16),($arg_0,20)}
int inc(int x)
#pragma RD 3 post info : {(a,14),(a,25),(b,15),(b,24),(x,3),(c,-1),(c,11),($inc$return_1,21)}
{
  #pragma RD 5 pre info : {(a,14),(a,25),(b,15),(b,24),(x,3),(c,-1),(c,11),($inc$return_1,21)}
  return x + 1;
  #pragma RD 5 post info: {(a,14),(b,15),(a,25),(b,24),(x,3),(c,-1),(c,11),($inc$return_1,21),
                           ($retvar,5),}
}

#pragma RD 2 pre info : {}
int main()
{
  #pragma RD 28 pre info : {}
  int a;
  #pragma RD 28 post info: {(a,-1)}
  #pragma RD 27 pre info : {(a,-1)}
  int b;
  #pragma RD 27 post info: {(b,-1),(a,-1)}
  #pragma RD 26 pre info : {(b,-1),(a,-1)}
  int c;
  #pragma RD 26 post info: {(b,-1),(a,-1),(c,-1)}
  #pragma RD 25 pre info : {(b,-1),(a,-1),(c,-1)}
  a = 3;
  #pragma RD 25 post info: {(b,-1),(a,25),(c,-1)}
  #pragma RD 24 pre info : {(b,-1),(a,25),(c,-1)}
  b = a;
  #pragma RD 24 post info: {(b,24),(a,25),(c,-1)}
  #pragma RD 10 pre info : {(b,20),(a,25),(a,14),(b,24),(c,-1),(c,11)}
  while(a < 10) {
    #pragma RD 13 pre info : {(b,20),(a,25),(a,14),(b,24),(c,-1),(c,11)}
    if (a < b) {
      #pragma RD 14 pre info : {(b,20),(a,25),(a,14),(b,24),(c,-1),(c,11)}
      a = (a + 1);
      #pragma RD 14 post info: {(a,14),(b,20),(b,24),(c,-1),(c,11)}
    }
    else {
      #pragma RD 20 pre info : {(b,20),(a,25),(a,14),(b,24),(c,-1),(c,11)}
      b = (inc(b) + inc(b));
      #pragma RD 20 post info: {(b,20),(a,14),(a,25),(c,-1),(c,11)}
    }
    #pragma RD 13 post info: {(a,25),(a,14),(b,20),(b,24),(c,-1),(c,11)}
    #pragma RD 11 pre info : {(b,20),(a,14),(a,25),(b,24),(c,-1),(c,11)}
    c = (a + b);
    #pragma RD 11 post info: {(b,20),(a,25),(a,14),(b,24),(c,11)}
  }
  #pragma RD 10 post info: {(b,20),(a,25),(a,14),(b,24),(c,-1),(c,11)}
  #pragma RD 8 pre info : {(b,20),(a,25),(a,14),(b,24),(c,-1),(c,11)}
  a = c;
  #pragma RD 8 post info: {(b,20),(b,24),(a,8),(c,-1),(c,11)}
  #pragma RD 7 pre info : {(b,20),(b,24),(a,8),(c,-1),(c,11)}
  return 0;
  #pragma RD 7 post info: {(b,20),(b,24),(c,-1),($retvar,7),(a,8),(c,11)}
}
```

Fig. 2. Example: Annotated program with reaching definition information being made consistent with associated labels of the original source code. Crossed out items are related to temporary variables of the IR and not included in the source code annotation. The underlined items have been transformed (also see Fig. 1) to be consistent at the source code level.

make this mechanism general and independent of the semantics of the transfer functions of an analysis by analyzing the use of special data types such as Label, VariableId, and ExpressionId, in the data specification of an analysis. Currently we require user-defined transformers for the mapping mechanism of an analysis.

6 Related Work

Harrold and Rothermel present a technique for separate analysis of modules [3]. The work focuses on one particular analysis, inter-procedural may alias analysis, but the design of the analyzer is general and similar to our setting. For inter-procedural analysis an inter procedural control flow graph (ICFG) is created. The separation in control flow and intermediate representation of statements and expressions is the same as in our approach. The analysis is a modular analysis, meaning that a module is a set of interacting procedures or a single procedure that has a single entry point. The approach allows to reuse the analysis results after analyzing a module and thus, is applicable to large scale software and real world applications. In our approach we can add analysis results as annotations to source-code, allowing to reuse analysis results in a subsequent analysis step. This can either be done on the IR-level or the annotated source code is read in again.
An approach for user-defined checks that are performed by a compiler is presented in [6]. User-defined checks may increase the confidence of a programmer with respect to his code, especially if used on a continuous basis during development. The checks that can be expressed can refer simultaneously to syntax, semantics, control flow, and data flow. The tool Condate allows more semantic-enabled and user-centric compilers, obtained by fusing together compilation with other powerful analyzers. By using SATIrE such tools can be built by using the annotation mechanism and checking manual annotations with an appropriate analysis.
For optimizing compilers the automatic generation of data flow analyses and optimizations out of concise specifications has been a trend for several years. The systems of [1, 2] concentrate on “classical” inter-procedural optimizations, whereas the system of [7] is particularly well suited for local transformations based on data dependency information. We integrated PAG because it is a tool that allows to generate analyzers from specifications for similar analysis problems.

7 Conclusion

Source-To-Source analysis allows to generate analysis results as source code annotations, but also to read them in and map them to the internal representation. The corner stones of source-to-source analysis can be summarized as follows

- Analysis results are generated as annotations.
  - Annotations can be read and mapped to IR (Front End)

- Annotations can be generated from IR (Back End)

– Input source code differs to output source code only in annotations.
– Annotations are generated at labeled source code locations only.

Source-to-source analysis allows to make analysis results independent of a respective tool and permits exchange of analysis results between different analysis tools, enabling the interoperability of program analysis tools. It also supports whole-program analysis by making analysis results persistent as source code annotations, allowing to reuse analysis results in subsequent analysis phases.

Acknowledgements. This work has been partially supported by the ARTIST2 Network of Excellence (http://www.artist-embedded.org/), and the research project "Integrating European Timing Analysis Technology" (ALL-TIMES) under contract No 215068 funded by the 7th EU R&D Framework Programme.

References

1. U. Aßmann. How to uniformly specify program analysis and transformation with graph rewrite systems. In *Proceedings of the 6th International Conference on Compiler Construction* (*CC'96*) (*Linköping, Sweden*), Lecture Notes in Computer Science, vol. 1060, pages 121 – 135. Springer-Verlag, Heidelberg, Germany, 1996.
2. U. Aßmann. On edge addition rewrite systems and their relevance to program analysis. In *Proceedings of the 5th International Workshop on Graph Grammars and Their Application to Computer Science* (*GGTA'94*) (*Williamsburg*), Lecture Notes in Computer Science, vol. 1073, pages 321 – 335. Springer-Verlag, Heidelberg, Germany, 1996.
3. M. J. Harrold and G. Rothermel. Separate computation of alias information for reuse. *IEEE Trans. Softw. Eng.*, 22(7):442–460, 1996.
4. F. Martin. PAG – an efficient program analyzer generator. *International Journal on Software Tools for Technology Transfer*, 2(1):46–67, 1998.
5. SATIrE. http://www.complang.tuwien.ac.at/markus/satire. Static Analysis Tool Integration Engine.
6. N. Volanschi. Condate: a proto-language at the confluence between checking and compiling. In *PPDP '06: Proceedings of the 8th ACM SIGPLAN symposium on Principles and practice of declarative programming*, pages 225–236, New York, NY, USA, 2006. ACM Press.
7. D. Whitfield and M. L. Soffa. An approach for exploring code-improving transformations. *ACM Transactions on Programming Languages and Systems*, 19(6):1053 – 1084, 1997.

Fundamente der Programmierung

Hermann von Issendorff
Institut für Netzwerkprogrammierung
Hauptstr. 40, D-21745 Hemmoor

Zusammenfassung

Bereits 1977 kritisierte John Backus in seiner berühmten Turing-Award-Lecture die Ineffektivität der John von Neumann zugeschriebenen Rechnerstruktur mit den Worten

> "Surely there must be a less primitive way of making big changes in the store than by pushing vast numbers of words back and forth through the von Neumann bottleneck. Not only is this tube a literal bottleneck for the data traffic of a problem, but, more importantly, it is an intellectual bottleneck that has kept us tied to word-at-a-time thinking instead of encouraging us to think in terms of the larger conceptual units of the task at hand. Thus programming is basically planning and detailing the enormous traffic of words through the von Neumann bottleneck, and much of that traffic concerns not significant data itself, but where to find it."

Seither hat es zwar verschiedene beachtliche Verbesserungen gegeben, z.B. durch den Cache-Speicher, im Prinzip hat sich aber an der Rechnerstruktur nichts geändert. Nach wie vor erfolgt die Ausführung von Programmen schrittweise unter ständigem Speichern und Lesen von Zwischenergebnissen statt. Ein Grund hierfür ist, dass ein Abweichen von dem von-Neumann-Schema zu komplexen Strukturen führt, für die es bisher keine formalen Beschreibungen gibt und die deshalb nicht beherrschbar sind.

Dieser Vortrag stellt ein Verfahren vor, mit dem sich komplexe beliebige diskrete physikalische Strukturen beschreiben lassen und das unter anderen ermöglicht, das Flaschenhalsproblem zu beseitigen. Das Verfahren beruht auf drei wesentlichen Entdeckungen, die zusammengefasst auf eine Programmiersprache führen, die räumliche Strukturen beschreibt, die zeitlich verschiedene Zustände annehmen können.

Die drei Entdeckungen sind:

1. Durch Abstraktion von Metrik und Funktionalität kann jedes diskrete natürliche System auf ein topologisches Netz von Knoten von bis zu drei Dimensionen reduziert werden.
2. Durch bijektive und bikontinuierlche Abbildung kann das Knotennetz in eine gerichtete planare und danach lineare Struktur umgeformt werden.
3. Die gerichtete lineare Struktur kann programmiersprachlich beschrieben werden.

Die Kombination der drei Entdeckungen ergibt eine Akton-Algebra genannte Programmiersprache, mit der alle diskreten natürlichen Systeme beschrieben werden können. Versieht man die Knoten mit Funktionalität, dann erhält man eine Programmiersprache, mit der sich Maschinenaktivitäten beschreiben lassen, z.B. die digitaler Rechner. Durch Komposition von Funktionalität lässt sich Akton-Algebra aber auch auf die Ebene jeder klassischen Programmiersprache anheben. Versieht man die Knoten mit Metrik oder physikalischen Kräften, dann erhält man eine Programmiersprache, mit der sich die Form und die Grösse materieller Systeme beschreiben lassen, z.B. das Layout von elektronischen Schaltungen oder die Struktur biologischer Moleküle.

Über die Entstehung der Akton-Algebra ist in den vergangenen Jahren an diesem Ort wiederholt berichtet worden. Sie hat jetzt mit ihrer formalen Definition ihren endgültigen Zustand erreicht. Auf dieser wird auch das wesentliche Gewicht des Vortrags liegen.

Vortragsanmeldung, Bad Honnef 2008

Demonstrably Correct Compilation of Java Bytecode

Michael Leuschel

Institut für Informatik, Universität Düsseldorf
Universitätsstr. 1, D-40225 Düsseldorf
leuschel@cs.uni-duesseldorf.de

Ensuring the correctness of the compilation process is an important consideration in the construction of reliable software. If the compiler generates code that is not faithful to the original program code of a system, then all efforts spent on proving the correctness of the system could be futile. Proving that target code is correct w.r.t. the program source is especially important for high assurance systems

In earlier work, AWE together with Susan Stepney and the company Logica have developed the DeCCo compiler which translates a Pascal-like high-level language (called PASP) into machine code for the ASP processor. This undertaking was a Herculean task, and is arguably a big step towards one of the Grand Challanges of computer science, the Verifying Compiler.

We have investigated using a DeCCo style approach for Java Bytecode rather than PASP, to provide a reusable, demonstrably correct compiler backend. We have also moved from using Z to B. This allows us to replace the hand proofs by mechanical proofs, and also allows formal code generation, as well as powerful tool support in form of animation and model checking. A small development of a compiler from simplified Bytecode to a simplified RISC architecture, has proven the value of these tools (each finding different bugs), and has also shown the promise of the approach. While a number of research advances will certainly be required to bring such an ambitious project to completion, the fact that we start from intermediate code leads us to believe that the overall goal can be achieved within the lifetime of a research project.

Modellreduktionstechniken für symbolische Kellersysteme

Dirk Richter (richterd@informatik.uni-halle.de),
Martin-Luther-Universität Halle-Wittenberg

Abstract: Zur Software-Modell-Prüfung sowie zum Modell basiertem Testen als auch bei der Codegenerierung sind die Größe und Komplexität von Modellen entscheidende Einflussfaktoren. Ich untersuche Modelle in Form von symbolischen Kellersystemen, da diese in der Lage sind, Rekursion exakt nachzubilden. Für diese symbolischen Kellersysteme habe ich verschiedene Modellanalysen und Modellreduktionstechniken in meinem Tool **HalSPSI** implementiert, welche die Software-Modell-Prüfung in meinen Tests für den Modellprüfer Moped erheblich beschleunigen bzw. die Modellprüfung erst ermöglichen oder sogar erübrigen. Letzteres ist z.B. dann der Fall, wenn durch statische Modellanalysen meines Tool **HalSPSI** die nicht Erreichbarkeit von Fehlerkonfigurationen nachgewiesen werden kann.

Schlüsselworte: Kellersystem, Modellanalyse, Remopla, Moped, Software-Modell-Prüfung

1 Einleitung

Symbolische Kellersysteme können mittels JMoped [1] aus Java Bytecode gewonnen und mittels des Modellprüfers Moped [2, 3, 1] überprüft werden. Unter Verwendung des Cross-Compilers Grasshopper (http://dev.mainsoft.com) ist man aber nicht nur in der Lage, Java 1.6 Bytecode dafür zu verwenden, sondern auch Microsoft Intermediate Language (MSIL bzw. CIL). Es ist auch prinzipiell möglich, die Gültigkeit von Java Modeling Language (JML) [4] Annotationen zu überprüfen, wenngleich sich dies in der Praxis als unhandlich herausstellt.

In [5] wurde bereits die Technik Slicing auf Konfigurationenebene für symbolische Kellersysteme vorgestellt. Dort anschließend sollen hier im Folgendem weitere Techniken erläutert werden. Unter anderem zeige ich, dass verschiedene Modellanalysen überraschenderweise im Gegensatz zur Anwendung bei herkömmlichen Programmiersprachen plötzlich **entscheidbar** werden. Im Gegensatz zu verbreiteten 'Finite-State' Modellprüfern wie BLAST [6], SPIN [7], NuSMV/SMV (http://www.cs.cmu.edu/ modelcheck), JavaPathFinder [8], Zing [9] oder Bogor (Bandera Projekt) [10] können hier Modellanalysen interprozedural durchgeführt werden und verbessern damit natürlich das Analyseergebnis. Auch ist es nicht nötig, sich auf eine konstante maximale Anzahl an Methodenaufrufen zu beschränken und dadurch eine exponentielle Modellvergrößerung zu riskieren.

2 Begriffe

$M = (S, \rightarrow, L_A)$ heißt **Kripkestruktur**, falls S und A (nicht notwendigerweise endliche) Mengen sind, $\rightarrow \subseteq S \times S$ und $L_A : S \rightarrow 2^A$. Bei gegebener Kripkestruktur M ist das **Er-**

reichbarkeitsproblem die Frage, ob es in M von einem Zustand $s \in S$ einen Pfad zu einem anderen Zustand $z \in S$ gibt ($s \rightarrow^* z$?). Im Falle von $s, z \subset S$ (anstatt $s, z \in S$) spreche ich vom **verallgemeinerten Erreichbarkeitsproblem**. Zur Beschreibung von (unendlich) großen Kripkestrukturen kann man Kellersysteme (Pushdown Systems) verwenden. $\mathcal{P} = (P, \Gamma, \hookrightarrow)$ heißt **Kellersystem**, falls P eine Menge von Zuständen, Γ eine endliche Menge (das Kelleralphabet) und $\hookrightarrow \subseteq (P \times \Gamma) \times (P \times \Gamma^*)$ eine Menge von Transitionen ist. Informal ist ein Kellersystem ein Kellerautomat ohne Eingabe. (p, w) heißt **Konfiguration**, falls $p \in P$ und $w \in \Gamma^*$. Auf Konfigurationen wird die Transitionsrelation $\hookrightarrow$ erweitert zu $\rightarrow \subseteq (P \times \Gamma^*) \times (P \times \Gamma^*)$ mit $(p, aw) \rightarrow (q, bw) :\Leftrightarrow (p, a) \hookrightarrow (q, b)$. Bei einem **Symbolischen Kellersystem** (Symbolical Pushdown Systems, **SPDS**) werden die Transitionen nur indirekt (symbolisch) mittels Relationen beschrieben. Es genügt daher lediglich diese Relationen zu spezifizieren, um Mengen von Transitionen zu beschreiben, was die Angabe des vollständigen Kellersystems vereinfacht [3]. Solche SPDS können wiederum mit Hilfe der Modellsprache Remopla [11] modelliert werden, welche zwar ähnlich zu Promelia ist (Eingabesprache für den SPIN Modellprüfer [7]), aber keine parallelen Prozesse, sondern statt dessen explizite Rekursion erlaubt. Statt des Erreichbarkeitsproblems für SPDS kann auch das LTL- oder CTL*-Modellprüfungsproblem betrachtet werden [12]. Dabei ist eine LTL- oder CTL*-Formel [13] für die durch das Kellsersystem beschriebene Kripkestruktur gegeben. Gefragt wird dann nach der Gültigkeit dieser Formel für einem Anfangszustand bzw. einer Menge von Anfangszuständen der Kripkestruktur. So konstruierte Modelle in Form eines Remopla-Programms können dann durch mein Tool **HalSPSI** verbessert und anschließend durch den Modellprüfer Moped [2] geprüft werden. In Listing 1-5 sind hierzu Remopla-Beispiele zu finden. Für weitere Details zur Konstruktion von Remopla Modellen für C und Java sei auf die Veröffentlichungen von Esparza/Schwoon [2] und Obdrzálek [14] sowie auf [5] verwiesen.

3 HalSPSI (Halle's Symbolical Pushdown System Improver)

Mein Tool **HalSPSI** ist ein Source-to-Source Compiler, welcher Symbolische Kellersysteme in Remopla-Syntax als Eingabe hat und veränderte Remopla-Modelle wieder ausgibt. Dabei sollen die Modelle derartig in Abhängigkeit der gesetzten Paramter transformiert werden, dass ein möglichst kleiner Zustandsraum für das Modell entsteht, damit der Modellprüfer einfacher das Modell überprüfen kann. **HalSPSI** liefert verschiedene Informationen (z.B. Metriken) über das zu analysierende Remopla Modell und die durchgeführten Modellanalysen und -Transformationen (einstellbar durch verschiedene Log-Levels) sowie eine Ausgabe von Kontrollflussgraphen als auch die Ausgabe von Remopla-Modellen mit Annotationen für die realisierten Modellanalysen. Hierzu ist in [5] z.B. eine Metrik (genannt ZR) beschrieben, welche den i.d.R. unendlich großen Zustandsraum quantifiziert. Die implementierten Modellanalysen und -Transformationen basieren vorranging auf dem Worklist Algorithmus von Martin [15].

Listing 1: Transformation eines Boolschen Ausdrucks in ein YICES-SMT-Problem

```
if                                  (define x1::int)
  :: (x1 == x1+1) -> ...           (assert (= x1 (+ x1 1)))
fi
```

3.1 Nicht-parasitäre Techniken

Als nicht-parasitäre Techniken bezeichne ich Modelltransformationen, welche **keinen** Einfluss weder auf die Erreichbarkeit von Konfigurationen noch auf die Aussagen von Temporalen Formeln wie LTL und CTL* haben. Entsprechend verändern parasitäre Techniken die Modelle so stark, dass entweder die Erreichbarkeit von Konfigurationen beeinflusst wird (dadurch, dass es neue False Negatives geben kann) oder sich die Aussage von LTL bzw. CTL*-Formeln ändert. Im letzteren Fall bleiben allerdings (durch zusätzliche indirekte Abhängigkeiten) oft wenigstens die Aussagen von LTL-X und CTL*-X Formeln erhalten.

3.1.1 Kontrollfluss-Slicing (-Ob)

Ich habe interprozedurales Kontrollfluss-Slicing auf Remopla-Modelle übertragen. In meiner Umsetzung wird ein Vorwärtsslice beginnend bei den Startkonfigurationen berechnet, welcher alle symbolischen Remopla-Konfigurationen des Modells enthält, die aus den Startkonfigurationen über den Kontrollfluss erreichbar sind. Der Kontrollflussgraph wird dann entsprechend verkleinert. Beginnend bei den Fehlerkonfigurationen berechne ich anschließend einen Rückwärtsslice, um symbolische Remopla-Konfigurationen zu identifizieren, welche zu Fehlern im Modell über den Kontrollfluss führen können. Alle nicht im Rückwärtsslice auf dem reduzierten Kontrollflussgraph enthaltenen symbolischen Konfigurationen können aus dem Modell entsprechend entfernt werden. Die Gültigkeit bzw. Nichtgültigkeit von LTL sowie CTL*-Formeln bleibt bei dieser Transformation selbstverständlich erhalten.

3.1.2 SMT-Reduktion (-Oy)

Durch automatische Transformationen kann es hin und wieder vorkommen, dass Artefakte wie $a + 2 > a$ oder $a * a + 5 > 0$ entstehen. Derartige (Teil-)Ausdrücke entstehen nicht nur durch die in dieser Veröffentlichung beschriebenen Techniken, sondern auch bei der (automatischen) Generierung von Remopla-Modellen. Sie können im Idealfall sogar komplett zu $True$ oder $False$ vereinfacht werden, damit der Modellprüfer weniger Zugriffe auf BDDs beim Auswerten von Ausdrücken benötigt. Um solche Boolschen Ausdrücke zu vereinfachen, wurde der SMT-Solver YICES[1] [16] in **HalSPSI** integriert, welcher anhand der Variablendefinition (definierter Wertebereich) testet, ob sich ein boolscher Ausdruck vereinfachen lässt. Hierzu wird, wie in Listing 1 zu sehen, ein gegebener Ausdruck als YICES-SMT-Problem beschrieben (es wird polnische Notation verwendet [16]).

[1]Gesamtsieger der SMT-Competition im Rahmen der CAV 2007 (Computer-Aided-Verification).

Satz 1 *(SMT-Reduzierbarkeit)*
Sei e ein boolscher Teilausdruck und $Y(e)$ das zugehörige SMT-Problem. Dann gilt:

$$Y(e) \text{ unerfüllbar} \Rightarrow [\![e]\!] \equiv false,$$
$$Y(\neg e) \text{ unerfüllbar} \Rightarrow [\![e]\!] \equiv true.$$

Der Beweis ist trivial. Sobald an einer Kontrollflussverzweigung ein Ausdruck vollständig zu $False$ oder $True$ ausgewertet wird, kann das Modell entsprechend durch "Abschneiden" der nicht benötigten Kontrollflüsse vereinfacht werden. Im Beispiel aus Listing 1 würde die Bedingung $x1 == x1 + 1$ zu $False$ vereinfacht, da es keine Belegung für $x1$ gibt, so dass die Bedingung erfüllt wird. Daraufhin wird die Verzweigung samt zugehöriger Konfigurationenübergänge aus dem Modell entfernt. Durch zusätzliche Intervallinformationen (vgl. Abschnitt 3.2.4) konnten boolsche Ausdrücke in meinen Untersuchungen öfter zu $True$ bzw. $False$ vereinfacht werden, da es dann weniger mögliche Belegungen für die im Ausdruck verwendeten Variablen gibt. Dies erhöht damit die Anzahl durchfühbarer Reduktionen und beeinflusst auch weiterhin nicht die Erreichbarkeit von Fehlerkonfigurationen oder die Aussage von LTL bzw. CTL*-Formeln.

3.1.3 Konstanten-Propagation und -Faltung (-Oc)

Nicht nur in boolschen Ausdrücken (wie gerade in Abschnitt 3.1.2 beschrieben) können Artefakte entstehen, sondern auch in arithmetischen Ausdrücken. Diese können mit Hilfe von Konstanten-Propagation und -Faltung vereinfacht werden. Gerade bei der Simulation eines Operandenstacks durch lokale Variablen (wie es bei Javabytecode mittels des Tools JMoped der Fall ist) kommen häufig Konstanten vor, welche für Berechnungen auf dem Operandenstack abgelegt werden. Diese Konstanten können aber direkt in die Berechnung propagiert werden, wodurch weniger BDD-Auswertungen nötig sind und im günstigsten Fall Variablen des Operandenstacks überflüssig werden[2]. Da sich durch diese Transformationen keine Variablenwerte ändern, bleibt nicht nur die Erreichbarkeit von Fehlerkonfigurationen erhalten, sondern auch die Aussagen von LTL bzw. CTL*-Formeln.

3.2 Parasitäre Techniken

3.2.1 Äquivalenzanalyse (-Oa)

Wird zu der im vorigen Abschnitt 3.1.3 beschriebenen Konstanten-Propagation und -Faltung noch eine "Copy-Propagation-Analysis" verwendet, so können damit so genannte "Copy-Chains"[3] verfolgt werden. Das Ziel ist das Aufbrechen dieser "Copy-Chains", damit letztendlich weniger Variablen im Modell benötigt werden und sich damit der Zustandsraum des Modells verringert. Konkret werden bei der in meinem Tool implementierten Äquivalenzanalyse Äquivalenzklassen an Konfigurationen für Variablen und zu Konstanten auswertbaren Ausdrücken bestimmt. Es befinden sich zwei Varliblen x und y an der Konfiguration p in der gleichen Klasse, falls deren konkreter Wert[4] gleich ist für jeden (interpro-

[2] Dies wird erst nach einer weiteren Analyse erkannt.

[3] Copy-Chains sind aufeinander folgende Zuweisungen mit Wertweiterreichung, wie sie typischweise bei einem simulierten Operadenstack bei Push- und Pop-Operationen auftreten.

[4] Gemeint ist der Variablenwert bei "Ausführung" des Remopla-Programms.

Listing 2: Modellmodifikation zur Reduktion auf das Erreichbarkeitsproblem (Remopla-Syntax)

```
p:...            p1: if
                     :: a != b -> goto p2;
                     :: else -> skip;
                     fi;
                 p:...
```

zeduralen) Kontrollfluss, mit dem die Konfiguration p erreicht werden kann. Für gängige Hochsprachen ist es natürlicherweise **unentscheidbar** (Reduktion Postsches Korrespondenzproblem) fest zu stellen, ob zwei Variablen x und y an einem Programmpunkt p' den gleichen Wert haben. Wie folgender Satz zeigt, ist dieses Problem für SPDS **entscheidbar**.

Satz 2 *(Entscheidbarkeit der Äquivalenzanalyse)*
Für (symbolische) Kellersysteme ist es ***entscheidbar****, ob zwei Variablen a und b (analog Variable und Konstante) an einer Konfiguration p den gleichen Wert besitzen oder nicht.*

Beweis: Nach Einführung eines neuen Konfigurationsübergangs $p1$ unmittelbar vor p (wie in Listing 2 zu sehen) gilt: a ist (immer) äquivalent zu b im Punkt p gdw. $p2$ ist nicht erreichbar[5]. q.e.d.

Sind die Äquivalenzklassen bestimmt, können Variablen mit Hilfe anderer Variablen ersetzt werden. Es werden dabei alle Verwendungen von Variablen an allen symbolischen Konfigurationen durch gewählte Repräsentanten der Äquivalenzklassen ersetzt, was zu einem Überdeckungsproblem führt und die Anzahl verwendeter Variablen reduziert. Aufgrund folgenden Satzes werden die Repräsentanten heuristisch gewählt und die jenigen Repräsentanten bevorzugt, welche besonders häufig in Äquivalenzklassen auftreten, da diese eine besonders hohe Wahrscheinlichkeit besitzen, andere Variablen zu überdecken.

Satz 3 *(Komplexität der Repräsentantenwahl)*
Die optimale Repräsentantenwahl [6] ist NP-hart.

Beweis: Reduktion des NP-vollständigen Überdeckungsproblems auf die optimale Repräsentantenwahl. Sei eine beliebige Überdeckungsmatrix $A = (a_{ij}) \in \{0,1\}^{m,n}$ gegeben. Gesucht ist eine minimale Auswahl an Zeilen, so dass deren logisches ODER den 1-Vektor $(1, 1, 1, ..., 1)$ bilden. Man konstruiere ein Kellersystem wie jenes in Abbildung 1. Dann wird durch die optimale Repräsentantenwahl für $L1..Ln$ das Überdeckungsproblem A gelöst. q.e.d.

Es ist natürlich klar, dass die Aussagen von LTL/CTL*-Formeln unverändert bleiben, solange lediglich lesende Verwendungen von Variablen durch andere ersetzt werden. Wird allerdings eine Variable überflüssig und könnte somit aus dem Modell entfernt werden, so geht dies i.A. nur, sofern die gegebene LTL/CTL*-Formel nicht über diese 'wegoptimierte' Variable spricht. Falls doch, so sind in jeder symbolischen Konfiguration des

[5] Ebenso wird hier $p2$ neu eingeführt.
[6] d.h. eine minimale Auswahl an Repräsentanten

Abbildung 1: Reduktion vom Überdeckungsproblem auf optimale Repräsentantenwahl $\{m_1, m_3\}$

A	x_1	x_2	x_3	x_4
m_1	1	1		1
m_2		1		1
m_3	1		1	1
m_4			1	1

```
  # {x1=m1=m3}        module use(int v, int p) {
L1: use(x1,1);          print(v);
  # {x2=m1=m2}          xi=undef, mi=undef; # i in [1..4]
L2: use(x2,2);          if
  # {x3=m3=m4}            :: p==1 -> m1=x2,m2=x2;
L3: use(x3,3);            :: p==2 -> m3=x3,m4=x3;
  # {x4=m1=m2=m3=m4}      :: p==3 -> m1=x4,m2=x4,m3=x4,m4=x4;
L4: use(x4,4);          fi; return; }
```

Kellersystems ggf. indirekte Abhängigkeiten einzuführen, um die durch die LTL/CTL*-Formel erzeugten indirekten Verwendungen (über Prädikate) ebenfalls zu überdecken und damit die an der LTL/CTL*-Formel beteiligten Variablen in die Lösung der Repräsentantenwahl zu zwingen. Die indirekte Verwendung einer Variablen v zur Überdeckung der LTL/CTL*-Formel ist natürlich nur dann an einer Konfiguration zu ergänzen, wenn die Konfiguration nicht bereits selbst die Überdeckung von v verlangt.

3.2.2 Richtungs-Entscheidungsanalyse (-Od)

Durch die im vorigen Abschnitt 3.2.1 erläuterte Modelltransformation entstehen typischerweise viele *tote* Zuweisungen [17]. Diese sind sogar für die symbolische Simulation des Remopla-Programms überflüssig und sollten identifiziert sowie eliminiert werden. Statt jedoch einer einfachen „Dead-Code-Analyse“ [17] habe ich mich darauf konzentriert, gleich diejenigen Variablen (und Konfigurationen) zu identifizieren, welche über die Erreichbarkeit von gegebenen Fehlerkonfigurationen **entscheiden**. Es ist natürlich klar, das eine im Sinne von Muchnick [17] tote Variable a an einem Ort p natürlich niemals die Erreichbarkeit beeinflussen kann. Auch die Entscheidungsanalyse ist für gängige Hochsprachen i.A. unentscheidbar wegen des Halteproblems. Dennoch ist dies (wie folgender Satz zeigt) für Kellersysteme entscheidbar.

Satz 4 *(Entscheidbarkeit der R-Entscheidungsanalyse)*
Für (symbolische) Kellersysteme ist es ***entscheidbar****, ob eine Variable a an einer Konfiguration p Einfluss auf die Erreichbarkeit eines Fehlerzustandes Err ausübt oder nicht.*

Beweis: Nach Konstruktion von n Kellersystemen, wie in Listing 3 zu sehen, gilt: a ist in p entscheidend für die Erreichbarkeit von Err gdw. unter den n rechts in Listing 3 abgebildeten Kellersystemen ist Err mindestens einmal erreichbar **und** mindestens einmal nicht erreichbar. q.e.d.

Listing 3: Entscheidungsanalyse formuliert als n Erreichbarkeitsprobleme (Remopla-Syntax)

```
                                q: a = 0;      # a = 0,1,2...n
p: if                           p: if
  :: a > 0 -> b = 1;              :: a > 0 -> b = 1;
  :: else  -> goto Err;           :: else  -> goto Err;
fi                              fi
```

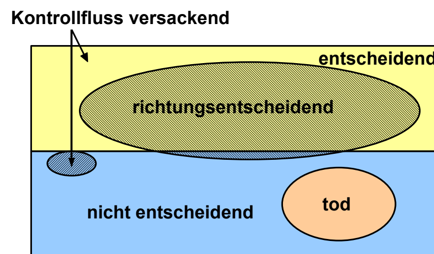

Abbildung 2: An der symbolischen Konfiguration p ist eine Variable v...

Da es allerdings zu aufwändig ist, zur Beschleunigung eines Erreichbarkeitsproblems gleich n neue Erreichbarkeitsprobleme zu lösen, habe ich in meinem Tool **HalSPSI** lediglich eine Modellanalyse integriert, um diejenigen Variablen zu bestimmen, welche lediglich potentiell einen Einfluss auf die Richtung im Kontrollfluss ausüben. Diese Variablen nenne ich **richtungsendscheidend**. Umgekehrt nenne ich Variablen **kontrollflussversackend**, falls sie die Erreichbarkeit beeinflussen, jedoch nicht richtungsentscheidend sind (siehe Abbildung 2). Nachdem mein Tool **HalSPSI** die richtungsendscheidenden Variablen bestimmt hat, werden sämtliche seiteneffektfreie Zuweisungen nichtrichtungsendscheidender Variablen aus dem Modell eliminiert, da diese Zuweisungen keinen großen Einfluss auf die Erreichbarkeit ausüben. Um auch die Semantik von LTL-X/CTL*-X-Formeln zu bewahren, sind wiederum a priori die an der LTL-X/CTL*-X-Formel beteiligten Variablen als indirekt richtungsendscheidend zu markieren. Neue False-Negatives kann es nur geben, falls eine Variable sowohl entscheidend, aber nicht richtungsendscheidend, also demnach kontrollflussversackend ist. Diese Fälle treten nur auf, wenn es nach Remopla-Semantik keine Nachfolgerkonfiguration gibt (bei arithmetischen Überläufen). Diese Stellen werden erfreulicherweise als Warnung durch die im Abschnitt 3.2.4 erläuterte Intervallanalyse gemeldet.

3.2.3 Stotterreduktion (-Ot)

Wie in Listing 4 zu sehen, können zur Transitions- und Konfigurationsreduktion gewisse symbolische Konfigurationen zusammengelegt (verschmolzen) werden, sofern diese sich gegenseitig nicht beeinflussen. Dadurch entstehen kleinere Modelle, welche der Modellprüfer schneller überprüfen kann. Werden die zu verschmelzenden Konfigurationen ungünstig gewählt, so gelangt man. wie in Listing 4 (Mitte) zu sehen, u.U. zu einem lokalen Optimum. Unter Verwendung von Sheduling-Techniken [18] ist es allerdings in Linear-

Listing 4: l2+l3 → lokales Optimum vs. l1+l2 und l3+l4 → globales Optimum

```
l1: x = 0;          l1: x = 0;              l12: x = 0, y = 6;
l2: y = 6;         l23: y = 6, a = x+1;     l34: a = x+1, b = y*2;
l3: a = x+1;        l4: b = y*2;
l4: b = y*2;
```

Listing 5: Eine durch Konstantenfaltung und/oder -Propagation nicht erkennbare Konstante y

```
x = 0;
L: y = x/2;
if
  :: (x = 0) -> x = 1; goto L;
  :: else -> break;
fi
```

zeit möglich, stets optimale Transformationen (Listing 4 rechts) zu finden. Die Erreichbarkeitsfrage für vorgegebene Konfigurationen wird (nach geschickter Wahl der Remopla-Programmmarken) nicht beeinflusst. Sofern eine LTL-X bzw. CTL*-X-Formel nicht über die an einer Verschmelzung beteiligten Variablen spricht, bleibt die Aussage dieser Formel bei der Transformation invariant. Zusätzliche künstliche Abhängigkeiten für die an einer Formel beteiligten Variablen verhindern wiederum das Verschmelzen entsprechender symbolischer Konfigurationen und gestatten damit analog die Invarianz der Aussage dieser Formel.

3.2.4 Intervallanalyse (-Oi)

In SPDS werden Variablenwerte durch Bitvektoren im Zustand und Kelleralphabet des Kellersystems kodiert. Eine Verkleinerung dieser Bitvektoren reduziert (inbesondere im Falle von Rekursion) den Zustandsraum des Modells erheblich. Durch die in meinem Tool **HalSPSI** implementierte Intervallanalyse werden die Wertebereiche von Variablen durch Intervalle überschätzt und die Variablendefinitionen auf die nötigen Bits verkleinert. Insbesondere hat sich diese Technik bei Verkleinerung von Arrays auf deren maximale Lesegrenzen als sehr nützlich zur Reduktion erwiesen. Neben nicht erwünschten potentiellen Überläufen arithmetischer Berechnungen werden durch die Intervallanalyse auch semantisch nicht erreichbare Konfigurationen identifiziert und aus dem Modell (analog zum Kontrollfluss-Slicing in Abschnitt 3.1.1) ausgeschlossen. Zusätzlich profitiert aber auch die SMT-Reduktion von den Intervallinformationen, wie in Abschnitt 3.1.2 beschrieben. Dem nicht genug, wird zusätzlich die Erkennung von Konstanten verbessert, wie Listing 5 zeigt. Die Variable y hat dort stets den Wert 0, was durch die Intervallanalyse erkannt wird, nicht jedoch durch Konstantenfaltung oder -Propagation. Analog zur Äquivalenzanalyse bleiben hier auch nur diejenigen LTL- und CTL*-Formeln aussageninvariant, welche nur über nicht wegoptimierte Variablen sprechen.

Abbildung 3: Ausgewählte Methoden mit Zeit- und Speicherüberläufen (5 Stunden, 2 GB RAM)

Methode(n)	Gesamtzeit	GenZeit	timeouts	memouts
-Oaydti	1.96h	0.1h	0%	6.7%
-Oyi	8.8h	0.1h	0.7%	4%
-Ob	20.6h	0.1h	0.7%	20%
-Oaydt	29.7h	0.1h	2%	18.7%
-Oayd	43.9h	0.1h	3.3%	19.3%
ohne HalSPSI	48.9h	0.1h	0.7%	25.3%

Abbildung 4: Eigenschaften der implementierten Modellanalysen und -transformationen

Eigenschaft	-Ob	-Oy	-Oc	-Oa	-Od	-Ot	-Oi
parasitär				+	+	+	+
flusssensitiv	+		+	+	+	+	+
interprozedural	+		+	+	+		+
entscheidbar	+	+	+	+	+	+	+
keine False-Negtives	+	+	+	+		+	+
erreichbarkeitserhaltend	+	+	+	+		+	+
LTL/CTL* invariant	+	+	+	+			+
LTL-X/CTL*-X invariant	+	+	+	+	+	+	+

Satz 5 *(Entscheidbarkeit der Intervallanalyse)*
Seien $a, b \in \mathbb{R}$. Dann ist für (symbolische) Kellersysteme ***entscheidbar****, ob für eine Variable x an einer Konfiguration p stets gilt: $a \leq x \leq b$.*

Beweis: analog zur Äquivalenz- und Entscheidungsanalyse durch Reduktion auf das Erreichbarkeitsproblem.

4 Zusammenfassung und Ergebnisse

Für symbolische Kellersysteme habe ich verschiedene Modellanalysen und Modellreduktionstechniken implementiert, welche die Software-Modell-Prüfung in meinen Tests für den Modellprüfer Moped erheblich beschleunigen (um bis zu 96% auf nur noch 4% der zuvor benötigten Laufzeit). Die Tabelle in Abbildung 3 zeigt die Laufzeiten für einige ausgewählte Kombinationen der Modellanalysen für meinen Benchmark (30 Java-Beispiele mit assert(false)-Prüfung, Modellparameter 4-8 Bits, AMD X2 4200, 2400 Einzeltests). Dabei ist GenZeit die Zeit, welche zum Generieren und Optimieren des Modells benötigt wurde, vernachlässigbar klein gegenüber der Zeit, die zur Modellprüfung benötigt wird. In einigen Fällen wird die Modellprüfung aber auch erst überhaupt ermöglicht (bis zu 21,3% der Speicherüberläufe verhindert) und in anderen Fällen erübrigt sich die Modellprüfung sogar ganz. Letzteres z.B. dann, wenn durch eine Modellanalyse meines Tools **HalSPSI** bereits die Korrektheit des Modells nachgewiesen werden konnte (je nach verwendeter Parameter 7%-10%). Abbildung 4 fasst die aufgeführten Eigenschaften nochmals zusammen.

Literatur

[1] D. Suwimonteerabuth, S. Schwoon, J. Esparza. *jMoped: A Java Bytecode Checker Based on Moped*. Tools and Algorithms for the Construction and Analysis of Systems (TACAS), LNCS, Springer, 2005.

[2] J. Esparza, S. Schwoon. *A BDD-based model checker for recursive programs*. LNCS Volume 2102, Seite 324-336, Springer, 2001.

[3] S. Schwoon. *Model-Checking Pushdown Systems*. Technische Universität München, 2002.

[4] L. Burdy, Y. Cheon, D.R. Cok, M.D. Ernst, J.R. Kiniry, G.T. Leavens, R. Leino, E. Poll. *An overview of JML tools and applications*. International Journal on Software Tools for Technology Transfer (STTT), Volume 7, Seite 212-232, Springer, 2005.

[5] D. Richter, W. Zimmermann. *Slicing zur Modellreduktion von symbolischen Kellersystemen*. Proceedings of the 24. Workshop of GI-section 'Programmiersprachen und Rechenkonzepte', University Kiel, 2007.

[6] T. A. Henzinger, R. Jhala, R. Majumdar, G. Sutre. *Software Verification with BLAST*. In 10th International Workshop on Model Checking of Software (SPIN), LNCS Volume 2648, Seite 235–239. Springer, 2003.

[7] G. J. Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, 2004.

[8] W. Visser, K. Havelund, G. Brat, S. Park, F. Lerda. *Model Checking Programs*. Automated Software Engineering Volume 10(2), Seite 203-232, Kluwer Academic, 2003.

[9] T. Andrews, S. Qadeer, S. K. Rajamani, J. Rehof, and Y. Xie. *Zing: A Model Checker for Concurrent Software*. MSR Technical Report: MSR-TR-2004-10, 2004.

[10] J. Hatcliff, M. B. Dwyer. *Bogor: An Extensible Framework for Domain-Specific Model Checking*. Newsletter of European Association of Software Science and Technology (EASST), Technical Report, SAnToS-TR2004-9., 2004.

[11] S. Kiefer, S. Schwoon, D. Suwimonteerabuth. *Introduction to Remopla*. Institute of Formal Methods in Computer Science, University of Stuttgart, 2006.

[12] R. Mayr. *Process Rewrite Systems*. In Electronic Notes in Theoretical Computer Science, Volume 7, Seite 185-205, 1997.

[13] E. M. Clarke, O. Grumberg, D. Peled. *Model Checking*. MIT Press, Massachusetts Institute of Technology, Cambridge, 2000.

[14] J. Obdrzálek. *Formal Verification of Sequential Systems with Infinitely Many States*. Masaryk University, 2001.

[15] F. Martin. *Generating Program Analyzers*. University of Saarland, 2006.

[16] B. Dutertre, L. Moura. *The YICES SMT Solver*. Computer Science Laboratory, SRI International, USA, 2006.

[17] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, San Francisco, 1997.

[18] W. Zimmermann. *Planbare Algorithmen - Eine Methode zum maschinenunabhängigen parallelen Programmieren*. Seite 154-183, Shaker, 1999.

Wortbasierte Symbolische Simulation Hybrider Systeme in CLP

Elke Tetzner

Universität Rostock, IEF
D-18051 Rostock
tetzner@informatik.uni-rostock.de

1 Einleitung

Symbolische Simulationen [NC94,MT99,BSW02,Gar02,LC05,Wan07] haben sich in hybriden Systemen zur automatischen Verifikation von zeit- und sicherheitskritischen Eigenschaften bewährt. Unsere symbolische Simulation wird bezüglich [RU95,UR95] aus einer wortbasierten Sichtweise zum Nachweis von Eigenschaften sich synchronisierender und hierarchisch aufgebauter hybrider Automaten ausgeführt. Die Überprüfung des Verhaltens hybrider Systeme erfolgt somit auf der Grundlage formaler Sprachen zum Zweck der Gewinnung von Erkenntnissen über entscheidbare Probleme, wobei die symbolische Simulation als Prozess der Akzeptanz von Zeitwörtern eingesetzt wird. Ein Zeitwort wird durch eine Folge von Symbolen für auftretende Ereignisse gebildet, wobei jedem Ereignis eine symbolische Zeit des Auftretens, eine Variablenmenge sowie Bedingungen über der Zeit und der Variablenmenge zugeordnet sind. Die genaue Definition von Zeitwörtern hängt von zugrundeliegenden Automaten ab. Als Grundlage dienen hier hybride Automaten im Sinn von [ACHH93,ACH$^+$95]. Welche Besonderheiten diese Automaten aufweisen, wie dementsprechend Zeitwörter formuliert sein müssen und welche Ergebnisse die symbolische Simulation unter solchen Gegebenheiten liefert, ist in den nächsten Abschnitten dargestellt.

2 Hybride Automaten

Hybride Automaten bilden eine Erweiterung endlicher Automaten, an deren Komponenten Variablen gebunden sind, welche sich entsprechend gegebener Funktionen entwickeln und gegebener Relationen verhalten müssen. Die Erweiterung erfolgt derart, dass *kontinuierliche Komponenten*, Lokationen, und *diskrete Komponenten*, Übergänge zwischen den Lokationen, entstehen. In einer *Lokation* können unendlich viele Variablenbelegungen unter Beachtung einer Bedingung, *Invariante*, und entsprechend beschriebener Fortschrittsfunktionen, *Aktivitäten*, auftreten. Der konkrete Zustand eines hybriden Automaten ist durch ein Tupel aus der vorliegenden Lokation und Variablenbelegung erklärt. Die Variablenbelegung, unter welcher ein Übergang von einer Lokation zu einer nachfolgenden Lokation stattfinden soll, muss festgelegten *Transitionsbedingungen* genügen und kann durch Zuweisungsfunktionen, *Aktionen*, neu belegt werden. In der Abbildung 1 sind die Bestandteile zur Beschreibung von Deklarationen und dem Verhalten eines hybriden Automaten noch einmal graphisch notiert.

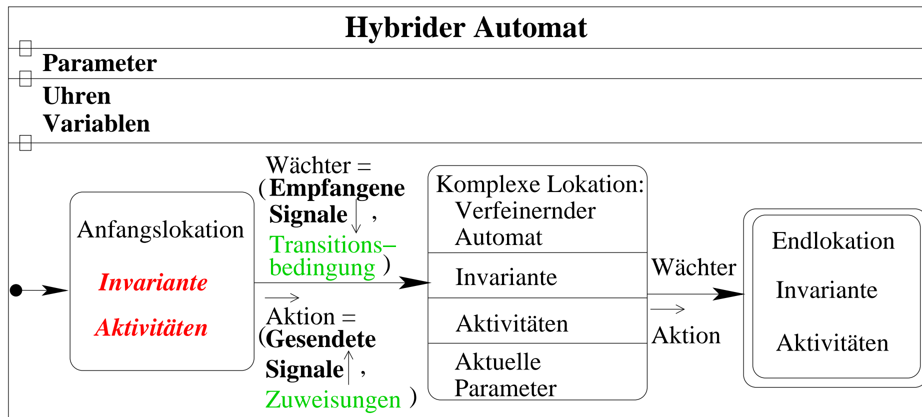

Abbildung 1. Funktionen und Relationen in hybriden Automaten

Anfangslokationen werden durch einen zusätzlichen Eingangspfeil und Endlokationen durch eine doppelte Umrandung gekennzeichnet. Lokationen, die durch weitere hybride Automaten aus Bibliotheken verfeinert werden können, sind als *komplexe Lokationen* spezifiziert und führen zu Hierarchisierung hybrider Automaten. Über gesendete und empfangene Signale können sich nebenläufige Automaten synchronisieren. Zu empfangende Signale werden mit einem nach unten gerichteten Pfeil im *Wächter* zusammen mit der Transitionsbedingung festgelegt und zu sendende Signale mit einem nach oben gerichteten Pfeil in der *Aktion* zusammen mit den Zuweisungen.

3 Zeitwörter

Automaten können Wörter akzeptieren. In hybriden Automaten werden insbesondene Zeitwörter betrachtet. Allgemein ist ein Zeitwort wie in folgender Definition festgelegt.

Definition 1. *Ein* **Zeitwort** ZW *bildet eine endliche bzw. unendliche Folge von Tupeln* $ZW = \langle S_1,\ T_1,\ X_1,\ \omega_1\rangle,...,\langle S_n,\ T_n,\ X_n,\ \omega_n\rangle$ *mit* $i \in \{1,..,n\}$ *und* $n \rightarrow \infty$*, wobei gilt:*

S_i = *Menge von Symbolen,*
T_i = *symbolische Zeit,*
X_i = *Menge von Variablen,*
ω_i = *Menge von Abbildungen der Variablen auf konkrete und symbolische Werte.*

Die Menge der Abbildungen ω_i hängen in Bezug auf die zugrundeliegenden Automaten von unterschiedlichen Bedingungen ab. Während die Abbildungen in Zeitautomaten [Alu99] durch Invarianten, Transitionsbedingungen und Aktionen beeinflusst werden, sind für die Abbildungen von Zeitwörtern bezüglich hybrider Automaten zusätzlich Aktivitäten relevant. Mit Hilfe von Aktivitäten kann das zeitliche Fortschreiten der kontinuierlichen Variablen von Lokation zu Lokation variieren. Neben den Invarianten wird

somit durch die Aktivitäten bestimmt, wie lange ein hybrider Automat in einer Lokation verbleiben kann bzw. wann der Automat die Lokation verlassen muss.
Da Wörter an Transitionen akzeptiert werden, sind Invarianten und Aktivitäten nicht explizit an der Beschreibung der Abbildungen ω_i beteiligt. Invarianten werden nach [BS97] in *unbedingte Transitionsbedingungen* transformiert, die eine Aussage darüber treffen, wann eine Transition ausgeführt werden *muss*. Demgegenüber besagt die Bedingung, welche an einer Transition beschrieben wurde, wann die Transition ausgeführt werden *kann*. Die Aktivitäten, welche ursprünglich in Form von Differentialgleichungen festgelegt sind, werden in Differenzengleichungen überführt, da für die Akzeptanz der Zeitwörter von dem kontinuierlichen Verlauf abstrahiert wird. Beobachtet werden mögliche Randwerte für Variablen, die bei dem Übergang von einer Lokation zu einer nachfolgenden Lokation auftreten. Dabei wird von der Annahme ausgegangen, dass der kontinuierliche Fortschritt innerhalb einer Lokation ohne kritische Abschnitte erfolgt. Solche Annahmen können vorher durch Verfahren wie Simulation kontinuierlicher Systeme nachgewiesen werden.
Abbildungen in ω von Zeitwörtern über hybriden Automaten lassen sich entsprechend folgender Grammatik formalisieren:

```
<ω> ::= ⟨ <VorTrans>, <NachTrans> ⟩
<VorTrans> ::= release( <DiffGleich>, <TransBeding> )
<TransBeding> ::= <UnbedingtTrans> ∧ <BedingtTrans>
<NachTrans> ::= conclude( <Aktionen>, <EintrittInv> )
```

Ein Zeitwort kann von einem hybriden Automaten akzeptiert werden, wenn die Variablenbelegungen einenseits Bedingungen zum Auslösen einer Transition in ’VorTrans’ und andererseits Bedingungen zum erfolgreichen Beenden der Transition in ’NachTrans’ erfüllen. Dabei setzen sich die Bedingungen aus ’VorTrans’ durch die aus den Aktivitäten entstandenen Differenzengleichungen ’DiffGleich’, den aus den Invarianten entstandenen unbedingten Transitionsbedingungen ’UnbedingtTrans’ und den an den Übergängen spezifizierten Transitionsbedingungen ’BedingtTrans’ zusammen. Die Bedingungen von ’NachTrans’ werden aus den Zuweisungen der ’Aktionen’ der auszuführenden Transition und der Invariante ’EintrittInv’, welche für die nachfolgende Lokation gilt, gebildet.

4 Symbolische Simulation

Sämtlichen Zeitwörtern, die von einem hybriden Automaten akzeptiert werden und in ihrer Gesamtheit das Verhalten des Automaten bilden, stehen vom Nutzer definierte Zeitwörter zur Spezifikation zeit- und sicherheitskritischer Eigenschaften gegenüber. Zum Nachweis der vom Nutzer spezifizierten Eigenschaften in hybriden Systemen wird die symbolische Simulation in CLP [JM94,FA97,HW07] (Constraint Logic Programming) eingesetzt. CLP vereinigt zwei Paradigmen:

– Lösen von Constraints durch effiziente mathematische Verfahren und
– Logische Programmierung zur deklarativen Beschreibung sowie Lösungssuche.

Ausgehend von dem Grundgedanken, das Akzeptanzproblem der vom Nutzer als Zeitwort definierten Eigenschaft durch eine Anfrage an das hybride System mit effizienten Techniken zu lösen, kann die symbolische Simulation in CLP wie folgt erklärt werden.

Begriff 1
Die **symbolische Simulation in CLP** *ist durch die Abarbeitung einer Anfrage*
*'**query**(*Eigenschaft, Hybrides System, Startlokation, Startzeit, StartVarBelegung*)'*
als Ziel G nach:

der Erfüllbarkeit der Eigenschaft in Form eines Zeitwortes
an das hybride System in Form einer Menge von Regeln in CLP

vom Anfangszustand $\langle G, true \rangle$ mit dem wahren Constraintspeicher $true$ festgelegt. Während der Ausführung werden an die Zeiten und Variablen der als Zeitwort vorliegenden Eigenschaft Bedingungen des Nutzers als auch des hybriden Systems gebunden, die als Constraintsystem im Constraintspeicher C durch den Constraintlöser vereinfacht, gelöst bzw. zu Widersprüchen geführt werden.

Anhand eines Beispiels werden der Lösungsvorgang und Ergebnisse der symbolischen Simulation in CLP illustriert.

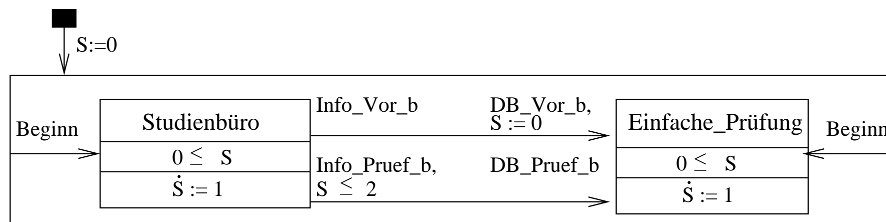

Abbildung 2. Synchronisation des Studienbüros mit dem Prüfungsablauf

In der Abbildung 2 ist ein stark vereinfachtes Beispiel zur Synchronisation eines Ablaufes in einem 'Studienbüro', bei welchem Informationen zum Bestehen von Prüfungen erfasst werden, mit dem Prozess zur Verwaltung in einer Datenbank 'Einfache_Prüfung', bei welchem die erfassten Informationen gespeichert werden, dargestellt. Das dargestellte System ist ein zur Umgebung offenes System, welches sich auch mit der Umgebung synchronisiert. Mit Hilfe einer Uhr 'S' für die Systemzeit werden die zeitlichen Bedingungen der Synchronisation festgelegt. Zu Beginn wird die Uhr 'S' an einer für alle an der Synchronisation teilnehmenden Prozesse beginnenden Transition auf Null gesetzt. Die beginnende Transition ist mit einem schwarz ausgefüllten Rechteck gekennzeichnet. Während der Ausführung der Prozesse muss die Uhr 'S' eine Systemzeit beinhalten, die gleich bzw. größer Null ist. Dabei schreitet die Uhr 'S' laut '$\dot{S}$:=1' mit einem Anstieg von einer Zeiteinheit pro Takt fort.
Beide Prozesse, 'Studienbüro' und 'Einfache_Prüfung', werden durch ein Ereignis 'Beginn' gestartet, welches entsprechend SDL [EHS97] als Signal aus der Umgebung empfangen wird. Wird im Studienbüro die Information 'Info_Vor_b' einer bestandenen Vor-

aussetzung erfasst, so wird diese Information unter Zurücksetzen der Uhr 'S' auf Null, als Signal 'DB_Vor_b' an 'Einfache_Prüfung' gesendet. Wird im Studienbüro die Information 'Info_Pruef_b' einer bestandenen Prüfung unter der Bedingung '$S \leq 2$' erfasst, so wird diese Information als Signal 'DB_Pruef_b' an den Datenbankprozess 'Einfache_Prüfung' gesendet.
In den Abbildungen 3 und 4 sind die Verhaltensbeschreibungen der Abläufe von 'Studienbüro' und 'Einfache_Prüfung' enthalten.

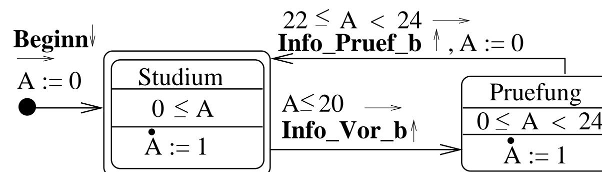

Abbildung 3. Ablauf im Studienbüro

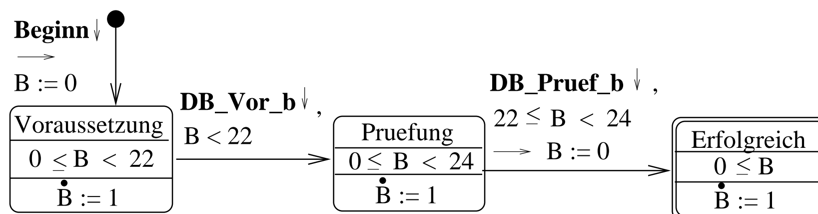

Abbildung 4. Ablauf der Prüfung

Eine Uhr 'A' zur Messung der lokalen Zeit wird für das Studienbüro zu Beginn auf Null gesetzt. In einer Anfangslokation 'Studium', welche auch die Endlokation zur Akzeptanz eines vollständig bestandenen Prüfungsablaufes bildet, verbleibt der Prozess bis eine Information einer erfolgreich bestandenen Voraussetzung zur Prüfungszulassung als Signal 'Info_Vor_b' an den Datenbankprozess 'Einfache_Prüfung' weitergeleitet werden kann. Dabei gilt für den zeitlichen Verlauf von 'A' die Bedingung '$0 \leq A$', wobei 'A' mit einer Zeiteinheit (Monat) pro Takt fortschreitet. Die Voraussetzung zur Prüfung *kann* laut einer Durchführungsordnung nur innerhalb der ersten 20 Monaten von einem Studenten erbracht werden. Die nachfolgende Lokation 'Pruefung' *muss* laut einer Studien- und Prüfungsordnung bereits im 24. Monat verlassen sein, d.h. später ist das Bestehen der Prüfung nicht mehr möglich. Laut der Bedingung '$22 \leq A < 24$' an der Transition zum Erfassen und Senden des Signals 'Info_Pruef_b' über eine bestandene Prüfung ist der offizielle Prüfungszeitraum auf den 22. und 23. Monat festgelegt, wodurch eine Prüfung andererseits durch eine Durchführungsordnung nicht vor dem 22.

Monat bestanden werden kann. Beim Bestehen der Prüfung wird die Uhr 'A' für einen erneuten Ablauf auf Null zurückgesetzt.
Durch ein Signal 'Beginn' aus der Umgebung wird wie in Abbildung 4 ein Prozess für einen Prüfungsvorgang in einer Datenbank aktiviert, wobei eine lokale Uhr 'B' mit Null initialisiert ist. Eine Prüfungsvoraussetzung muss und kann laut der Lokationsbedingung '$0 \leq$ B $<$ 22' und der nachfolgenden Transitionsbedingung bis zum 22.Monat erbracht werden. Ist die Voraussetzung erbracht, so führt eine bestandene Prüfung im Zeitraum vom 22. bis zum 24. Monat zu einem erfolgreichen Abschluss.
Zeitwörter des synchronisierend hybriden Automaten der Abbildung 2 können nun wie folgt notiert werden:

$\langle$\{**Beginn**$\downarrow$\}, T[0] \{S, A, B\},
$\langle$release(\{\}, 'true'),
conclude(\{$Snach$=0, $Anach$=0, $Bnach$=0\},
$0 \leq Snach \wedge 0 \leq Anach \wedge 0 \leq Bnach < 22$)$\rangle\rangle$,
$\langle$\{**DB_Vor_b**$\downarrow$,**Info_Vor_b**$\uparrow$\}, T[1], \{S, A, B\},
$\langle$release(\{S=$Svor$+(T[1]-T[0]), A=$Avor$+(T[1]-T[0]), B=$Bvor$+(T[1]-T[0])\},
$(0 \leq S \wedge 0 \leq A \wedge 0 \leq B \leq 22) \wedge (A < 20 \wedge B < 22)$),
conclude(\{\},$0 \leq Snach \wedge 0 \leq Anach < 24 \wedge 0 \leq Bnach < 24$)$\rangle\rangle$,
$\langle$\{**DB_Pruef_b**$\downarrow$,**Info_Pruef_b**$\uparrow$\}, T[2], \{S, A, B\},
$\langle$release(\{S=$Svor$+(T[2]-T[1]), A=$Avor$+(T[2]-T[1]), B=$Bvor$+(T[2]-T[1])\},
$(0 \leq S \wedge 0 \leq A \leq 24 \wedge 0 \leq B \leq 24) \wedge (22 \leq A < 24 \wedge 22 \leq B < 24)$),
conclude(\{$Anach$=0, $Bnach$=0\},$0 \leq Snach \wedge 0 \leq Anach \wedge 0 \leq Bnach$)$\rangle\rangle$

Dabei ist die Variable der symbolischen Zeit T für jede Transition mit einer fortlaufenden Nummer indiziert. Für die Variablen der Uhren S, A und B gilt:

$Svor$, $Avor$, $Bvor$	=	Werte der Variablen bei erfolgreichem Abschluss der vorherigen Transition
$Snach$, $Anach$, $Bnach$	=	Werte der Variablen bei erfolgreichem Abschluss der betrachteten Transition.

Diese Unterscheidung wurde vorgenommen, um syntaktisch hervorzuheben, wann welcher Wert einer Variablen verwendet wird. Die Unterscheidung ist auch für die symbolischen Simulation in CLP von Bedeutung, da dort keine expliziten Zuweisungsoperatoren vorhanden sind und Variablen mathematischen Werten entsprechen. In den Zeitwörtern wurden Invarianten bereits in unbedingte Transitionsbedingungen transformiert.
Weiterhin ist ein Nutzer an dem Bestehen der Voraussetzung zur Prüfung innerhalb von 19 Monaten interessiert, was durch folgendes Zeitwort formuliert werden kann:

$\langle$\{**Beginn**$\downarrow$\}, T[0] \{S, A, B\},true$\rangle$,
$\langle$\{**DB_Vor_b**$\downarrow$,**Info_Vor_b**$\uparrow$\}, T[1], \{S, A, B\},$T[1] \leq 19\rangle$,
$\langle$\{**DB_Pruef_b**$\downarrow$,**Info_Pruef_b**$\uparrow$\}, T[2], \{S, A, B\},true$\rangle$

Wird nach dieser Eigenschaft in unserem hybriden System gefragt, so ergibt sich während des Ablaufes der symbolischen Simulation eine Berechnung, wie diese in Abbildung 5 nachvollzogen werden kann. Jedem Symbol des Zeitwortes ist ein Aufruf des Prädikates zur Abarbeitung durch die symbolische Simulation zugeordnet, wobei drei

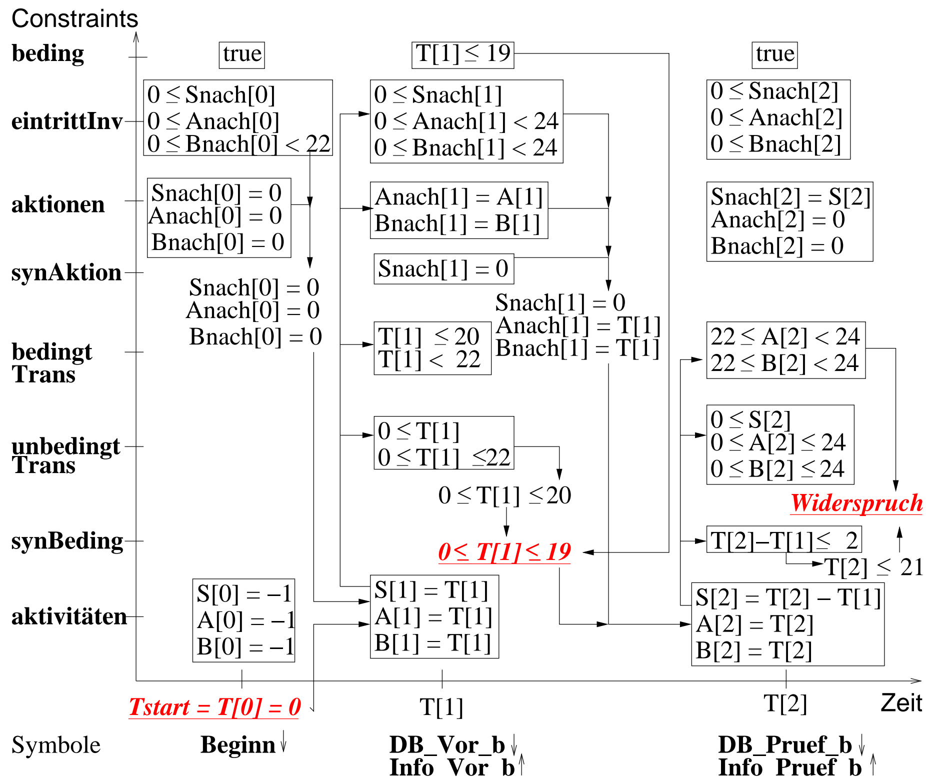

Abbildung 5. Ergebnisse des Laufes

Aufrufe von 0 bis 2 zu unterscheiden sind. Alle Variablen sind hier entsprechend der Zugehörigkeit zu einem Aufruf mit der Nummer des Aufrufes initialisiert. Während eines Aufrufes werden sämtliche Constraints des hybriden Systems und des Nutzers, die zu dem betrachteten Symbol gehören, im Constraintspreicher gesammelt und als Constraintsystem gelöst. Im Constraintspeicher verbleibende Variablen mit ihren Constaints bilden zusammen mit den Variablenbedingungen des folgenden Aufrufes weiter zu lösende Constraintsysteme, wodurch induktiv Beziehungen zu allen weiteren Aufrufen entstehen und die Gesamtlösung ermittelt werden kann. Aus der zusätzlichen Bedingung des Nutzers ergibt sich im Zusammenhang mit ’T[2]-T[1] $\leq$ 2’ und ’22 $\leq$ T[2] < 24’ als Bedingungen des dritten Aufrufes ein **Widerspruch**. Entweder können zur Erfüllung der Nutzeranforderung die Prüfungszeiträume zwischen 22 und 24 Monaten nicht eingehalten werden oder die Bedingung der übergeordneten Verordnung, dass eine Prüfung maximal 2 Monate nach dem Bestehen der Voraussetzung absolviert werden muss. Als Konsequenz der Lösung sind entweder die Ansprüche des Nutzers zu ändern oder das Modell ist zu verbessern.

5 Ausblick

Noch werden Zeitwörter in flachen Strukturen überprüft. Zur Verbesserung der Effizienz bezüglich Zeit und Speicherplatz sind in der Zukunft geschachtelte Wörter wie bei [AM06,AAB+07] zu untersuchen. Durch den Einsatz von Techniken wie direktes Backjumping zu aufgetretenen Konflikten anstelle des üblichen Backtracking und Konfliktlernen wie aus [EFH08] sowie priorisierte Constraints entsprechend gewählter Anwendungsgebiete [BLLT07] kann weiterhin eine effizientere Ausführung der symbolischen Simulation in CLP erreicht werden. Grenzen zu entscheidbaren Unterklassen hybrider Systeme sind im Allgemeinen schwer zu ermitteln [AMPS08], wodurch verschiedene Richtungen in Ansätzen zur Untersuchung solcher Probleme notwendig sind.

Referenzen

[AAB+07] R. Alur, M. Arenas, P. Barceló, K. Etessami, N. Immerman, and L. Libkin. First-Order and Temporal Logics for Nested Words. In *LICS*, pages 151–160. IEEE Computer Society, 2007.

[ACH+95] R. Alur, C. Courcoubetis, N. Halbwachs, T.A. Henzinger, X. Nicollin P.-H. Ho, A. Olivero, J. Sifakis, and S. Yovine. The Algorithmic Analysis of Hybrid Systems. In *TCS*, number 138 in TCS, pages 3–34. Elsevier Science, 1995.

[ACHH93] R. Alur, C. Courcoubetis, T.A. Henzinger, and P.-H. Ho. Hybrid Automata: An Approach to the Specifi cation and Verifi cation of Hybrid Systems. In *Hybrid Systems*, number 736 in LNCS, pages 209–229. Springer, 1993.

[Alu99] R. Alur. Timed automata. In *CAV'99*, volume 1633 of *LNCS*, pages 8–22. Springer, 1999.

[AM06] R. Alur and P. Madhusudan. Adding Nesting Structure to Words. In Oscar H. Ibarra and Zhe Dang, editors, *Developments in Language Theory*, volume 4036 of *LNCS*, pages 1–13. Springer, 2006.

[AMPS08] E. Asarin, V. Mysore, A. Pnueli, and G. Schneider. Low dimensional hybrid systems: Decidable, undecidable, don't know. Journal paper, to be submitted soon to *Information and Computation*, 2008.

[BLLT07] H. Buchholz, A. Landsmann, P. Luksch, and E. Tetzner. Verifi kation zeitkritischer Eigenschaften paralleler Programme. In *PARS'07*, pages 156–165, Hamburg-Harburg, Germany, Dezember 2007.

[BS97] S. Bornot and J. Sifakis. Relating time progress and deadlines in hybrid systems. In *HART '97*, pages 286–300, London, UK, 1997. Springer-Verlag.

[BSW02] R.-J. Back, C. Cerschi Seceleanu, and J. Westerholm. Symbolic simulation of hybrid systems. In *APSEC '02*, pages 147–155, Washington, DC, USA, 2002. IEEE Computer Society.

[EFH08] A. Eggers, M. Fränzle, and C. Herde. SAT modulo ODE: A direct SAT approach to hybrid systems. Reports of SFB/TR 14 AVACS 37, SFB/TR 14 AVACS, April 2008. ISSN: 1860-9821.

[EHS97] J. Ellsberger, D. Hogrefe, and A. Sarma. *SDL-Formal Object-oriented Language for Communicating Systems*. Prentice Hall, London, 1997.

[FA97] T. Frühwirth and S. Abdennadher. *Constraint-Programmierung*. Springer Verlag, Berlin, 1997.

[Gar02] D. Garriou. Symbolic simulation of synchronous programs. *Theoretical Computer Science*, 65(5), 2002.

[HW07] P. Hofstedt and A. Wolf. *Einführung in die Constraint-Programmierung - Grundlagen, Methoden, Sprachen, Anwendungen*. Springer, Berlin, 2007.

[JM94] J. Jaffer and M.J. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 20:503–581, 1994.

[LC05] J. Lee and S. Chi. Using symbolic devs simulation to generate optimal traffi c signal timings. *Simulation*, 81(2):153–170, 2005.

[MT99] C. Mauras and R. Thoraval. A propos de la vérifi cation de programmes synchrones et de l'analyse de programmes logiques aves contraintes. In *JFPLC*, pages 39–54. Hermes, 1999.

[NC94] S. Narain and R. Chadha. Symbolic discrete - event simulation. In P.R. Kumar and P. Varaiya, editors, *Discrete Event Systems, Manufacturing Systems and Communication Networks*, number 73 in IMA volumes in mathematics and its applications, pages 201–224. Springer, 1994.

[RU95] G. Riedewald and L. Urbina. Symbolische Simulation Hybrider Systeme in Constraint Logic Programming. Preprint CS-02-95/CS-03-95, University Rostock, Department of Computer Science, 1995.

[UR95] L. Urbina and G. Riedewald. A framework for symbolic simulation of hybrid systems in constraint logic programming. In *WLP*, pages 29–38, 1995.

[Wan07] F. Wang. Symbolic simulation-checking of dense-time automata. In *FORMATS*, volume 4763 of *LNCS*, pages 352–368. Springer, 2007.

Datenfluss in bedarfsgesteuerten Berechnungen

Sebastian Fischer[1] and Herbert Kuchen[2]

[1] Institut für Informatik
Christian-Albrechts-Universität zu Kiel
[2] Institut für Wirtschaftsinformatik
Westfälische Wilhelms-Universität Münster

Quelltext-Überdeckung ist ein häufig verwendeter Maßstab zur Bewertung der Qualität von Programmtests. Sie erlaubt eine Beurteilung der durchgeführten Testläufe anhand der von ihnen ausgeführten Teile des Programms. Überdeckungs-Kriterien für imperative Sprachen reichen von einfachen Kriterien wie

- *Anweisungs-Überdeckung*, bei der überprüft wird, welche im Programm notierten Anweisungen ausgeführt werden, über
- *Kontrollfluss-Überdeckung*, bei der überprüft wird, welche Knoten bzw. Kanten des dem Programm zugeordneten Kontrollfluss-Graphen, durchlaufen werden, bis hin zu
- *Datenfluss-Überdeckung*, bei der überprüft wird, welche Zuweisungen an Variablen welche ihrer Verwendungen beeinflussen.

Jüngst gewinnt Quelltext-Überdeckung auch im Bereich funktionaler Programmierung an Bedeutung, da die Programmiersprache Haskell [1] in zunehmendem Maße für die Entwicklung massentauglicher Anwendungen genutzt wird. Die Entwickler des Window-Managers Xmonad [2], verwenden Haskell Program Coverage (HPC [3]), um die Qualität ihrer durchgeführten Tests zu bewerten. HPC verwendet ein vergleichsweise einfaches Überdeckungs-Kriterium, überprüft nämlich, welche im Programm notierten Ausdrücke von der Berechnung angefordert wurden. Diese Form der Überdeckung hat zwei wesentliche Vorzüge:

1. sie lässt sich sehr effizient aufzeichnen und
2. sie lässt sich dem Benutzer anschaulich päsentieren – sowohl in Form prozentualer Zusammenfassungen als auch durch farbig angereicherte Programmtexte, in denen markiert ist, welche Ausdrücke nicht verwendet wurden.

Gelegentlich suggeriert HPC allerdings zu viel Vertrauen in durchgeführte Tests. So gibt es Beispiele, in denen HPC 100% Überdeckung bescheinigt, Fehler im Programm jedoch nach den durchgeführten Testläufen unentdeckt bleiben. Prinzipiell kann kein Überdeckungs-Kriterium die Fehlerfreiheit von Programmen garantieren. Es lohnt sich dennoch, solche Kriterien zu entwickeln, die möglichst gründliche Tests erfordern.

Im letzten Jahr haben wir ein Werkzeug zur Testfall-Generierung für die Programmiersprache Curry [4] vorgestellt, das eine minimale Anzahl von Testfällen anhand von Kontrollfluss-Überdeckungs-Information berechnet (vgl. [5]). Dieses Werkzeug verwendet den der Programmiersprache zu Grunde liegenden Auswertungsmechanismus *Narrowing*, der es ermöglicht, Berechnungen mit unbekannter

Information durchzuführen. Testfälle werden hier dadurch erzeugt, dass die zu testende Funktion mit unbekannten Eingaben aufgerufen wird und die Eingaben während der Berechnung gebunden werden.

Wir haben unser Werkzug um ein neues Kriterium für Datenfluss-Überdeckung in deklarativen Programmen mit bedarfsgesteuerter Auswertung erweitert (vgl. [6]). Anders als in imperativen Sprachen basiert unser Kriterium nicht auf Zuweisungen an und Verwendungen von Variablen sondern auf algebraischen Datentypen und Musteranpassung. Im Wesentlichen fließen Daten von Programmstellen, an denen ein Konstruktor notiert ist, zu Programmstellen, an denen ein Muster diesen Konstruktor verwendet. Eine andere Form von Datenfluss entsteht durch Funktionen höherer Ordnung: (partiell angewendete) Funktionen *fließen* zu ihrer Applikation.

Da im Allgemeinen unentscheidbar ist, welcher Datenfluss durch ein Programm möglich ist und ferner eine Approximation ohne eine Programmausführung nicht-triviale Programm-Analysen erfordert, haben wir eine Programmtransformation entwickelt, die ein beliebiges deklaratives Programm so anreichert, dass es neben seinem eigentlichen Ergebnis auch den durch die Berechnung induzierten Datenfluss berechnet. Das transformierte Programm ist selbst rein deklarativ, verwendet also keine Seiteneffekte zur Protokollierung des Datenflusses. Auch wird die Reihenfolge der Auswertung des ursprünglichen Programms nicht verändert. Die berechnete Überdeckung entspricht also der bedarfsgesteuerten Auswertung des ursprünglichen Programms.

Literatur

1. Peyton Jones, S., ed.: Haskell 98 Language and Libraries—The Revised Report. Cambridge University Press (2003)
2. Stewart, D., Sjanssen, S.: Xmonad. In: Haskell '07: Proceedings of the ACM SIGPLAN workshop on Haskell workshop, New York, NY, USA, ACM (2007) 119–119
3. Gill, A., Runciman, C.: Haskell program coverage. In: Haskell '07: Proceedings of the ACM SIGPLAN workshop on Haskell workshop, New York, NY, USA, ACM (2007) 1–12
4. Hanus, M., et al.: Curry: An integrated functional logic language (version 0.8.2). Available at URL `http://www.informatik.uni-kiel.de/~curry` (2006)
5. Fischer, S., Kuchen, H.: Systematic generation of glass-box test cases for functional logic programs. In: Proc. of the 9th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP 2007), ACM Press (2007) 75–89
6. Fischer, S., Kuchen, H.: Data-flow testing of declarative programs. In: Proc. of the 13th ACM SIGPLAN International Conference on Functional Programming (ICFP 2008), ACM Press (2008) to appear.

Empirischer Vergleich von Tools für das Testen deklarativer Programme

Herbert Kuchen

Institut für Wirtschaftsinformatik
Westfälische Wilhelms-Universität Münster

Für das Testen deklarativer Sprachen haben in letzter Zeit Tools wie QuickCheck starke Beachtung gefunden, die das Black-Box-Testen durch zufällig generierte Eingaben unterstützen. Varianten von QuickCheck wie LazySmallCheck versuchen, diesen Ansatz dadurch zu verbessern, dass gültige Testeingaben aus der zu testenden Eigenschaft abgeleitet werden und nicht mehr durch maßgeschneiderte Generatoren erzeugt werden müssen. Diese Ansätze zum Black-Box-Testen wurden empirisch mit dem Glass-Box-Testtool CyCoTest verglichen, das automatisch Testfälle erzeugt, die die Überdeckung des Kontroll- und/oder Datenflusses der zu testenden Module sicherstellen. Als Grundlage fÃijr den empirischen Vergleich wurden 10 Beispielanwendungen betrachtet, unter denen sowohl Basisdatenstrukturen wie AVL-Baum und Heap als auch klassische Algorithmen wie Strassen, Dijkstra und Kruskal sowie praktische Anwendungen wie die Berechnung des günstigsten Bahntickets sind. Als Ergebnis zeigt sich, dass CyCoTest knapp gefolgt von QuickCheck die meisten Fehler aufspüren kann, wobei QuickCheck hierzu deutlich weniger Zeit benötigt.

A Summary Function Model for the Validation of Interprocedural Analysis Results

Karsten Klohs ⋆1

Department of Computer Science
University Paderborn
Paderborn, Germany

Abstract. The validation of data flow results safely separates complex program analyses from the use of their results. This is useful in mobile code scenarios where a code consumer with limited computational capabilities wishes to enforce that the code exhibits properties described by the analysis results in order to check security policies or to safely apply optimisations to the program.
Any valid data flow solution has to solve the system of data flow equations which describes the data flow problem for the given program. The check that a given solution solves the system of data flow equations is more efficient than the original analysis because it requires a single pass over the equation system only. Essentially, the validation phase does not have to recompute the fix-point computations of the iterative data flow algorithm because a fix-point is given by the transmitted result.
This general principle can be applied to the validation of interprocedural summary functions which yields a validation strategy for interprocedural analysis results. An important requirement is that the consumer can compare summary functions with each other. We present a function model which provides a checkable order relation on summary functions as well as all other operations needed during the validation process. The model is based on expressions which establish the connection to the inducing data flow problem in a generic way.
The additional integration of function variables into the summary function model allows for the representation and late integration of analysis results from unavailable program parts. This gives rise to an incremental validation scenario, where the code consumer can subsequently validate analysis results of several software modules.

Validation, Data Flow Analysis, Interprocedural Analysis, Mobile Code

1 Problem Statement

According to the general validation principle, a given solution is valid with respect to a data flow problem if it solves the corresponding system of inequalities. A set of summary functions, invocation contexts and intermediate solutions make

⋆ taiko@upb.de, Fax: 0049 5251 60 6697

up an interprocedural solution. They can be validated by checking a more complex system of inequalities which specifies the interprocedural data flow problem. The application of the validation principle in the interprocedural scenario is discussed in depth in [Klo08].

The validation pass over the system of inequalities requires a data structure for summary functions which supports the following operations:

Function Application because the application of a summary function yields the intermediate results in the second phase of the analysis that computes data flow values.

Function Composition because function composition acts as the transfer function in the summary function computation phase.

Function Meet because it defines the semantics of join points.

Function Comparison to decide whether the inequalities that make up the equation system hold.

The original paper [Klo08] presents a model for summary functions that supports all of these operations efficiently. This contribution extends the definition of the summary function model by function variables and normalisation rules but postpones a thorough formal treatment of the new features which is about to appear in the subsequent publications.

2 Summary Function Model

A summary function ψ_{nm} maps the program state of point n to the state of point m and comprises the effects of all executions paths between these two points. Our summary function model comprises several modelling ideas:

1. The program state is decomposed into an *environment* - i.e. a mapping from an arbitrary set of data flow variables to data flow values. Dependencies between different pieces of the program state can be captured more precisely in such a fine-grained model.
2. The representation of a summary function consists of *data flow expressions* which reduce the summary function computation to operations supplied by the inducing data flow problem. The inducing data flow problem is defined by instruction-level transfer functions and a value lattice only. Thus, the summary function model "lifts" the definition of an intraprocedural analysis to an interprocedural analysis in a generic way.
3. The summary function model supplies a simple *comparison criterion*. The existence of an efficient comparison operation is vital for the validation process.
4. We define a set of *normalisation rules* which reduce a data flow expression to a canonical form. The reduction process corresponds to a partial evaluation of the expressions and it is essential to keep the size of the function representation under control.

5. Finally, the use of *function variables* in data flow expressions can model the potential effects of unavailable parts of the program. The function variables can either be substituted by summary functions as soon as the corresponding code becomes available or their effects can be safely approximated at any point in time. This additional degree of freedom supports an incremental validation scenario where the validator subsequently validates and integrates analysis results for classes which are loaded at different points in time.

The presentation of the summary function model is structured as follows. Firstly, we define the summary function model formally. Secondly, we briefly describe how the summary function model supports the required function operation like function meet, function composition, and function comparison. Subsequently, we consider several normalisation rules, that partially evaluate the data flow expressions and which lead to a compact normal form.

2.1 Model Definition

We start with a definition of the program state in terms of an environment which maps a set of arbitrary data flow variables to data flow values.

Definition 1 (Program State). *Let* $Var = \{x, y, z, \ldots\}$ *denote an arbitrary set of data flow variables and let* L *be the lattice of data flow values of an inducing analysis. Then we model the program state at a program point* m *by an* environment env_m*, i.e. a mapping from data flow variables to data flow values:*

$$env_m = \langle x \to x_m, y \to y_m, z \to z_m \ldots \rangle$$

Thus, the variable x refers to some data flow fact "x", while x_m denotes the value of the data flow fact x at program point m.

Our central modelling idea is to define the semantics of a summary function ψ_{mn} with respect to a single data flow fact x by the following equation

$$x_m = f^x_{nm}(env_n) \qquad \text{with} \qquad f^x_{mn}(env_n) = e^x_{nm}$$

The function f^x maps the program state at point n to the value of x at point m denoted by x_m. We call function f^x_{mn} *evaluation function* of x because the evaluation of the expression e^x_{mn} yields the result of the function. The data flow expression e^x_{mn} is the *defining expression* of f^x_{mn}.

It is important to observe that an evaluation function takes the *whole* environment as parameter but evaluates to a single data flow flow value for x. A *summary function* which manipulates the whole environment consists of a tuple of evaluation functions - one for each data flow fact. Evaluation functions and their defining data flow expressions are superscribed with the name of the data flow fact they evaluate to. Thus,

Definition 2 (Summary Function). *The summary function* ψ_{mn} *which maps the program state* env_m *at program point* m *to the program state* env_n *at point* n *is defined by*

$$\psi_{mn} = \langle f^x_{mn}, f^y_{mn}, f^z_{mn}, \ldots \rangle = \langle e^x_{mn}, e^y_{mn}, e^z_{mn}, \ldots \rangle$$

Figure 1 shows an example for the structure of the summary function and the environment in a small program where the program states consists of three local variables only.

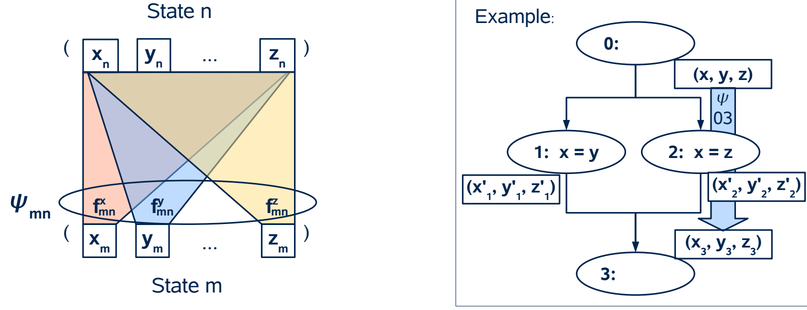

Fig. 1. Environments and the Summary Function Model

For the sake of simplicity, we abbreviate the environment $env_m = \langle x \rightarrow x_m, y \rightarrow y_m, \ldots \rangle$ by $(x_m, y_m, \ldots)$ and we notate function definitions which take the environment as parameter similarly to function definitions in a programming language, thus $\psi_{mn}(env) = \psi_{mn}(x, y, \ldots)$.

Many of the traditional analysis choose an direct correspondence between the variables of the program and the data flow variables to model the program state. However, data flow variables in the set Var can also refer to different program entities like available expressions, global fields etc.

The definition of data flow expressions, which define the evaluation functions completes the summary function model.

Definition 3. *Data Flow Expression A* data flow expression e *has the form*

$$e ::= c \quad | \quad x \quad | \quad e_1 \sqcap_L e_2 \quad | \quad t_i(e_1, \ldots, e_i) \quad | \quad s_i(e_1, \ldots, e_{|Var|})$$

where c is a data flow value of the inducing lattice, $x \in Var$ is a data flow variable, $t_i \in ET$ is an elementary transfer function of the inducing data flow problem, and $s_i \in FctVar$ is a free function variable.

This definition assumes that the inducing data flow problem is a meet-problem so that the safe approximation of two element is given by the greatest lower bound operator $\sqcap_L$. Join-problems are treated similarly.

The different kinds of data flow expressions deal with several aspects of the data flow problem in question:

Constant Expressions (c) do not depend on the input environment. They model the *generation* of data flow facts.

Data Flow Variables (x) refer to specific elements of the input environment. They can express value assignments etc. and act as insertion points during function application and function composition (see Section 2.2.

Safe Approximation Expressions ($\sqcap_L$) model the safe approximation of two data flow facts. This is vital do reduce the function meet to the meet-operator of the inducing lattice.

Elementary Transfer Functions (t_i) model more complex dependencies between data flow facts. They are required to increase the expressiveness of the model to data flow analyses like linear constant propagation. We will not discuss this part of the summary function model further in this contribution.

Function Variable Expressions (s_i) act as insertion point for summary functions that model the effects of code which becomes available later.

After the formal introduction of the summary function model, we continue with the definition of the operations on summary functions, before we briefly consider the reduction of data flow expressions to a compact canonical form.

2.2 Function Operations

The definition of the required function operations is straight-forward and can be summarized as follows:

Function Application → evaluation of expressions with variables substituted by parameter values
Function Composition → substitution of variables with subexpressions
Function Meet → meet of expressions
Function Comparison → comparison of subexpressions

Function Application and Composition Variable expressions give rise to the definition of function application and composition because they describe how a single piece of the output state - namely x - depends on the input state. The evaluation function $f^x_{mn}(x, y, z, \ldots) = e^x_{mn}$ can contain references to input state like x, y, or z. A concrete input state $env_m = (x_m, y_m, z_m)$ yields the value of x at program point n by substitution of variables in e^x_{mn} with the values in env_m , thus

$$\forall v \in Var, f^x_{mn}(x, y, z, \ldots) = e^x_{mn} : \qquad x_n = f^x_{mn}(x_m, y_m, z_m, \ldots) =_{def} e^x_{mn}|_{[v/v_m]}$$

Similarly, function composition reduces to substitution of variables, too. Consider the subsequent function application of $f_{11'}$ and $f_{22'}$ in Figure 2. The evaluation function $f^x_{11'}$ defines the state $x_{1'} = f^x_{11'}(x_1)$ in terms of x_1 while $f^x_{22'}(x_2) = e^x_{22'}$ defines the state $x_{2'}$ in terms of x_2. Furthermore, the states $x_{1'}$ and x_2 are equal, so that $x_2 = x_{1'} = f^x_{11'}(x_1) = e^x_{11'}$. Consequently, the defining expression $e^x_{11'}$ can substitute x_2 in $e^x_{22'}$. This yields a defining expression $e^x_{12'}$ which describes the dependency of $x_{2'}$ to the input state x_1. Thus,

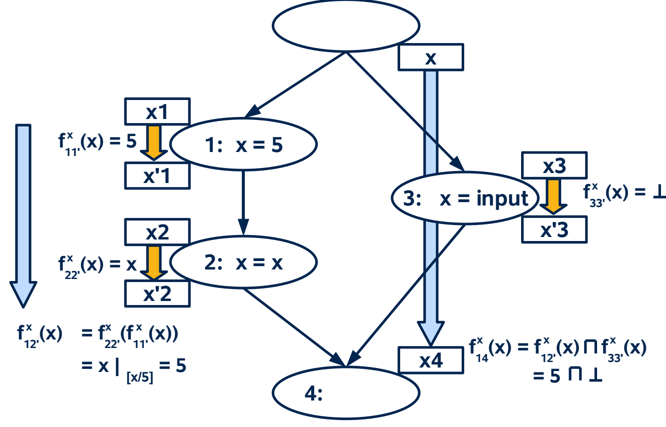

Fig. 2. Evaluation Functions

$$f^x_{lm}(x) = e^x_{lm}\ ,\ f^x_{mn}(x) = e^x_{mn}:$$
$$f^x_{ln} = f^x_{mn} \circ f^x_{lm} =_{def} f^x_{mn}(e^x_{lm}) = e^x_{mn}|_{[x/e^x_{lm}]} = e^x_{ln}$$

Essentially, the substitution removes the intermediate state at the point between the two functions. The identity expression at point 3 effectively propagates the defining expressions from point 1. Constant expressions stop the propagation because they do not contain variables that can be substituted. However, they generate a new data flow information by a constant expression.

Function Meet and the Order Relation of Functions The flow of control merges at join points in the program. After the join point only those data flow facts remain valid which are valid on all paths immediately before the join point. This is captured by the safe approximation operator $\sqcap_L$ of the inducing data flow lattice. The meet of summary functions reduces to the meet of expressions. Consider the situation in the example program in Figure 2 where two summary functions map the input state x to the two points $x_{2'}$ and $x_{3'}$ immediately before the join point denoted by x_4. The meet of these two functions maps the input state directly to the state at the join point. This state is defined by the conservative approximation of the predecessor states $x_4 = x_{2'} \sqcap_L x_{3'}$ which in turn are defined by the expressions in $f^x_{12'}$ and $f^x_{33'}$ respectively. Thus, the meet of these expressions captures the semantics of the join point and defines the function meet:

$$f^x_{14} = f^x_{12'} \sqcap_\psi f^x_{33'} =_{def} e^x_{12'} \sqcap_E e^x_{33'} = 5 \sqcap_L \perp$$

The definition of a meet operation always gives rise to the definition of an order relation because $x \sqcap y = y \Leftrightarrow x \sqsupseteq y$. Accordingly, the meet of data flow expressions leads to a simple criteria to decide the order relation of expressions.

Theorem 1 (Order Relation on Expressions). *An expression e_1 safely approximates an expression e_2 if it contains strictly more subexpressions than e_2. Two expressions are equal if they contain exactly the same subexpressions.*

Functions are in order relation if their defining expressions are in order relation.

3 Normalisation Rules

The simple definition of the meet of expression compares two expressions purely syntactically. As a consequence, *semantically* equivalent expressions like $4 \sqcap_L 3$ and $\perp$ are *not* considered to be equal. The meet of these expressions yields

$$(4 \sqcap_L 3) \sqcap_E \perp = 4 \sqcap_L 3 \sqcap_L \perp$$

Thus, the result expressions tend to be larger than necessary. However, it is possible to define simplifications - e.g. folding of constant expressions - which lead to a much more compact representation.

An expression in normal form consists of the conservative approximation expression of a single constant value, data flow variables, and function application expressions where each function occurs only once and whose parameter expressions are also in normal form.

The following reduction rules lead to this normal form:

Constant Folding (CF)

$$c_1 \sqcap_L c_2 \xrightarrow{CF} c_3 \qquad \text{with} \qquad c_3 = c_1 \sqcap_L c_2$$

The constant folding reduction replaces constant terms by their lower bound and it ensures that a single constant will remain on the outermost level of the nesting structure of each expression.

Duplicate Variable Removal (VAR)

$$x \sqcap_L x \xrightarrow{VAR} x$$

The VAR-reduction reduces the occurrences of a single variable to a single representative. It is justified by the fact that the conservative approximation operator $\sqcap_L$ is reflexive.

Bottom Shortcut (BSC)

$$e \sqcap_L \bot \xrightarrow{BSC} \bot$$

The BSC-reduction exploits the special status of the least element $\bot$ in the inducing lattice. This element represents the loss of all information. No matter to which concrete lattice element the expression e evaluates, the final result of the conservative approximation with $\bot$ will always yield $\bot$. Therefore, the original compound expression can be represented much more efficiently by $\bot$ which is known to be the result of any possible evaluation.

The fine-grained representation is vital for the effectiveness of the BSC-reduction because it is much more likely that data flow information is lost for a single variable than for the whole program state.

Push Out Upper Bound (POUB)

If $\quad [t(p)]|_{[x_i:=\top, s_i(e)=\top]} \sqcap c_{old} = c_{new} \sqsubset c_{old} \quad$ then $\quad t(p) \sqcap c_{old} \xrightarrow{POUB} t(p) \sqcap c_{new}$

The intuition of the POUB-reduction can be summarised as follows: even though we do not know the precise semantics of elementary transfer functions, we can still determine an *upper bound* for the expression $t(p)$. The reason is that the substitution of all variable occurrences in the parameter expression p leads to an upper bound for this expression and the result of the function application to such an upper bound leads is an upper bound of the function due to the monotony of t.

The POUB-reduction does *not* loose any precision, even though the constant term may be replaced by a weaker one. The POUB-reduction determines the *best* possible result for the evaluation of t. Thus, the final result can only be weaker than the new value of the constant term.

It's main purpose of the rule is to enable additional BSC-reductions. For example, consider an elementary transfer function which maps the most pessimistic element $\bot$ to itself - which is quite often the case. Furthermore, assume that the substitution of all variables in the parameter expressions leads to the most pessimistic element. Then,

$$\begin{aligned} t(e|_{[x_i:=\top, s_i(e)=\top]}) = t(\bot) \sqcap c &\xrightarrow{POUB} t(\bot) \sqcap \bot \\ &\xrightarrow{BSC} \bot \end{aligned}$$

Distributivity (DSTR)

$$\begin{aligned} t_i(p_1) \sqcap_L t_i(p_2) &\xrightarrow{DSTR} t_i(p_1 \sqcap_L p_2) \\ s_i(p_1) \sqcap_L s_i(p_2) &\xrightarrow{DSTR} s_i(p_1 \sqcap_L p_2) \end{aligned}$$

The distributivity rule ensures that each normal form has a single application of a specific function on each level of the nested expression structure. Furthermore, it enables additional normalisations of the combined parameter expressions.

Discussion The final comparison criterion for data flow expressions is similar to the criterion stated at the end of Section 2: A data flow expression e_1 is weaker than an expression e_2 if *its normal form* contains strictly more subexpressions than the *normal form* of e_2. The most pessimistic expression $\perp$ plays a special role because it represents the expression which consists of all potential subexpression.

The normalisation rules lead to a normal form which is unique. Furthermore, the order relation of expressions directly leads to the definition of an appropriate order relation on summary functions. The reason is that weaker expressions can only evaluate to weaker results which is the vital property of the definition of weaker summary functions. Due to space limitations we omit the proofs in this contribution.

One of the most important properties of the normalisation rules is, that they restrict the "width" of a data flow expressions, because each variable, elementary transfer function and function variable can occur only once on each level. Furthermore, the BSC-reduction reduces the size of expressions significantly, whenever the analysis in question yields safe lower bound. Examples include constant propagation which often compute that a value cannot be constant and type inference analysis for statically typed languages which can use the declared type as a safe lower bound.

However, the "depth" of the expressions is not limited yet, because function expression can contain arbitrary expressions as parameter expressions. This problem can solved in two different ways. Firstly, we can limit the maximum nesting depth of an expression. This approach is somehow related to the call-string approach, when it is restricted to call strings of a fixed length [1]. This restriction also limits the number of methods on the call stack which are considered for the summary function computation. The second way to deal with the nesting depth is to take specific properties of the inducing data flow analysis into account. The termination of the inducing analysis is usually guaranteed by the fact that the data flow lattice is finite or that at least the elementary transfer function yield a fix-point after a finite number of steps. As a consequence, the interprocedural analysis can also be restricted to a finite number of nested function applications because a further nesting cannot contribute to the final result anymore. The difference is that the second approach drops dependencies to further calls which maintains the precision of the result while the first approach safely approximates dependencies to further calls which results in a loss of precision. However, the second approach requires problem specific knowledge while the first one can be applied to any analysis.

4 Missing Parts of the Program

The integration of function variables into the expression model enables the deferred integration of missing summary functions. A function variable can be

[1] This is the usual strategy to guarantee the termination of the call-string approach for arbitrary analyses

substituted as soon as the corresponding summary function becomes available. This way it is possible to ship analysis results, which constitute a partial solution of the analysis problem . At the same time, the substitution of summary functions maintains the precision of the result. In contrast, most analysis which have to deal with unknown program parts safely approximate the potential effects and accept the loss of precision.

Interestingly, the corresponding result of such overly conservative analysis can be derived from the expression representation at any point in time: the validator just have to substitute the remaining function variables by the most pessimistic element of the inducing lattice and evaluate the result expression.

Even more importantly, function variable expressions are already subject to the normalisation process because they are first class elements of the expression model. As a consequence, the normalisation process has the potential to remove dependencies to unavailable parts of the program ahead of time. The most prominent situation occurs at join points. Assume that the analysis infers that a variable x does not contain a constant value on one path and that the value of the variable depends on the result of an unknown method call on the other path, then

$$e^x = \bot \sqcap s_{method_i}(\ldots) \stackrel{BSC}{\longrightarrow} \bot$$

This directly captures the intuition, that the unavailable method i cannot change the result for x anymore, because the analysis has lost all precision on a known path already.

The integration of function variables as first class values in the summary function model separates our approach from existing approaches to incomplete program analysis. Rountev [RSX08] recently proposed a extension of the graph-based interprocedural analysis model of Reps, Sagiv and Horwitz [RHS95], [SRH96]. This extension combines the graph-representation of summary functions with the stepwise construction of summary functions developed for component-level analysis [RMR03]. However, this approach cannot drop dependencies to unavailable calls ahead of time. Furthermore, the integration of function variables into the expression model preserves of the validation principle which is a problem that has not been considered by any other approach we are aware of.

References

Klo08. Karsten Klohs. A summary function model for the validation of interprocedural analysis results. In *Proceedings of the 7th International Workshop on Compiler Optimization meets Compiler Verification, COCV'08*. Electronic Notes in Theoretical Computer Science (ENTCS), Elsevier, (to appear), 2008.

RHS95. Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 49–61, New York, NY, USA, 1995. ACM Press.

RMR03. Atanas Rountev, Ana Milanova, and Barbara G. Ryder. Fragment class analysis for testing of polymorphism in java software. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 210–220. IEEE Computer Society, 2003.

RSX08. Atanas Rountev, Mariana Sharp, and Guoqing Xu. Ide dataflow analysis in the presence of large object-oriented libraries. In Laurie Hendren, editor, *Proceedings of the 17th International Conference on Compiler Construction*, volume 4959/2008, pages 53–68. Springer-Verlag, 2008.

SRH96. Mooly Sagiv, Thomas Reps, and Susan Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. In *TAPSOFT '95: Selected papers from the 6th international joint conference on Theory and practice of software development*, pages 131–170, Amsterdam, The Netherlands, 1996. Elsevier Science Publishers B. V.

Parametricity for Haskell with Imprecise Error Semantics
(Extended Abstract)

Florian Stenger* and Janis Voigtländer

Institut für Theoretische Informatik
Technische Universität Dresden
01062 Dresden, Germany

{stenger,voigt}@tcs.inf.tu-dresden.de

Abstract. Error raising, propagation, and handling in the functional programming language Haskell can be imprecise in the sense that a language implementation's choice of local evaluation order, and optimising transformations to apply, may influence which of a number of potential failure events hidden somewhere in a program is actually triggered. While this has pragmatic advantages from an implementation point of view, it also complicates the meaning of programs and thus requires extra care when reasoning about them. The proper semantic setup is one in which every erroneous value represents a whole set of potential (but not arbitrary) failure causes [PRH+99]. The associated propagation rules are somewhat askew to standard notions of program flow and value dependence. As a consequence, standard reasoning techniques are cast into doubt, and rightly so. We study this issue for one such reasoning technique, namely the derivation of (equational and inequational) free theorems from polymorphic types [Rey83,Wad89]. Continuing earlier work [JV04], we revise and extend the foundational notion of relational parametricity, as well as further material required to make it applicable. More generally, we believe that our new development and proofs help direct the way for incorporating further and other extensions and semantic features that deviate from the "naive" setting in which reasoning about Haskell programs often takes place.

1 Introduction

Functional languages come with a rich set of conceptual tools for reasoning about programs. For example, structural induction and equational reasoning tell us

* This author was supported by the DFG under grant VO 1512/1-1.

that the standard Haskell functions

$$\begin{array}{l} \mathit{takeWhile} :: (\alpha \to \mathsf{Bool}) \to [\alpha] \to [\alpha] \\ \mathit{takeWhile}\ p\ [\,] \qquad = [\,] \\ \mathit{takeWhile}\ p\ (x : y)\ \mid\ p\ x \qquad = x : \mathit{takeWhile}\ p\ y \\ \qquad\qquad\qquad\qquad \mid\ \mathit{otherwise} = [\,] \end{array}$$

$$\begin{array}{l} \mathit{map} :: (\alpha \to \beta) \to [\alpha] \to [\beta] \\ \mathit{map}\ h\ [\,] \qquad = [\,] \\ \mathit{map}\ h\ (x : y) = h\ x : \mathit{map}\ h\ y \end{array}$$

satisfy the following law for appropriately typed p, h, and l:

$$\mathit{takeWhile}\ p\ (\mathit{map}\ h\ l) = \mathit{map}\ h\ (\mathit{takeWhile}\ (p \circ h)\ l)\,. \tag{1}$$

But programming language reality can be a tough game, leading to unexpected failures of such near-obvious laws. For example, [PRH+99] proposes a design for error handling based on a certain degree of impreciseness. The major implementations GHC and Hugs (as well as one distribution of the language Clean) have integrated this design years ago. However, the attendant semantics betrays law (1) to be wrong. An instantiation showing this is $p = \mathit{null}$, $h = \mathit{tail}$, and $l = [[i] \mid i \leftarrow [1..(\mathit{div}\ 1\ 0)]]$ (or any other immediately failing expression of type list-of-lists), where

$$\begin{array}{ll} \mathit{null} :: [\alpha] \to \mathsf{Bool} & \mathit{tail} :: [\alpha] \to [\alpha] \\ \mathit{null}\ [\,] \qquad = \mathsf{True} & \mathit{tail}\ [\,] \qquad = \mathit{error}\ \text{“Prelude.tail: empty list”} \\ \mathit{null}\ (x : y) = \mathsf{False} & \mathit{tail}\ (x : y) = y \end{array}$$

are standard Haskell functions as well. The problem with (1) now is that its left-hand side yields exactly the “divide by zero”-error coming from l, whereas its right-hand side may also yield the “Prelude.tail: empty list”-error. This is so due to the semantics of pattern-matching in the design of [PRH+99] (and also [MLP99]). In short, it prescribes that when pattern-matching on an erroneous value as scrutinee, not only are any errors associated with it propagated, but also are the branches of the pattern-match investigated in “error-finding mode” to detect any errors that may arise there independently of the scrutinee. This is done in order to give the language implementation more freedom in arranging computations, thus allowing more transformations on the code prior to execution. But here it means that when *takeWhile* (*null* ∘ *tail*) encounters an erroneous value, also $(\mathit{null} \circ \mathit{tail})\ x$ is evaluated, with x bound to a special value $\mathit{Bad}\ \emptyset$ that exists only to trigger the error-finding mode. And indeed, the application of *tail* on that x raises the “Prelude.tail: empty list”-error, which is propagated by *null* and then unioned with the “divide by zero”-error from l. In contrast, *takeWhile null* on an erroneous value does not add any further errors, because the definition of *null* raises none. And, on both sides of (1), *map h* only ever propagates, but never introduces errors.

Thus, if we do not want to take the risk of introducing previously nonexistent errors, we cannot use (1) as a transformation from left to right, even though this

might have been beneficial (by bringing p and h together for further analysis or for subsequent transformations potentially improving efficiency). The supposed semantic equivalence simply does not hold. Note that this does not necessarily mean that for a given language implementation we will always, or ever, see different errors on the left- and right-hand sides of (1). After all, for the example instantiation above the semantics prescribes that the right-hand side may yield either of the two errors in question, so for a given interpreter or compiler it might very well happen that always the same as on the left-hand side appears. But that would be pure coincidence on which we cannot rely. If all the guarantee we have is that the language implementation builds on the semantics of [PRH$^+$99], then we have to accept that the arbitration between the two potential errors may in principle vary with set of flags, time of day, phase of the moon, and so on. Impreciseness in the semantics has its price, and if we are not ready to abandon the overall design (which would be tantamount to taking away considerable freedom from language implementors), then we better learn how to cope with it when reasoning about programs.

The above discussion regarding a concrete instantiation of p, h, and l gives negative information only, namely that (1) may break down in some cases. It does not provide any positive information about conditions on p, h, and l under which (1) actually *is* a semantic equivalence. Moreover, it is relative to the particular definition of *takeWhile* given at the very beginning, whereas laws like (1) are often derived more generally as *free theorems* [Rey83,Wad89] from types alone, without considering concrete definitions. In the study reported here we undertake to develop the theory of free theorems for Haskell with imprecise error semantics. This continues earlier work [JV04] for Haskell with all potential error causes (including nontermination) conflated into a single erroneous value $\bot$. That earlier work gives that in this setting (1) is a semantic equivalence provided $p \neq \bot$ and h is strict and total in the sense that $h\ \bot = \bot$ and for every $x \neq \bot$, $h\ x \neq \bot$. The task set ourselves involves finding the right generalisations of such conditions for a setting in which not all errors are equal. Questions like the following ones arise:

- From which erroneous values should p be different?
- For strictness, is it enough that h preserves the least element $\bot$, which in the design of [PRH$^+$99] denotes the union of all error causes, including nontermination?
- Or do we need that also every other erroneous value (denoting a collection of only some potential error causes, maybe just a singleton set) is mapped to an erroneous one? To the same one? Or to $\bot$?
- For totality, is it enough that "non-$\bot$ goes to non-$\bot$"?
- Does this allow "non-$\bot$ goes to non-$\bot$ but erroneous"?
- Or do we need "nonerroneous goes to nonerroneous"?

Fortunately, we are not left groping in the dark. Our investigation can be very much goal-directed by studying proof cases of the *(relational) parametricity theorem*, which is the foundation for all free theorems, and trying to adapt the

proof to the imprecise error setting. This leads us to discover, among other formal details and ingredients, the appropriate generalised conditions sought above (first as restrictions on relations, then specialised to the function level). In fact, we think that beside the results we provide for the imprecise error setting, our study can also serve as a guide on how to go about extending relational parametricity to new language features and semantic designs in general. We have established both equational and inequational parametricity theorems, including one for the refinement order of [MLP99]. And while we do not deal with error recovery through exception handling in the IO monad, we have made some initial steps into the realm of exceptions as first class citizens by integrating a primitive (Haskell's *mapException*) that allows manipulating already raised errors (respectively, their descriptive arguments) from inside the language.

2 Standard Parametricity

We start out from a standard denotational semantics for a polymorphic lambda-calculus that corresponds to Haskell without distinguishing error causes, i.e., with only one erroneous value $\bot$.

The syntax of types and terms is given as follows:

$$\begin{aligned}\tau &::= \alpha \mid \mathsf{Int} \mid [\tau] \mid \tau \to \tau \mid \forall\alpha.\tau \\ t &::= x \mid n \mid t + t \mid [\,]_\tau \mid t : t \mid \mathbf{case}\ t\ \mathbf{of}\ \{[\,] \to t\,;\ x : x \to t\} \mid \\ &\quad\ \ \lambda x : \tau.t \mid t\ t \mid \Lambda\alpha.t \mid t\ \tau \mid \mathbf{fix} \mid \mathbf{let!}\ x = t\ \mathbf{in}\ t\,,\end{aligned}$$

where α ranges over type variables, x over term variables, and n over the integers. We include integers and lists as representatives for numeric types and algebraic datatypes, and addition as an exemplary numeric operation. Note that the calculus is explicitly typed and that type abstraction and application are explicit in the syntax as well. General recursion is captured via a fixpoint combinator, while selective strictness (à la *seq*) is provided via a strict-let construct.

Fig. 1 gives the typing rules for our calculus. Standard conventions apply here. In particular, typing environments Γ take the form $\alpha_1, \ldots, \alpha_k, x_1 : \tau_1, \ldots, x_l : \tau_l$ with distinct α_i and x_j, where all free variables occurring in a τ_j have to be among the listed type variables.

For example, the function *map* can be defined as the following term and then satisfies $\vdash map : \tau$, where $\tau = \forall\alpha.\forall\beta.(\alpha \to \beta) \to [\alpha] \to [\beta]$:

$$\begin{aligned}&\mathbf{fix}\ \tau\ (\lambda m : \tau.\Lambda\alpha.\Lambda\beta.\lambda h : \alpha \to \beta.\lambda l : [\alpha]. \\ &\qquad \mathbf{case}\ l\ \mathbf{of}\ \{[\,] \to [\,]\,;\ x : y \to (h\ x) : (m\ \alpha\ \beta\ h\ y)\})\,.\end{aligned}$$

The denotational semantics interprets types as *pointed complete partial orders* (for short, *pcpos*; least element always denoted $\bot$). The definition in Fig. 2, assuming θ to be a mapping from type variables to pcpos, is pretty standard. The operation $lift_\bot$ takes a complete partial order, adds a new element $\bot$ to the carrier set, defines this new $\bot$ to be below every other element, and leaves the ordering otherwise unchanged. To avoid confusion, the original elements are

$$\Gamma, x : \tau \vdash x : \tau \qquad \Gamma \vdash n : \mathsf{Int} \qquad \Gamma \vdash [\,]_\tau : [\tau] \qquad \Gamma \vdash \mathbf{fix} : \forall \alpha.(\alpha \to \alpha) \to \alpha$$

$$\frac{\Gamma \vdash t_1 : \mathsf{Int} \qquad \Gamma \vdash t_2 : \mathsf{Int}}{\Gamma \vdash (t_1 + t_2) : \mathsf{Int}} \qquad \frac{\Gamma \vdash t_1 : \tau \qquad \Gamma \vdash t_2 : [\tau]}{\Gamma \vdash (t_1 : t_2) : [\tau]} \qquad \frac{\alpha, \Gamma \vdash t : \tau}{\Gamma \vdash (\Lambda \alpha.t) : \forall \alpha.\tau}$$

$$\frac{\Gamma \vdash t : [\tau_1] \qquad \Gamma \vdash t_1 : \tau_2 \qquad \Gamma, x_1 : \tau_1, x_2 : [\tau_1] \vdash t_2 : \tau_2}{\Gamma \vdash (\mathbf{case}\ t\ \mathbf{of}\ \{[\,] \to t_1\ ;\ x_1 : x_2 \to t_2\}) : \tau_2}$$

$$\frac{\Gamma, x : \tau_1 \vdash t : \tau_2}{\Gamma \vdash (\lambda x : \tau_1.t) : \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash t_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash t_2 : \tau_1}{\Gamma \vdash (t_1\ t_2) : \tau_2}$$

$$\frac{\Gamma \vdash t : \forall \alpha.\tau_1}{\Gamma \vdash (t\ \tau_2) : \tau_1[\tau_2/\alpha]} \qquad \frac{\Gamma \vdash t_1 : \tau_1 \qquad \Gamma, x : \tau_1 \vdash t_2 : \tau_2}{\Gamma \vdash (\mathbf{let!}\ x = t_1\ \mathbf{in}\ t_2) : \tau_2}$$

Fig. 1. Typing Rules.

tagged, i.e., $\mathit{lift}_\bot\, S = \{\bot\} \cup \{\lfloor s \rfloor \mid s \in S\}$. The complete partial order lifted in the definition of $[\![\mathsf{Int}]\!]_\theta$ is the flat one without ordering between integers. For list types, prior to lifting, $[\,]$ is only related to itself, while the ordering between "$- : -$"-values is component-wise. The recursive occurrence of $[\![[\tau]]\!]_\theta$ is resolved by taking the greatest fixpoint, thus providing for infinite lists. The function space lifted in the definition of $[\![\tau_1 \to \tau_2]\!]_\theta$ is the one of monotonic and continuous maps between $[\![\tau_1]\!]_\theta$ and $[\![\tau_2]\!]_\theta$, ordered point-wise. Finally, polymorphic types are interpreted as appropriately lifted sets of functions from pcpos to values restricted as in the last line of Fig. 2, and again ordered point-wise (i.e., $g_1 \sqsubseteq g_2$ iff for every pcpo D, $g_1\ D \sqsubseteq g_2\ D$).

$$\begin{aligned}
[\![\alpha]\!]_\theta &= \theta(\alpha) \\
[\![\mathsf{Int}]\!]_\theta &= \mathit{lift}_\bot\, \{\ldots, -2, -1, 0, 1, 2, \ldots\} \\
[\![[\tau]]\!]_\theta &= \mathit{lift}_\bot\, (\{[\,]\} \cup \{a : b \mid a \in [\![\tau]\!]_\theta,\, b \in [\![[\tau]]\!]_\theta\}) \\
[\![\tau_1 \to \tau_2]\!]_\theta &= \mathit{lift}_\bot\, \{f : [\![\tau_1]\!]_\theta \to [\![\tau_2]\!]_\theta\} \\
[\![\forall \alpha.\tau]\!]_\theta &= \mathit{lift}_\bot\, \{g \mid \forall D \text{ pcpo. } (g\ D) \in [\![\tau]\!]_{\theta[\alpha \mapsto D]} \setminus \{\bot\}\}
\end{aligned}$$

Fig. 2. Standard Semantics of Types.

The semantics of terms in Fig. 3 is now also standard. It uses λ for denoting anonymous functions, and the following two operators:

$$h\ \$\ a = \begin{cases} f\ a & \text{if } h = \lfloor f \rfloor \\ \bot & \text{if } h = \bot \end{cases} \qquad h\ \$\$\ D = \begin{cases} g\ D & \text{if } h = \lfloor g \rfloor \\ \bot & \text{if } h = \bot\,. \end{cases}$$

The expression $\bigsqcup((h\,\$)^i\ \bot)$ in the definition for **fix** means the supremum of the chain $\bot \sqsubseteq h\ \$\ \bot \sqsubseteq h\ \$\,(h\ \$\ \bot) \cdots$. Altogether, we have that if $\Gamma \vdash t : \tau$ and $\sigma(x) \in [\![\tau']\!]_\theta$ for every $x : \tau'$ occurring in Γ, then $[\![t]\!]_{\theta,\sigma} \in [\![\tau]\!]_\theta$.

The key to parametricity results is the definition of a family of relations by induction on a calculus' type structure. The appropriate such *logical relation* for

$$
\begin{array}{ll}
[\![x]\!]_{\theta,\sigma} & = \sigma(x) \\
[\![n]\!]_{\theta,\sigma} & = \lfloor n \rfloor \\
[\![t_1 + t_2]\!]_{\theta,\sigma} & = \begin{cases} \lfloor n_1 + n_2 \rfloor & \text{if } [\![t_1]\!]_{\theta,\sigma} = \lfloor n_1 \rfloor,\ [\![t_2]\!]_{\theta,\sigma} = \lfloor n_2 \rfloor \\ \bot & \text{otherwise} \end{cases} \\
[\![[\,]_\tau]\!]_{\theta,\sigma} & = \lfloor [\,] \rfloor \\
[\![t_1 : t_2]\!]_{\theta,\sigma} & = \lfloor [\![t_1]\!]_{\theta,\sigma} : [\![t_2]\!]_{\theta,\sigma} \rfloor \\
[\![\mathbf{case}\ t\ \mathbf{of}\ \{[\,] \to t_1\,;\ x_1 : x_2 \to t_2\}]\!]_{\theta,\sigma} & = \begin{cases} [\![t_1]\!]_{\theta,\sigma} & \text{if } [\![t]\!]_{\theta,\sigma} = \lfloor [\,] \rfloor \\ [\![t_2]\!]_{\theta,\sigma[x_1 \mapsto a,\, x_2 \mapsto b]} & \text{if } [\![t]\!]_{\theta,\sigma} = \lfloor a : b \rfloor \\ \bot & \text{if } [\![t]\!]_{\theta,\sigma} = \bot \end{cases} \\
[\![\lambda x : \tau.t]\!]_{\theta,\sigma} & = \lfloor \lambda a.[\![t]\!]_{\theta,\sigma[x \mapsto a]} \rfloor \\
[\![t_1\ t_2]\!]_{\theta,\sigma} & = [\![t_1]\!]_{\theta,\sigma}\ \$\ [\![t_2]\!]_{\theta,\sigma} \\
[\![\Lambda\alpha.t]\!]_{\theta,\sigma} & = \begin{cases} \lfloor \lambda D.[\![t]\!]_{\theta[\alpha \mapsto D],\sigma} \rfloor & \text{if } [\![t]\!]_{\theta[\alpha \mapsto \{\bot\}],\sigma} \neq \bot \\ \bot & \text{if } [\![t]\!]_{\theta[\alpha \mapsto \{\bot\}],\sigma} = \bot \end{cases} \\
[\![t\ \tau]\!]_{\theta,\sigma} & = [\![t]\!]_{\theta,\sigma}\ \$\$\ [\![\tau]\!]_\theta \\
[\![\mathbf{fix}]\!]_{\theta,\sigma} & = \lfloor \lambda D.\lfloor \lambda h. \bigsqcup ((h\,\$)^i\ \bot) \rfloor \rfloor \\
[\![\mathbf{let!}\ x = t_1\ \mathbf{in}\ t_2]\!]_{\theta,\sigma} & = \begin{cases} [\![t_2]\!]_{\theta,\sigma[x \mapsto a]} & \text{if } [\![t_1]\!]_{\theta,\sigma} = a \neq \bot \\ \bot & \text{if } [\![t_1]\!]_{\theta,\sigma} = \bot \end{cases}
\end{array}
$$

Fig. 3. Standard Semantics of Terms.

our current setting is defined in Fig. 4, assuming ρ to be a mapping from type variables to binary relations between pcpos. We use id_D to denote the identity relation on the pcpo D. The operation *list* takes a relation $\mathcal{R}$ and maps it to

$$list\,\mathcal{R} = \{(\bot,\bot),\ (\lfloor [\,] \rfloor, \lfloor [\,] \rfloor)\} \cup \{(\lfloor a : b \rfloor, \lfloor c : d \rfloor) \mid (a,c) \in \mathcal{R},\ (b,d) \in list\,\mathcal{R}\}\,,$$

where again the greatest fixpoint is taken. For two pcpos D_1 and D_2, $Rel(D_1, D_2)$ collects all relations between them that are *strict*, *continuous*, and *bottom-reflecting*. Strictness and continuity are just the standard notions, i.e., membership of the pair $(\bot,\bot)$ and closure under suprema. A relation $\mathcal{R}$ is bottom-reflecting if $(a,b) \in \mathcal{R}$ implies that $a = \bot$ iff $b = \bot$. The corresponding explicit condition on f and g in the definition of $\Delta_{\tau_1 \to \tau_2,\rho}$ serves the purpose of ensuring that bottom-reflectingness is preserved throughout the logical relation. Overall, reasoning like [JV04] then gives the following parametricity theorem.

Theorem 1 *If $\Gamma \vdash t : \tau$, then for every θ_1, θ_2, ρ, σ_1, and σ_2 such that*

- *for every α occurring in Γ, $\rho(\alpha) \in Rel(\theta_1(\alpha), \theta_2(\alpha))$, and*
- *for every $x : \tau'$ occurring in Γ, $(\sigma_1(x), \sigma_2(x)) \in \Delta_{\tau',\rho}$,*

we have $([\![t]\!]_{\theta_1,\sigma_1}, [\![t]\!]_{\theta_2,\sigma_2}) \in \Delta_{\tau,\rho}$.

$$\begin{array}{ll}
\Delta_{\alpha,\rho} & = \rho(\alpha) \\
\Delta_{\mathsf{Int},\rho} & = id_{lift_\perp\{\ldots,-2,-1,0,1,2,\ldots\}} \\
\Delta_{[\tau],\rho} & = list\ \Delta_{\tau,\rho} \\
\Delta_{\tau_1\to\tau_2,\rho} & = \{(f,g) \mid f = \perp \text{ iff } g = \perp, \forall (a,b) \in \Delta_{\tau_1,\rho}.\ (f\ \$\ a, g\ \$\ b) \in \Delta_{\tau_2,\rho}\} \\
\Delta_{\forall\alpha.\tau,\rho} & = \{(u,v) \mid \forall D_1, D_2 \text{ pcpos}, \mathcal{R} \in Rel(D_1,D_2). \\
 & \qquad (u\ \$\$\ D_1, v\ \$\$\ D_2) \in \Delta_{\tau,\rho[\alpha\mapsto\mathcal{R}]}\}
\end{array}$$

Fig. 4. Standard Logical Relation.

3 Imprecise Error Semantics

We want to treat different failure causes as semantically different, rather than conflating them into a single erroneous value $\perp$. To this end, we add a new term-former **error** with typing rule $\Gamma \vdash \textbf{error} : \forall\alpha.\mathsf{Int} \to \alpha$.

Our treatment of errors shall be that of [PRH+99] and [MLP99]. Their main innovation, and the reason for calling the semantics "imprecise", is the use of *sets* of possible failure causes. Formally, let

$$\mathcal{E} = \{\mathsf{ErrorCall}\ n \mid n \in \{\ldots, -2, -1, 0, 1, 2, \ldots\}\} \tag{2}$$

and $\mathcal{E}^{nt} = \{\mathsf{NonTermination}\} \cup \mathcal{E}$, where NonTermination and ErrorCall are (for now) only descriptive tags for use in the denotational semantics, but without direct syntactical counterparts in the underlying calculus. The set of all erroneous values is then

$$V_{err} = \{Bad\ e \mid e \in \mathcal{P}(\mathcal{E}) \cup \{\mathcal{E}^{nt}\}\}$$

and its elements are ordered by

$$Bad\ e \sqsubseteq Bad\ e' \ \text{ iff } \ e \supseteq e'\,. \tag{3}$$

[PRH+99] then replaces the standard operation $lift_\perp$ by

$$lift_{err}\ S = V_{err} \cup \{Ok\ s \mid s \in S\}\,.$$

The approximation order on such error-lifted complete partial orders (henceforth, for short, *elcpos*) is given by (3) on erroneous values, by taking over the order from S for nonerroneous values, and by mandating that $\perp = Bad\ \mathcal{E}^{nt}$ is below all, even nonerroneous, values, while otherwise erroneous and nonerroneous values are pairwise incomparable.

With these definitions in place, Fig. 5 should hold no surprises, as it is in complete analogy to Fig. 2. Of course, we now assume that θ maps to elcpos only. Some of the term semantics definitions in Fig. 6 are directly taken over from Fig. 3, modulo lifting via Ok rather than via $\lfloor - \rfloor$, and thus need not be further discussed here. The definitions of $[\![t_1\ t_2]\!]^{err}_{\theta,\sigma}$ and $[\![\textbf{fix}]\!]^{err}_{\theta,\sigma}$ are as in Fig. 3, but use the following variant of the operator \$:

$$h\ \$\ a = \begin{cases} f\ a & \text{if } h = Ok\ f \\ Bad\ (e \cup E(a)) & \text{if } h = Bad\ e\,, \end{cases} \quad \text{where} \quad E(a) = \begin{cases} \emptyset & \text{if } a = Ok\ v \\ e & \text{if } a = Bad\ e\,. \end{cases}$$

The crucial point here, taken over from [PRH+99], is that application of an erroneous function value incurs all potential failures of the argument as well. Also essentially taken over are the definitions of $[\![t_1 + t_2]\!]^{err}_{\theta,\sigma}$ and $[\![\mathbf{case}\ t\ \mathbf{of}\ \{[\,] \rightarrow t_1\,;\ x_1 : x_2 \rightarrow t_2\}]\!]^{err}_{\theta,\sigma}$, except that we do not check for overflow in the case of addition. To bring about erroneous values other than $\bot$ in the first place, we have the obvious definition of $[\![\mathbf{error}]\!]^{err}_{\theta,\sigma}$. The definitions of $[\![\Lambda\alpha.t]\!]^{err}_{\theta,\sigma}$ and $[\![t\ \tau]\!]^{err}_{\theta,\sigma}$ follow the corresponding ones in Fig. 3, but with the following variant of the operator \$\$:

$$h \ \$\$\ D = \begin{cases} g\ D & \text{if } h = Ok\ g \\ Bad\ e & \text{if } h = Bad\ e\,. \end{cases}$$

Finally, the definition of $[\![\mathbf{let!}\ x = t_1\ \mathbf{in}\ t_2]\!]^{err}_{\theta,\sigma}$ follows the one in Fig. 3, but similarly to the definition of $[\![\mathbf{case}\ t\ \mathbf{of}\ \{[\,] \rightarrow t_1\,;\ x_1 : x_2 \rightarrow t_2\}]\!]^{err}_{\theta,\sigma}$, and in line with the operational semantics of [MLP99], t_2 is evaluated in "error-finding mode" to contribute further potential failure causes in case t_1 is already erroneous.

$$\begin{aligned}
[\![\alpha]\!]^{err}_{\theta} &= \theta(\alpha) \\
[\![\mathsf{Int}]\!]^{err}_{\theta} &= \mathit{lift}_{err}\,\{\ldots, -2, -1, 0, 1, 2, \ldots\} \\
[\![[\tau]]\!]^{err}_{\theta} &= \mathit{lift}_{err}\,(\{[\,]\} \cup \{a : b \mid a \in [\![\tau]\!]^{err}_{\theta},\ b \in [\![[\tau]]\!]^{err}_{\theta}\}) \\
[\![\tau_1 \rightarrow \tau_2]\!]^{err}_{\theta} &= \mathit{lift}_{err}\,\{f : [\![\tau_1]\!]^{err}_{\theta} \rightarrow [\![\tau_2]\!]^{err}_{\theta}\} \\
[\![\forall\alpha.\tau]\!]^{err}_{\theta} &= \mathit{lift}_{err}\,\{g \mid \forall D \text{ elcpo. } (g\ D) \in [\![\tau]\!]^{err}_{\theta[\alpha\mapsto D]} \setminus V_{err}\}
\end{aligned}$$

Fig. 5. Error Semantics of Types.

4 Parametricity for Imprecise Error Semantics

In this extended abstract, we do not detail the formal development leading to an analogue of Theorem 1 for the imprecise error semantics. Rather, we just give the resulting definitions and formal statements here.

Definition 2 *A relation $\mathcal{R}$ is* error-strict *if $id_{V_{err}} \subseteq \mathcal{R}$.*

Definition 3 *A relation $\mathcal{R}$ is* error-reflecting *if $(a, b) \in \mathcal{R}$ implies that $E(a) = E(b)$ and $T(a) = T(b)$, where*

$$T(x) = \begin{cases} Ok & \text{if } x = Ok\ v \\ Bad & \text{if } x = Bad\ e \end{cases}$$

For given elcpos D_1 and D_2, let $Rel^{err}(D_1, D_2)$ collect all relations between them that are error-strict, continuous, and error-reflecting.

$$
\begin{array}{ll}
[\![x]\!]^{err}_{\theta,\sigma} & = \sigma(x) \\
[\![n]\!]^{err}_{\theta,\sigma} & = Ok\ n \\
[\![t_1 + t_2]\!]^{err}_{\theta,\sigma} = & \begin{cases} Ok\ (n_1 + n_2) & \text{if } [\![t_1]\!]^{err}_{\theta,\sigma} = Ok\ n_1,\ [\![t_2]\!]^{err}_{\theta,\sigma} = Ok\ n_2 \\ Bad\ (E([\![t_1]\!]^{err}_{\theta,\sigma}) \cup E([\![t_2]\!]^{err}_{\theta,\sigma})) & \text{otherwise} \end{cases} \\
[\![[\,]_\tau]\!]^{err}_{\theta,\sigma} & = Ok\ [\,] \\
[\![t_1 : t_2]\!]^{err}_{\theta,\sigma} & = Ok\ ([\![t_1]\!]^{err}_{\theta,\sigma} : [\![t_2]\!]^{err}_{\theta,\sigma}) \\
\multicolumn{2}{l}{[\![\mathbf{case}\ t\ \mathbf{of}\ \{[\,] \to t_1\,;\ x_1 : x_2 \to t_2\}]\!]^{err}_{\theta,\sigma} =} \\
\multicolumn{2}{l}{\quad \begin{cases} [\![t_1]\!]^{err}_{\theta,\sigma} & \text{if } [\![t]\!]^{err}_{\theta,\sigma} = Ok\ [\,] \\ [\![t_2]\!]^{err}_{\theta,\sigma[x_1 \mapsto a,\, x_2 \mapsto b]} & \text{if } [\![t]\!]^{err}_{\theta,\sigma} = Ok\ (a : b) \\ Bad\ (e \cup E([\![t_1]\!]^{err}_{\theta,\sigma}) \cup E([\![t_2]\!]^{err}_{\theta,\sigma[x_1 \mapsto Bad\,\emptyset,\, x_2 \mapsto Bad\,\emptyset]})) & \text{if } [\![t]\!]^{err}_{\theta,\sigma} = Bad\ e \end{cases}} \\
[\![\lambda x : \tau . t]\!]^{err}_{\theta,\sigma} & = Ok\ (\lambda a . [\![t]\!]^{err}_{\theta,\sigma[x \mapsto a]}) \\
[\![t_1\ t_2]\!]^{err}_{\theta,\sigma} & = [\![t_1]\!]^{err}_{\theta,\sigma}\ \$\ [\![t_2]\!]^{err}_{\theta,\sigma} \\
[\![\Lambda\alpha . t]\!]^{err}_{\theta,\sigma} & = \begin{cases} Ok\ (\lambda D . [\![t]\!]^{err}_{\theta[\alpha \mapsto D],\sigma}) & \text{if } [\![t]\!]^{err}_{\theta[\alpha \mapsto V_{err}],\sigma} = Ok\ v \\ Bad\ e & \text{if } [\![t]\!]^{err}_{\theta[\alpha \mapsto V_{err}],\sigma} = Bad\ e \end{cases} \\
[\![t\ \tau]\!]^{err}_{\theta,\sigma} & = [\![t]\!]^{err}_{\theta,\sigma}\ \$\$\ [\![\tau]\!]^{err}_{\theta} \\
[\![\mathbf{fix}]\!]^{err}_{\theta,\sigma} & = Ok\ (\lambda D . Ok\ (\lambda h . \bigsqcup ((h\,\$)^i\ \bot))) \\
[\![\mathbf{let!}\ x = t_1\ \mathbf{in}\ t_2]\!]^{err}_{\theta,\sigma} = & \begin{cases} [\![t_2]\!]^{err}_{\theta,\sigma[x \mapsto Ok\ v]} & \text{if } [\![t_1]\!]^{err}_{\theta,\sigma} = Ok\ v \\ Bad\ (e \cup E([\![t_2]\!]^{err}_{\theta,\sigma[x \mapsto Bad\,\emptyset]})) & \text{if } [\![t_1]\!]^{err}_{\theta,\sigma} = Bad\ e \end{cases} \\
[\![\mathbf{error}]\!]^{err}_{\theta,\sigma} & = Ok\ (\lambda D . Ok\ (\lambda a . \begin{cases} Bad\ \{\mathsf{ErrorCall}\ n\} & \text{if } a = Ok\ n \\ Bad\ e & \text{if } a = Bad\ e \end{cases}))
\end{array}
$$

Fig. 6. Error Semantics of Terms.

The new logical relation is defined in Fig. 7, where the versions of \$ and \$\$ from Section 3 rather than those from Section 2 are used, and where

$$
\begin{aligned}
list^{err}\,\mathcal{R} = {} & id_{V_{err}} \cup \{(Ok\ [\,], Ok\ [\,])\} \\
& \cup \{(Ok\ (a : b), Ok\ (c : d)) \mid (a, c) \in \mathcal{R},\ (b, d) \in list^{err}\,\mathcal{R}\}\,.
\end{aligned}
$$

We have proved the following analogue of Theorem 1.

Theorem 4 *If $\Gamma \vdash t : \tau$, then for every θ_1, θ_2, ρ, σ_1, and σ_2 such that*

- *for every α occurring in Γ, $\rho(\alpha) \in Rel^{err}(\theta_1(\alpha), \theta_2(\alpha))$, and*
- *for every $x : \tau'$ occurring in Γ, $(\sigma_1(x), \sigma_2(x)) \in \Delta^{err}_{\tau',\rho}$,*

we have $([\![t]\!]^{err}_{\theta_1,\sigma_1}, [\![t]\!]^{err}_{\theta_2,\sigma_2}) \in \Delta^{err}_{\tau,\rho}$.

Having established Theorem 4, we can use it to derive free theorems that hold with respect to the imprecise error semantics. When doing so, we typically want to specialise relations (arising from the quantification in the definition of $\Delta^{err}_{\forall\alpha.\tau,\rho}$) to functions. To this end, the following definition is useful. The notation $\emptyset$ is used for empty mappings from type or term variables to elcpos and values, respectively.

$$
\begin{aligned}
\Delta^{err}_{\alpha,\rho} &= \rho(\alpha)\\
\Delta^{err}_{\mathsf{Int},\rho} &= id_{lift_{err}\{\ldots,-2,-1,0,1,2,\ldots\}}\\
\Delta^{err}_{[\tau],\rho} &= list^{err}\,\Delta^{err}_{\tau,\rho}\\
\Delta^{err}_{\tau_1\to\tau_2,\rho} &= \{(f,g)\mid T(f)=T(g),\\
&\qquad\qquad \forall(a,b)\in\Delta^{err}_{\tau_1,\rho}.\ (f\ \$\ a, g\ \$\ b)\in\Delta^{err}_{\tau_2,\rho}\}\\
\Delta^{err}_{\forall\alpha.\tau,\rho} &= \{(u,v)\mid \forall D_1,D_2\ \text{elcpos}, \mathcal{R}\in Rel^{err}(D_1,D_2).\\
&\qquad\qquad (u\ \$\$\ D_1, v\ \$\$\ D_2)\in\Delta^{err}_{\tau,\rho[\alpha\mapsto\mathcal{R}]}\}
\end{aligned}
$$

Fig. 7. Logical Relation for Imprecise Error Semantics.

Definition 5 *A term h with* $\vdash h : \tau_1 \to \tau_2$ *and* $[\![h]\!]^{err}_{\emptyset,\emptyset} = Ok\ f$ *is*

- error-strict *if* $f\ a = a$ *for every* $a \in V_{err}$*, and*
- error-total *if* $T(f\ a) = Ok$ *for every* $a \in [\![\tau_1]\!]^{err}_{\emptyset} \setminus V_{err}$.

An h with $T([\![h]\!]^{err}_{\emptyset,\emptyset}) = Bad$ *is neither error-strict nor error-total.*

For example, *null* is error-strict and error-total, while *tail* is neither of both.

Taking up the introductory example, we can then derive the following free theorem.

Theorem 6 *Let t be a term with* $\vdash t : \forall\alpha.(\alpha \to \mathsf{Bool}) \to [\alpha] \to [\alpha]$.[1] *Let h be a term with* $\vdash h : \tau_1 \to \tau_2$ *that is error-strict and error-total. Let p be a term with* $\vdash p : \tau_2 \to \mathsf{Bool}$ *and* $T([\![p]\!]^{err}_{\emptyset,\emptyset}) = Ok$*. Then for every term l with* $\vdash l : [\tau_1]$,

$$[\![map\ \tau_1\ \tau_2\ h\ (t\ \tau_1\ (\lambda x : \tau_1.p\ (h\ x))\ l)]\!]^{err}_{\emptyset,\emptyset} = [\![t\ \tau_2\ p\ (map\ \tau_1\ \tau_2\ h\ l)]\!]^{err}_{\emptyset,\emptyset}\ .$$

The inequational setups and the treatment of *mapException*, as mentioned in the introduction, are not elaborated on in this extended abstract.

References

[JV04] P. Johann and J. Voigtländer. Free theorems in the presence of seq. In *Principles of Programming Languages, Proceedings*, pages 99–110. ACM Press, 2004.

[MLP99] A. Moran, S.B. Lassen, and S.L. Peyton Jones. Imprecise exceptions, Co-inductively. In *Higher Order Operational Techniques in Semantics, Proceedings*, volume 26 of *ENTCS*, pages 122–141. Elsevier, 1999.

[PRH+99] S.L. Peyton Jones, A. Reid, C.A.R. Hoare, S. Marlow, and F. Henderson. A semantics for imprecise exceptions. In *Programming Language Design and Implementation, Proceedings*, pages 25–36. ACM Press, 1999.

[Rey83] J.C. Reynolds. Types, abstraction and parametric polymorphism. In *Information Processing, Proceedings*, pages 513–523. Elsevier, 1983.

[Wad89] P. Wadler. Theorems for free! In *Functional Programming Languages and Computer Architecture, Proceedings*, pages 347–359. ACM Press, 1989.

[1] Of course, this first requires to extend the calculus and proofs by integrating a boolean type and associated term-formers with appropriate typing rules, semantics, and so on. Since the details are entirely straightforward, we omit them.

The expression lemma
– Abstract –⋆

Ralf Lämmel[1] and Ondrej Rypacek[2]

[1] The University of Koblenz-Landau, Germany
[2] The University of Nottingham, UK

Abstract. Algebraic data types and catamorphisms (folds) play a central role in functional programming as they allow programmers to define recursive data structures and operations on them uniformly by structural recursion. Likewise, in object-oriented (OO) programming, recursive hierarchies of object types with virtual methods play a central role for the same reason. There is a semantical correspondence between these two situations which we reveal and formalize categorically. To this end, we assume a coalgebraic model of OO programming with functional objects. The development may be helpful in deriving refactorings that turn sufficiently disciplined functional programs into OO programs of a designated shape and vice versa.

Key words: expression lemma, expression problem, functional object, catamorphism, fold, the composite design pattern, program calculation, distributive law, free monad, cofree comonad.

⋆ The paper appeared in "Proceedings of Mathematics in Program Construction, MPC 2008.".

Stackless Stack Inspection — A Portable Escape Route From Vicious Circles

Baltasar Trancón y Widemann
Baltasar.Trancon@uni-bayreuth.de

Universität Bayreuth

Abstract. Contrary to common belief, recursive functions can be applied effectively to cyclic data structures. But this requires a special calling convention: Cyclic recursion has to be detected by inspecting the call stack at run-time, searching for nested function incarnations with identical inputs. The cycle can then be handled by returning immediately with particular results that depend on the intended semantics, e.g., primitive corecursion or least/greatest predicate fixpoint. While the handling phase is easy to implement portably in a low-level programming language, portable access to arguments of any other than the active call stack frame is not provided by common programming platforms. This article presents a clean solution to that problem for C-like implementation languages, based on a versatile stack management approach called the "stackless transform". It also provides an example for effective use of pointers of up to fifth degree.

1 Cyclic Functional Programming

Cyclic functional programming is the extension of strict purely functional programming to cyclic data. That is, the referential transparent and side-effect free evaluation of functions and predicates on data that can be represented by cells and pointers forming arbitrary finite graph structures.

Semantically, cyclic functional programming retains the call-by-value evaluation of simple functional languages, but differs from its conventional basis by introducing alternative means of definition: Functions may not only be defined by recursion (terminates with acyclic results only), but also by corecursion (may terminate with a cyclic result). Predicates may have more than one fixpoint solution, and at least the least and greatest ones are equally qualified as intended meanings. It is the user's choice to define a function either recursively or corecursively, and to select the desired fixpoint of a predicate definition.

Operationally, these extensions are implemented by special calling conventions. Cyclic reincarnation of a function or predicate can be detected by inspecting the call stack at run-time, looking for two nested frames with identical inputs. When such a pair is found, the cyclic situation can be handled in various ways: For corecursive functions, the outputs of the outer frame are copied to the outputs of the inner frame (dubbed the *ditto* operation). This is a sound thing to do, since for a pure function identical inputs yield identical outputs, and these

```
map(f, l) =
  if l = [] then return []
  else
    h := f(l.head);
    t := map(f, l.tail);
    return cons(h, t)
```

Fig. 1. Return-Value Style Code

may be “shared” cyclically. Of course, this policy requires that outputs must be fixed before a potentially cyclic call is made. For predicates, always returning *true*/*false* for the inner incarnation of a cycle pair yields the least/greatest fixpoint, respectively. Intermediate fixpoints can be realized by varying the cyclic Boolean output value in certain ways. See [Trancón y Widemann, 2008] for the details.

For an example of how cyclic functional programming works, consider the famous *map* function that applies a given function to all elements of a list. It is defined in the Haskell Prelude like this:

$$\begin{aligned} &map\, f\, [\,] &&= [\,] \\ &map\, f\, (x : xs) &&= f\, x : map\, f\, xs \end{aligned}$$

A simple compiler for a strict functional language (unlike the lazy Haskell) might translate this definition to procedural code like the fragment depicted in figure 1.

This code uses procedure return values for result data flow. However, code in “destination passing style” [Larus, 1989] like the fragment depicted in figure 2, that uses output parameters and assignment instead, has definitive advantages [Bigot and Debray, 1999]. It can call a non-initializing variant of the list constructor *before* all nested computations, to the effect that

1. in a traditional, cycle-free situation, it can be executed efficiently as a loop, even though the original functional definition is not tail-recursive – a trick known as “tail recursion modulo constructor” [Warren, 1980].
2. in a cyclic situation, the aforementioned cycle handling mechanism that uses the *ditto* operation can be implemented. This is a correct realization for the

```
map(in f, l; out r) =
  if l = [] then r := []
  else
    r := new cons;
    f(in l.head; out r.head);
    map(in f, l.tail; out r.tail)
```

Fig. 2. Destination-Passing Style Code

interpretation of the above definition as a primitively corecursive function.[1] We might thus extend the definition of *map* in a cyclic functional programming language by the following equation.

map f **cycle** = **ditto**

For the case of the *map* function, cyclic lists are also handled by lazy evaluation in a sufficiently effective way: The consumption of any finite number of elements from the result of *map* evaluates only finitely many recursive invocations, and hence terminates. But the approach of cyclic functional programming also works for recursive predicates, i.e., search problems on cyclic lists, where lazy evaluation fails miserably. This way, operations such as *any*, *all* and *filter* can be generalized easily and effectively to cyclic lists. By combination of corecursive functions and cyclic predicates, even complex operations such as *quicksort* can be extended cleanly to the cyclic case.

Once a cycle has been detected, it is straightforward to handle in virtually any implementation language by providing the appropriate special return value. For instance, the *ditto* operation is implemented in C by the following statement,

```
*inner->r = *outer->r;
```

assuming that `inner` and `outer` point to two nested stack frames that constitute begin and end of a cycle, that the invoked function has a single output, and that the field `r` of the frame structure points to the destination variable for this output. This is the case for the *map* example. A similar statement can be added for each additional output.

Unfortunately, it is more difficult to get access to the stack frames in question in the first place. There is no obvious candidate for an implementation language that is efficiently executable, abstracts from hardware details and yet provides the necessary access to the call stack.

2 Traditional Stack Inspection

Many "managed" languages that are executed on top of a virtual machine runtime environment provide a limited form of access to the call stack through virtual machine primitives and auxiliary libraries. The technique called *stack inspection* focuses on security issues. Running code may be granted or denied the privilege to access certain sensitive resources, based on the trustworthiness of its source. To this end, sets of permissions are assigned to fragments of code at load-time, or assumed by trustworthy code at run-time. Security is maintained by checking active permissions before a resource is accessed.

This mechanism is concerned with relating the level of trust required by a resource to the level of trust granted to calling code, not with discerning patterns

[1] albeit restricted to cyclic lists, whereas the semantic corecursive function is conceptually well-defined for truly infinite lists as well

in the input parameters of nested callers. Hence access to inputs of pending stack frames is not included in either stack inspection primitives or the stack information exposed to user-level code. This makes stack inspection services as they come with off-the-shelf run-time environments unsuitable for the purpose of cycle detection.

The semantics of security-oriented stack inspection are not robust against program transformations that alter the structure of the call stack, most notably tail call elimination [Fournet and Gordon, 2003]. For our approach to stack inspection, we can give simple and effective sufficient conditions for the semantic safety of tail call elimination, even in the presence of cycle detection [Trancón y Widemann, 2008].

3 Stackless Programming

It has been suggested to move frames from the stack to the heap altogether [Appel and Shao, 1996], in order to obtain the amount of control over the life time of frames that is necessary to implement *first-class continuations*. Since the heap-allocated frames can be ordinary data structures, this would also solve the problem of accessing input parameters. But there is a heavy penalty to this solution: While it is possible in principle to use an intermediate language such as C as the implementation language, many features of the language (stack-allocated local variables, procedure calls) and its compiler (separate compilation, inlining, common subexpression elimination) cannot be used. The effort required to produce correct, reusable and efficient code is thus increased dramatically.

For both educational and economical reasons, we envisage a translation scheme for cyclic functional programs that produces code at a level of abstraction similar to the pseudo-code fragments for the *map* function given above. Therefore we do not consider a truly stackless implementation. Instead we focus on a promising hybrid approach called the *stackless transform* [PyPy, 2008]. It has the appealing property that it can be realized in a portable fashion in virtually any language that supports stack-based control flow and heap-allocated data structures. No additional primitives or external library code is needed.

The stackless transform produces procedures that may be called in either of two modes: In *direct* mode, parameters of a call are supplied by an ordinary stack frame. Calls to subprocedures are also made in direct mode, allocating nested stack frames. If all operations (primitive or call) succeed, the procedure returns successfully, deallocating its stack frame. Otherwise, the stack frame is *unwound* by allocating a data structure of appropriate type on the heap and storing local values (parameters and variables) and a reference to the failed operation. Then the procedure returns with a failure, causing the unwinding of enclosing stack frames as well. Once unwound, a frame (now on the heap) may be resumed by calling the procedure in *continuation* mode. In this mode, parameter values are restored from the heap to a newly allocated stack frame, and execution is resumed after the operation that failed previously (which can, by induction, be assumed to have been retried successfully in the meantime). The

deallocation of heap frames is delegated to automatic memory management. A sequential thread of control is formed by maintaining a virtual stack that accumulates unwound frames, and running a trampoline loop that iteratively resumes the innermost frame until the virtual stack is empty. Non-sequential control flow, such as *reusable continuations*, *coroutines*, and *multithreading* can be implemented by variation of this pattern.

4 Stackless Stack Inspection

The stackless transform seems like a promising candidate for implementing cycle detection as portable code. The remainder of this paper describes the pitfalls that prevent a straightforward adaptation of the scheme, and argues that these problems can be overcome with moderate effort.

To demonstrate the main problem, let us adopt the syntax and basic semantics of C/C++. Let us assume that parameter values are passed by reference: If a data type is represented as `struct C`, then an input parameter of this type is represented as `struct C *`, and an output parameter as `struct C **`. Now consider the following function definition:

$$foo\ x \ = \ bar\ (baz\ x)$$

The destination-passing style requires allocating a temporary variable to store the intermediate value of the expression *baz x*, as in the following code (stackless transform not shown):

```
foo(struct C *x, struct C **r) {
  struct D *tmp;
  baz(x, &tmp);
  bar(tmp, r);
}
```

Now assume that *foo* runs in direct mode (so the temporary variable is stack-allocated), and that the nested call to *baz* is unwound. After unwinding, the heap-allocated frame for *baz* will contain a pointer to the stack, namely to the temporary variable of the calling frame. Successively unwinding this frame will destroy the connection and leave a dangling pointer in the output parameter of the *baz* frame!

This example demonstrates is a general restriction of reversible stack unwinding mechanisms: they fail (or rather, require special precautions) for pointers between stack frames. That is not an issue for many popular language paradigms, most notably simple object-oriented languages, where all references are of first degree (pointers to the heap). But it is an issue for any language with either first-class references or parameter passing modes, as in the case for our proposed translation scheme of cyclic functional programming.

Fortunately, the pointers of higher degree that ensue from the output parameter passing mode are of a very regular pattern: They live for a single call, and the callee has three options:

1. use the pointer exactly once to store an output value at the target location,
2. pass the pointer unused to a nested callee,
3. transfer the pointer to the heap by unwinding.

In summary, pointers that represent output parameters obey an *affine logic*: they may be either consumed or stored at most once, but never duplicated. This implies that they can be made *symmetric* by a simple scheme, thus avoiding the dangling-pointer problem by updating pointers to variables being unwound.

This scheme can be realized concisely by C++ smart pointer classes, a class `var` for stack variables and a class `ref` for stack-allocated references (to either the stack or the heap). Heap-allocated references may be realized as ordinary pointers, memory management requirements aside.

```
template <class T> struct var {
  T val, **own;
  var();
  void unwind(T &);
};

template <class T> struct ref {
  T *loc, ***reg;
  ref(T *);
  ref(var<T> &);
  ref(const ref<T> &);
  void unwind(T *&);
  void set(cons T &);
};
```

Remember that `T` is usually instantiated with a pointer of first degree of the form `struct C *`. Note the ordinary value and pointer fields (`var::val` and `ref::loc`, respectively), and the auxiliary pointers for maintaining symmetry (`var::own` and `ref::reg`, respectively). The operations are defined by the following protocol (repetition of `template <class T>` omitted for brevity):

– The auxiliary pointer `var::own` points to the unique heap reference to this stack variable, if any. Initially, there is none.

```
var<T>::var() : own(0) {}
```

– A reference to a stack variable is created by pointing to both fields simultaneously.

```
ref<T>::ref(var<T> * v) : loc(&v.val), reg(&v.own) {}
```

– A reference to the heap does not use its auxiliary pointer.

```
ref<T>::ref(T * l) : loc(l), reg(0) {}
```

– References may be duplicated by the ordinary copy constructor.

```
ref<T>::ref(const ref<T> & r) : loc(r.loc), reg(r.reg) {}
```

The protocol of output parameter passing ensures that the original is no longer used.

– When a reference is unwound to the heap, only the principal pointer is stored. The auxiliary pointer, if nonzero, is used to register the newly created pointer from the heap to the stack. If the auxiliary pointer is zero, a harmless heap-to-heap reference has been created.

```
void ref<T>::unwind(T *& heap) {
  *heap = loc;
  if (reg) *reg = &heap;
}
```

– When a variable is unwound to the heap, only the principal pointer is stored. The auxiliary pointer, if nonzero, is used to update the unique heap reference to this variable to a heap-to-heap reference.

```
void var<T>::unwind(T & heap) {
  *heap = val;
  if (own) *own = &heap;
}
```

– When a reference is used to pass an output value, the auxiliary pointer is not touched.

```
void var<T>::set(const T & val) {
  *loc = val;
}
```

The protocol of output parameter passing ensures that the reference is not reused.

Note that the the type of `ref<struct C *>::reg` is `struct C ****`, a pointer of fourth degree.

5 Difference Parameter Mode

Reconsider the `cons` operator from figure 2. Its correct use depends on the implicit knowledge that two fields `head` and `tail` are created and left uninitialized. Suppose we wish to make this information explicit, thereby allowing the abstraction of `cons` to an auxiliary procedure. This can be achieved by introducing a third parameter mode. Parameters of this mode are meta-output; they yield references to “holes” in the actual output, that is, references to uninitialized data fields that must be filled in by the caller before he may use the output data. In analogy to logic programming, where the technique is well-known as *difference lists*, we call these parameters *difference parameters*. A variant of the *map* function that uses an auxiliary list constructor procedure with difference parameters for both fields is depicted in figure 3.

```
map(in f, l; out r) =
  if l = [] then r := []
  else
    var h, t;
    cons(out r; diff h, t);
    f(in l.head; out h);
    map(in f, l.tail; out t)
```

Fig. 3. Difference-Passing Style Code

It is easy to see that the difference mode relates to the output mode like the output mode to the input mode. A simple pointer realization for the base type `struct C` would hence be `struct C ***`. The smart pointer equivalent, analogous to the previous section, is `var<struct C **>` and `ref<struct C **>`. Note that the type of `ref<struct C **>::reg` is the formidable `struct C *****`, a pointer of fifth degree.

6 Conclusion

We have presented cyclic functional programming as an implementation technique for functional programs that makes use of cycle detection and handling to realize corecursive functions and predicates on finite cyclic data. It requires no lazy evaluation, but destination-passing style outputs and the initialization of outputs prior to (co)recursive calls for effective termination. The cycle handling step is trivial to implement, but the cycle detection step requires detailed access to the call stack, which is hard to obtain in a portable way using a decently high-level implementation language.

We have demonstrated that the idea of stackless programming, particularly the hybrid approach of the stackless transform, is suitable to obtain the required information by unwinding the call stack to a chain of heap data structures. We have pointed out that stack unwinding requires some care in the presence of stack-to-stack references, and presented an economic solution scheme that suffices for the kind of stack-to-stack references that arise from destination-passing style output parameters.

We have also given a concrete example of the application of pointers up to fourth (output parameters) or fifth (difference parameters) degree. Such examples are seldom found in literature, and mostly rather curious than demonstrably useful.

Acknowledgments

Thanks to Christian Heinlein, HTW Aalen, for helpful discussions.

References

[Appel and Shao, 1996] Andrew W. Appel and Zhong Shao. An empirical and analytic study of stack vs. heap cost for languages with closures. *Journal of Functional Programming*, 6(1):47–74, Jan 1996.

[Bigot and Debray, 1999] P. A. Bigot and S. K. Debray. Return value placement and tail call optimization in high level languages. *Journal of Logic Programming*, 38(1):1–29, 1999.

[Fournet and Gordon, 2003] Cédric Fournet and Andrew D. Gordon. Stack inspection: Theory and variants. *ACM Trans. Program. Lang. Syst.*, 25(3):360–399, 2003.

[Larus, 1989] J. R. Larus. *Restructuring Symbolic Programs for Concurrent Execution on Multiprocessors.* PhD thesis, University of California, Berkeley, 1989.

[PyPy, 2008] Pypy translation: The stackless transform, May 2008. `http://codespeak.net/pypy/dist/pypy/doc/translation.html#the-stackless-transform`.

[Trancón y Widemann, 2008] B. Trancón y Widemann. *Strikte Verfahren Zyklischer Berechnung.* PhD thesis, Technische Universität Berlin, 2008.

[Warren, 1980] D. H. D. Warren. DAI Research Report 141, University of Edinburgh, 1980. Not available.