Korrektheit der Übersetzung objektorientierter Wissensrepräsentationssprachen mit statischer Vererbung

Wolfgang Goerigk

Bericht Nr. 9304
Oktober 2001

Institut für Informatik und Praktische Mathematik der
Christian-Albrechts-Universität zu Kiel
Preußerstraße 1-9
W-2300 Kiel 1

Korrektheit der Übersetzung objektorientierter Wissensrepräsentationssprachen mit statischer Vererbung

Wolfgang Goerigk

Bericht Nr. 9304
Oktober 2001

Dieser Bericht enthält die Dissertation des Verfassers

Referent: Prof. Dr. Hans Langmaack
Korreferent: Prof. Dr. Wolfram M. Lippe

Zusammenfassung

In der vorliegenden Arbeit wird ein Ansatz zur Beschreibung der Semantik objektorientierter Programmiersprachen entwickelt, der es erlaubt, die Bedeutung der Vererbung in Programmen getrennt von der Semantik des objektorientierten Kerns zu definieren. Dieses Vorgehen hat gegenüber den rein denotationellen Semantiken, die für Sprachen mit einfacher Vererbung existieren, den Vorteil, daß unterschiedliche mathematische Methoden zur Definition der Vererbung und der Semantik benutzt werden können. Unter Zuhilfenahme einer Vererbungsstrategie, die auf dem Vererbungsgraphen des Programmes beruht, können wir so die Semantik objektorientierter Programme auch mit multipler Vererbung mathematisch beschreiben. Der Ansatz ist offen gegenüber der Modifikation der Vererbungsstrategie und läßt so Experimente mit unterschiedlichen Strategien insbesondere in objektorientierten Wissensrepräsentationssprachen zu.

Wir zeigen ebenfalls, daß sich die Semantik von Methoden, Nachrichten und Objekten mit modifizierbarem lokalem Zustand adäquat denotationell beschreiben läßt, und zwar auf eine Weise, die den begrifflichen Unterschied zwischen Programm und Daten erhält. Objekte sind Daten.

Wir geben einen Übersetzer an, der ein Programm nach Durchführung der Vererbung in ein Programm einer Scheme-ähnlichen Teilsprache übersetzt und beweisen die Korrektheit dieses Übersetzers. Unterstellen wir eine korrekte Implementation der Zielsprache, so stellt die vorliegende Arbeit eine Technik zur mathematischen Definition der Semantik und eine als korrekt nachgewiesene Implementation für eine objektorientierte Sprache mit multipler Vererbung bereit, wobei die konkrete Bedeutung der Vererbung in Form einer Vererbungsstrategie als Parameter in den Ansatz eingeht.

Schlüsselworte

Continuation-Semantik, denotationelle Semantik, Frame, Klasse, Komplettkompilation, Methode, multiple Vererbung, Nachricht, Objekt mit lokalem Zustand, objektorientierte Programmiersprache, objektorientierte Wissensrepräsentationssprachen, Semantik, Übersetzerkorrektheit, Vererbung, Vererbungsgraph, Vererbungsstrategie

Abstract

Different techniques have been proposed in order to describe the semantics of sequential object oriented programming languages. Most of them use denotational semantics to define SMALLTALK-like languages with single inheritance. The situation is not yet clear in the presence of multiple inheritance. The program semantics strongly depends on which components a class actually *inherits* from its superclasses. Hence, the definition of *inheritance* is crucial to the meaning of an object oriented program. At present there is no accepted notion of inheritance for languages with multiple inheritance. Different topological sorting algorithms, for example, have been used in a graph theoretical framework in order to define inheritance in Lisp-like object oriented languages.

The major goal of our work is to develop an approach to the definition of object oriented languages. One important feature is that we allow the use of different and adequate mathematical methods for defining *inheritance* and the semantics of the *object oriented kernel*. We use the notion of an *inheritance strategy* which essentially defines a syntactical transformation expanding programs into associated programs without inheritance. Moreover, we show denotational semantics to be adequate in order to define the kernel, i.e. the semantics of methods, message passing, and objects with local state.

Our approach leads to an implementation technique which depends on the actual definition of the inheritance strategy as well. We define a compiler for programs without inheritance into a Scheme-like subset of our language. We prove this compiler to be correct. Hence, we prove an implementation technique for an object oriented language with multiple inheritance to be correct with respect to a given definition of the inheritance strategy.

The inheritance strategy is a *parameter* of our approach. Both the semantics and the implementation depend on its definition. Therefore, we allow to experiment with different inheritance strategies without loosing mathematical exactness.

Keywords

class, compiler correctness, complete compilation, continuation semantics, denotational semantics, frame, inheritance, multiple inheritance, inheritance graph, inheritance strategy, message passing, method, object oriented knowledge representation language, object oriented programming language, objects with local state, semantics

Inhaltsverzeichnis

1

Einleitung

Die Entwicklung von wissensbasierten Problemlösungen und Expertensystemen ist eine komplexe Aufgabenstellung. Zu ihrer Lösung ist unter anderem die Repräsentation von unterschiedlichen Wissensinhalten erforderlich. Moderne Entwicklungswerkzeuge lösen heute die klassischen KI-Programmiersprachen Common Lisp [Ste84] [Ste90] [Kee89] und PROLOG [CM84] bei der Expertensystementwicklung mehr und mehr ab; man benutzt hybride Wissensrepräsentationssprachen, die für die verschiedenartigen Wissensinhalte jeweils adäquate Repräsentationsformalismen zur Verfügung stellen.

Objektorientierte Sprachen spielen im Zusammenhang mit hybriden Wissensrepräsentationssprachen und den zugehörigen Werkzeugsystemen zur Entwicklung wissensbasierter Expertensysteme heute eine wichtige Rolle, und zwar auf zweierlei Weise.

Klassendefinitionen, Instanzbildung, multiple Vererbung zwischen Klassen und auch die Definition von Methoden und das Senden von Nachrichten, also die Konzepte objektorientierter Sprachen, sind Bestandteil der Wissensrepräsentationssprachen geworden. Sie bilden dort einen der Repräsentationsformalismen, den *objektorientierten* oder *Frameformalismus*. Es hat sich herausgestellt, daß Begriffe, Begriffshierarchien, technische Funktionseinheiten, eben die *Objekte*, mit denen ein Expertensystem umgeht, adäquat objektorientiert repräsentiert werden können. Da eine Expertensystementwicklung vernünftigerweise mit der Definition dieser Begriffswelt beginnt, kommt dem objektorientierten Formalismus neben den ebenfalls unabdingbaren produktionsregelorientierten oder logikorientierten Formalismen eine zentrale Bedeutung zu.

In diesem Zusammenhang stehen Klassen, Vererbung und die Instanziierung im Vordergrund, da sie zur Modellierung der Objekte des Gegenstandsbereiches eines Expertensystems in erster Linie benötigt werden. Methodendefinition und Nachrichtenaustausch treten in den Hintergrund, bieten aber dennoch eine mächtige Schnittstelle zu den anderen Wissensrepräsentationsformalismen und sind adäquat zur Repräsentation algorithmischen Wissens. Objektorientierte Wissensrepräsentationssprachen sind *datentyporientiert* und Objekte sind in erster Linie *Daten*.

Eine ebenfalls gewichtige Rolle spielen objektorientierte Sprachen bei der Implementierung der Werkzeugsysteme selbst. Sie eignen sich gut zur *modularen* Entwicklung großer Softwaresysteme. In diesem Zusammenhang treten die Methodendefinitionen in den Vor-

dergrund. Die Klassen stellen in erster Linie Funktionalität zur Verfügung, die durch den Vererbungsmechanismus in geeigneter Weise kombiniert wird. Bestimmte Klassen beinhalten dann alle zur Lösung einer speziellen Aufgabe innerhalb des Systems nötigen Funktionen und Datenstrukturen. Sie werden instanziiert und die entstehenden Objekte spielen die Rolle von *Modulen* des Programmes. Die Klassendefinitionen beschreiben die *Schnittstelle*, über die die anderen Teile des Programmes mit diesen Instanzen durch das Senden von Nachrichten kommunizieren können.

Klassische Vertreter objektorientierter Sprachen, die auf diese Weise schon vielfach zur Entwicklung großer Systeme eingesetzt wurden, sind Simula [DN66], SMALLTALK [GR83], C++ [Str86], *Eiffel* [Mey85] sowie vor allem Common Lisp [Ste84] [Ste90] mit seinen objektorientierten Erweiterungen Flavors [Sym85], NewFlavors [Sym86] oder neuerdings CLOS [Kee89]. Die objektorientierte Programmierung findet sehr große Beachtung. Sogar PASCAL [JW78] besitzt heute in den weit verbreiteten Implementierungen einen objektorientierten Aufsatz.

Nahezu alle bekannten und erfolgreichen Expertensystemwerkzeuge sind auf der Basis einer der genannten objektorientierten Sprachen entwickelt worden. Kennzeichnend für diese Implementierungen ist, daß der objektorientierte Formalismus der Wissensrepräsentationssprache in der Regel mehr oder weniger direkt auf die zugrundeliegende objektorientierte Implementierungssprache abgebildet wird. Frames werden zu Klassen der Implementierungssprache, die Objekte des Expertensystems werden als Instanzen der Klassen realisiert. Dies hat verschiedene Konsequenzen:

Die meist operationell beschriebene Semantik der Wissensrepräsentationssprache, speziell die des objektorientierten Repräsentationsformalismus, ist geprägt durch die Semantik der zugrundeliegenden objektorientierten Systemprogrammiersprache. Dies gilt insbesondere für den Vererbungsmechanismus, der auf der einen Seite inhaltliche Beziehungen zwischen den Begriffen und Objekten einer Wissensbasis repräsentiert, auf der anderen Seite aber eher technisch motiviert definiert ist, um auf geeignete Weise die Softwarekomponenten von Klassen zusammenzustellen.

Eine zweite nachteilige Konsequenz ist eher von praktischer Relevanz. Die Mächtigkeit der verwendeten Implementierungssprache orientiert sich natürlich nicht in erster Linie an den Erfordernissen der Wissensrepräsentationssprache, sondern an denen der Implementierung des Gesamtsystems. Ist diese Sprache andererseits bereits zur Ausführung der entwickelten Expertensystemanwendungen nötig (s.u.), so lassen diese sich nur schwer z.B. auf kleinere Anwendungsrechner portieren, da Kleinrechner für hochkomplexe objektorientierte Sprachen wie beispielsweise Common Lisp und CLOS nicht leistungsfähig genug sind. Lisp-Implementierungen auf Kleinrechnern enthalten meist keine objektorientierten Erweiterungen, und der Wechsel zu einer anderen objektorientierten Sprache verbietet sich aufgrund semantischer Inkompatibilitäten, insbesondere im Zusammenhang mit der Semantik von Vererbung.

Der Kern eines Expertensystems, das mit einem Werkzeugsystem wie z.B. BABYLON[1] [dBC85] [CdV89] erstellt wurde, besteht aus der *Wissensbasis* und der *Inferenzkomponente*, die als Interpreter vom Werkzeugsystem zur Verfügung gestellt wird und die Ausführung der

[1] BABYLON ist die hybride Wissensrepräsentationssprache des von der Forschungsgruppe *Expertensysteme* der Gesellschaft für Mathematik und Datenverarbeitung (GMD) entwickelten gleichnamigen Expertensystemwerkzeugs.

Wissensbasis erlaubt. Die Wissensbasis ist also ein *Programm* der hybriden Wissensrepräsentationssprache.

Gelingt es nun, die durch den Interpreter gegebene Semantik dieser Sprache unabhängig von der Bedeutung der zugrundeliegenden objektorientierten Implementierungssprache zu definieren, so lassen sich Expertensysteme auch durch *Kompilation der Wissensbasis* implementieren.

Dieser neue Ansatz zur Implementierung, Portierung und gleichzeitigen Effizienzsteigerung von Expertensystemen wird in [AG87a], [AG87b], [AGS88], [AGS91], [Goe89a] und [Goe89b] vorgestellt. Konkret ist im Rahmen eines Forschungsprojektes zur „effizienten Implementierung wissensbasierter Expertensysteme“[2] ein Übersetzer für BABYLON[3] entstanden, der eine Wissensbasis, nachdem sie in dem Werkzeugsystem BABYLON fertig entwickelt und validiert worden ist, zunächst in ein semantisch äquivalentes Programm in reinem Common Lisp übersetzt. In einem weiteren Schritt wird sie mit einem ebenfalls im Rahmen dieses Projektes entstandenen Übersetzer von Common Lisp nach C [Bur91] [Knu91] [BGK91] in ein C-Programm übersetzt und läßt sich so als eigenständige Anwendung auf eine breite Palette von Anwendungsrechnern portieren. Der durch die Kompilation erreichte Effizienzgewinn um den Faktor 10 bis 20 gegenüber der interpretierten Ausführung läßt die Nutzung der Expertensysteme auch auf kleinen Anwendungsrechnern ohne merklichen Effizienzverlust zu.

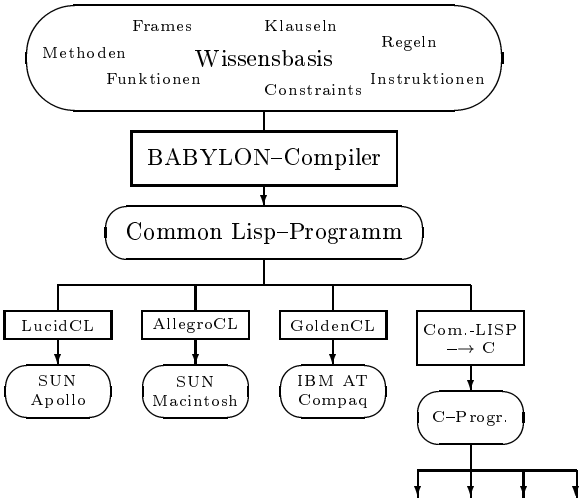


Abbildung 1.1: Wissensbasiskompilation

Obwohl, streng genommen, die einzelnen Sprachformalismen für sich mächtig genug sind,

[2]Dieses Projekt wurde auf Initiative und mit Unterstützung der Dr. Ing. Rudolf HELL GmbH in Kiel und in Kooperation mit der Forschungsgruppe „Expertensysteme“ der GMD durchgeführt.

[3]BABYLON enthält neben dem objektorientierten Formalismus einen produktionsregelorientierten Formalismus zur Beschreibung der häufig heuristischen Vorgehensweise eines Experten bei der Lösung von Problemen, einen logikorientierten zur Definition komplexerer Prädikate oder zur Beschreibung von Relationen zwischen Objekten sowie einen Constraint-Formalismus auf der Basis des in [Gü88] beschriebenen Systems CONSAT. Zudem ist der Durchgriff auf die Implementierungssprache Lisp möglich, z.B. mit der Methodensprache des objektorientierten Formalismus, so daß auch algorithmisches Wissen repräsentiert werden kann.

um die jeweils anderen zu ersetzen, haben doch alle ihre Berechtigung. Zum einen erleichtern verschiedene Formalismen den Prozeß der Entwicklung eines Expertensystems, da sie zur Repräsentation unterschiedlicher Wissensinhalte verschieden gut geeignet (*adäquat*) sind. Begriffe, Begriffshierarchien und Objekte lassen sich in dem objektorientierten Formalismus adäquater beschreiben als beispielsweise in dem logikorientierten, obwohl, wie [Sch87] zeigt, objektorientierte Sprachbestandteile in PROLOG simuliert werden können. Der Zwang, zur Repräsentation bestimmter Wissensinhalte einen nicht adäquaten Formalismus benutzen zu müssen, führt zu einem unnötigen Kodierungsschritt, der den schnellen, inkrementellen Programmentwicklungsprozeß hemmt, der für die Expertensystementwicklung sehr hilfreich ist. Überdies haben Expertensysteme, genauer ihre Wissensbasen, nicht nur die Aufgabe, als *Programm* zu funktionieren; sie spielen zusätzlich die nicht weniger wichtige Rolle von Dokumenten, um die dem System zugrundeliegenden Wissensinhalte in auch für Menschen verständlicher Form zu dokumentieren und sie so in zukünftigen Projekten weiterverwenden zu können.

Speziell für den objektorientierten Formalismus stellt sich das Problem der Adäquatheit für die Wissensrepräsentation ebenfalls im Zusamenhang mit der Semantik der Vererbung. Wir haben bereits erwähnt, daß die Vererbung in Wissensrepräsentationssprachen inhaltliche Beziehungen zwischen den Begriffen und Objekten einer Wisssensbasis ausdrückt, während sie in den tatsächlich vorhandenen Sprachen, wie auch in BABYLON, durch die sehr technisch orientierten Vererbungsmechanismen der zugrundeliegenden Implementationssprachen implementiert wird. Die Diskussion um adäquate *Vererbungsstrategien* für objektorientierte Wissensrepräsentationssprachen ist noch nicht abgeschlossen.

Das Problem, das sich aus der geschilderten Situation ergibt, hat uns motiviert, der Frage nachzugehen, ob nicht eine mathematisch exakte Definition der Semantik objektorientierter Sprachen gelingt, die unabhängig von konkreten Implementierungstechniken und zudem weitgehend offen für die Definition adäquater Vererbungsmechanismen ist. Diese Frage beantworten wir in der vorliegenden Arbeit positiv. Wir schlagen eine Methode zur Beschreibung der Semantik objektorientierter Sprachen vor, die es gestattet, den Vererbungsmechanismus *getrennt* von der dynamischen Semantik des objektorientierten *Kerns* der Sprache zu definieren. Zum einen erlaubt dieses Vorgehen, unterschiedliche mathematische Methoden zur Definition der Vererbung und zur Definition der dynamischen Semantik zu benutzen. Dies führt in der Regel zu verständlicheren Sprachdefinitionen.

Zum anderen läßt sich die dynamische Semantik unabhängig von der konkret benutzten *Vererbungsstrategie* definieren. Ein Experimentieren mit unterschiedlichen Vererbungsstrategien wird so leichter möglich.

Bereits ohne das Vorhandensein von Vererbung unterscheiden sich objektorientierte Sprachen *in ihrem Kern* von den klassischen imperativen Sprachen. [CW86] charakterisiert sie als objektorientierte Erweiterungen imperativer Sprachen, die *Objekte* als *Datenabstraktionen* mit lokalem *Zustand* und einer Schnittstelle zu *benannten Operationen* unterstützen. Objekte

besitzen einen *assoziierten Objekttyp*, nämlich die Klasse oder den *Frame*, als dessen Instanz sie gebildet wurden.

Objekte sind dynamisch getypte *Record*-artige Daten, deren *Instanzvariablen* oder *Slots* (*Schlitze*) ihren lokalen Zustand bilden, der durch schreibende Zugriffe modifizierbar ist.

Die benannten Operationen nennen wir *Methoden*. Nur die Methoden des Objektes haben Zugriff auf den Zustand des Objektes. Diese Eigenschaft nennt man *Verkapselung*. Die meisten objektorientierten Sprachen sind selbst *hybrid*. Sie enthalten eine imperative oder, in unserem Fall, applikative Teilsprache, die zusätzlich zu den rein objektorientierten Sprachbestandteilen zur Formulierung der Rümpfe der Methoden benutzt werden kann. Eine Ausnahme bildet SMALLTALK [GR83] als reine objektorientierte Sprache.

Methoden werden durch das *Senden von Nachrichten* aufgerufen. Ist e_0 ein Ausdruck, der ein Objekt bezeichnet, so bedeutet

$$e_0 \,.\, m\,(\,e_1\,)$$

das Senden einer Nachricht mit dem Namen m und dem Argument e_1 an das Objekt e_0 . Zu einem Nachrichtennamen m sind in einem objektorientierten Programm in der Regel verschiedene Methoden definiert. „Instanzen unterschiedlicher Frames können auf eine Nachricht durch verschiedene Methoden reagieren." Eine solche Nachricht bedeutet also den Aufruf einer der im Programm zu m definierten Methoden.

Da Methoden nur durch Nachrichten an Objekte aufgerufen werden, ist es sinnvoll, eine Bezeichnung für das Empfängerobjekt im Rumpf der Methode zur Verfügung zu haben. Wir nennen diesen zusätzlichen *impliziten* Parameter von Methoden *self*.

Die Bedeutung einer Nachricht hängt also auf zwei verschiedene Weisen von dem Objekt ab, das die Nachricht empfängt:

- Die Auswahl der richtigen Methode hängt von dem *Objekttyp*, also dem Frame des Empfängerobjektes ab.
- Die Bedeutung des Methodenrumpfes selbst hängt von einer Bindung von *self* an das Empfängerobjekt ab.

Inzwischen sind, beginnend mit der Arbeit von M. Wolczko [Wol87], eine Reihe von Arbeiten zur denotationellen Semantik objektorientierter Sprachen entstanden, vornehmlich im Umfeld von SMALLTALK und damit für Programme mit einfacher Vererbung. U.S. Reddy beschreibt in [Red88] Objekte als *Closures*. Dort sind die Methoden funktionale Komponenten der Objekte. Die in ihnen freien Vorkommen von *self* werden bei der Instanziierung durch eine Fixpunktbildung an das erzeugte Objekt selbst gebunden. Das impliziert, daß bei jeder Instanziierung die Methodenfunktionen des entstehenden Objektes neu gebildet werden. Verschiedene Objekte müssen verschiedene Kollektionen von Methodenfunktionen beinhalten, auch wenn sie der gleichen Klasse angehören. Die Instanziierung erzeugt funktionale Datenobjekte der allgemeinsten Form (sog. *upward closures*), denn die Objekte sind i.a. global in dem Programm verfügbare Datenstrukturen. Die Nachricht $e_0 \,.\, m\,(\,e_1\,)$ bedeutet damit tatsächlich den Aufruf einer durch einen *Record*-Zugriff erhaltenen Funktion. Noch deutlicher wird diese Vorstellung in [Car84], wo die Semantik einer streng getypten Sprache mit impliziter multipler Vererbung beschrieben wird. S. Kamin definiert in [Kam88] eine denotationelle

Semantik für eine Teilmenge von SMALLTALK, in der die Methoden wie in unserem Ansatz den Klassen zugeordnet sind. [Red88] und [Kam88] beschäftigen sich mit dem in SMALLTALK vorliegenden Fall *einfacher Vererbung*. In diesem Fall läßt sich auch der Vererbungsmechanismus ohne die Definition einer komplexen Vererbungsstrategie durch eine Fixpunktbildung denotationell beschreiben.

Im Unterschied zu [Red88] oder [Car84] wollen wir im Rahmen dieser Arbeit möglichst weitgehend begrifflich zwischen *Programm* und *Daten* unterscheiden. Wir fassen die Objekte als *Daten* auf, während wir die Methoden eindeutig dem *Programm* zuordnen. Sie sind Bestandteil der Framedefinitionen. Zwar tritt eine Methodendefinition

$$defmethod\ A\ m\ x\ =\ e$$

als eigenständige syntaktische Einheit auf, sie ist aber durch die Nennung des Framenamen A in ihrer Definition dem Frame A zugeordnet. Eine Nachricht entspricht nun nicht dem Aufruf einer funktionalen Komponente des Empfängerobjektes e_0 , sondern dem Aufruf eines *Funktionals*

$$\lambda\, self\, .\, \lambda\, x\, .\, e\, .$$

Bezeichnen wir der Einfachheit halber dieses Funktional wieder mit m , so bedeutet die Nachricht $e_0\, .m\, (\, e_1\,)$ in unserem Ansatz einen Aufruf der Form

$$m\, (\, e_0\,)\, (\, e_1\,)\, .$$

Natürlich muß dafür gesorgt sein, daß die durch den Typ des Objektes e_0 bestimmte richtige Methode auf diese Weise aufgerufen wird.

Die *generische* (von der *Art* des Empfängerobjektes bestimmte) Auswahl der richtigen Methode sowie das Vorhandensein dynamisch getypter *Record*-artiger Daten mit lokalem Zustand sind die Eigenschaften des *Kerns objektorientierter Sprachen*, die diese von den klassischen imperativen Sprachen unterscheiden. Die Semantik dieses *Kerns* definieren wir im Kapitel 4 dieser Arbeit *denotationell* [Sto77] [Sco81] [Sco76].

Wir betrachten objektorientierte Programme mit *multipler Vererbung*. Ein Frame kann im allgemeinen mehrere Superframes haben, aus denen er Instanzvariablen oder Methoden potentiell erbt. Nicht alle Methoden oder Instanzvariablen der Superframes werden tatsächlich vererbt; enthält der Frame lokal eine *speziellere* Methode oder Instanzvariable gleichen Namens, so *verschattet* diese Definition die *allgemeinere* aus dem Superframe. Ein Frame erhält eine Komponente nur einmal. Dies ist die übliche Vorstellung von Vererbung, wie sie aus Sprachen wie SMALLTALK oder Lisp mit seinen objektorientierten Erweiterungen bekannt ist.

Im Fall *einfacher Vererbung*, z.B. in SMALLTALK, ist allein durch die genannte *Verschattungsregel* klar, aus welchen Komponenten sich eine Klasse zusammensetzt. Eine Klasse *besitzt* eine Komponente entweder lokal oder erbt sie aus ihrer Superklasse, falls eine solche vorhanden ist. Diese *operationelle* Vorstellung des *Zusammensuchens* der Komponenten entlang der

durch die *Vererbungsrelation* gegebenen linearen Liste der Superklassen wird in der Literatur auch als *method lookup semantics* bezeichnet [CP89]. [CP89] und [Red88] haben unabhängig voneinander eine denotationelle Definition dieses Vererbungsmechanismus angegeben, die das beschriebene iterative Vorgehen rekursiv, also durch Fixpunktbildung modelliert. Auch in [Kam88] wird die Vererbung durch Fixpunktbildung definiert.

Im Fall multipler Vererbung tritt eine zusätzliche Komplikation auf, die dadurch entsteht, daß ein Frame eventuell eine Instanzvariable oder Methode von mehr als einem seiner Superframes erben soll (vgl. Kapitel 3). Um auch in diesem Fall die Konfliktlösung auf die Anwendung der oben beschriebenen Verschattungsregel zurückzuführen, betrachtet man *topologische Sortierungen* des *Vererbungsgraphen* $\mathbf{VG}_p(A)$ eines Frames A in einem Programm p , die künstlich eine lineare Anordnung der Menge der Superframes erzeugen. Die Vererbungsstrategien, die z.B. in den objektorientierten Erweiterungen Flavors [Sym85] oder NewFlavors [Sym86] von Lisp definiert sind, basieren auf einer solchen *Linearisierung* des Vererbungsgraphen. In CLOS [Kee89] wird die lineare Anordnung der Superframes als *class precedence list* bezeichnet.

Es existieren durchaus verschiedene plausible Strategien, die Linearisierung des Vererbungsgraphen eines Frames zu bestimmen, die auch zu unterschiedlichen Resultaten führen. Die Vererbungsstrategien aller drei genannten Lisp-Erweiterungen beispielsweise unterscheiden sich bezüglich des Linearisierungsverfahrens voneinander. Die benutzten topologischen Sortierverfahren stellen nichttriviale Algorithmen auf Relationen und Graphen dar. Es ist nicht klar, wie diese auf ähnlich elegante Weise wie in den oben erwähnten Arbeiten denotationell definiert werden können. Wir vermuten, daß sich die denotationelle Beschreibung objektorientierter Sprachen dann auf eine bis zur Unverständlichkeit führende Weise verkomplizieren würde.

Aus Sicht objektorientierter Wissensrepräsentationssprachen ist zudem ein weiteres Experimentieren mit verschiedenen *Vererbungsstrategien* wünschenswert, denn auch die Adäquatheit der genannten, auf Linearisierungen des Vererbungsgraphen beruhenden Strategien ist nicht erwiesen.

Auch dieses Problem wird durch die *Trennung* der Definition der Vererbung von der der dynamischen Semantik des Kerns der Sprache gelöst. Wir betrachten in der vorliegenden Arbeit *komplette* Programme, also nicht inkrementelle oder unvollständige Programmfragmente. Deshalb kennen wir alle für die Bedeutung der Vererbung relevanten Programmteile. Alle Frame- und Methodendefinitionen des Programms sind *statisch* bekannt und ändern sich nicht. Der Vererbungsmechanismus läßt sich in dieser Situation als eine globale Operation auf dem gesamten Programm auffassen, die dem Programm die durch die Vererbung zusätzlich entstehenden Methodendefinitionen hinzufügt und die Instanzvariablenlisten der Framedefinitionen durch das Hinzufügen der ererbten Instanzvariabalen vervollständigt. Damit läßt sich der Begriff einer *Vererbungsstrategie* sehr allgemein definieren: Wir benötigen die Beschreibung zweier Verfahren, die, gegeben der komplette Deklarationsteil des Programmes, zu diesem effektiv *alle* Methodendefinitionen und die Liste *aller* Instanzvariablen aller Frames des Programmes bestimmen.

Mit einer konkret gegebenen Vererbungsstrategie, die aus einem Paar effektiv berechenbarer Abbildungen besteht, kann das Programm dann durch eine ebenfalls globale *syntaktische Transformation* in ein Programm ohne Vererbung überführt werden. Dabei lassen wir nicht jede beliebige Vererbungsstrategie zu, sondern verlangen von den erlaubten (*vernünftigen*) Vererbungsstrategien einige plausible Eigenschaften, beispielsweise, daß sie die lokalen Komponenten eines Frames erhalten.

Im Kapitel 2 definieren wir zunächst die Syntax der in dieser Arbeit betrachteten objektorientierten Sprache und erläutern die Bedeutung der einzelnen syntaktischen Bestandteile informell. Programme dieser Sprache bestehen aus einem Deklarationsteil und einem Hauptprogramm. Neben Frame- und Methodendefinitionen enthält der Deklarationsteil ebenfalls Definitionen gewöhnlicher Funktionen. Wir definieren eine *hybride* objektorientierte Sprache als Erweiterung einer applikativen Teilsprache, die Ähnlichkeiten mit Scheme [Abe91] aufweist. Die Sprache insgesamt ist nicht mehr funktional. Durch die Instanzvariablen der Objekte enthält sie das Konzept von *Variablen* und *Zuweisungen*.

Ebenfalls im Kapitel 2 überführen wir die Programme in eine abstraktere mathematische Notation, die es erlaubt, Programmbestandteile durch Anwendung von Abbildungen zu selektieren und Programmeigenschaften „formaler“ zu beschreiben. Der wesentliche Schritt, der bei dieser *syntaktischen* Operation durchgeführt wird, ist die Vereinheitlichung von Funktions- und Methodendefinitionen. Sowohl die Funktionsdefinitionen als auch die Methodendefinitionen des Programmes werden in Bindungen von *Abstraktionen* an ihre *Namen* überführt. Ein Methodenname setzt sich aus dem Framenamen und dem Nachrichtennamen zusammen. An diesen Namen binden wir das Funktional

$$\lambda\, self\,.\,\lambda\, x\,.\,e$$

und machen so den impliziten Parameter *self* des Methodenrumpfes zu einem expliziten Parameter der Methode.

Um das Vorgehen bei der Definition der Semantik und auch bei der Übersetzung zu verdeutlichen, haben wir in Abbildung 1.2 die syntaktischen Transformationen, die entstehenden Programme und ihre Semantik im Überblick dargestellt. Abbildung 1.3 stellt die entsprechenden syntaktischen und semantischen Bereiche dar. Der Bereich $\mathit{Value} \times \mathit{Store}$ ist der semantische Bereich der *Programmergebnisse*.

Aus dem Deklarationsteil d eines Programmes $p = d\,;\, e$ entsteht zunächst ein Tripel

$$\begin{aligned} \gamma \;\in\; \mathit{InhGraph} \;=_{\mathrm{Def}}\; & (\,\mathit{Ident} \xrightarrow{\mathit{fin}} \mathit{Abstraction}\,) \;\times \\ & (\,\mathit{Framename} \xrightarrow{\mathit{fin}} \mathit{Framename}^*\,) \;\times \\ & (\,\mathit{Framename} \xrightarrow{\mathit{fin}} \mathit{InstVar}^*\,) \end{aligned}$$

endlicher Abbildungen, dessen erste Komponente die Funktions- und Methodendefinitionen des Programmes enthält. Die zweite und dritte Komponente enthalten die lokalen Listen der

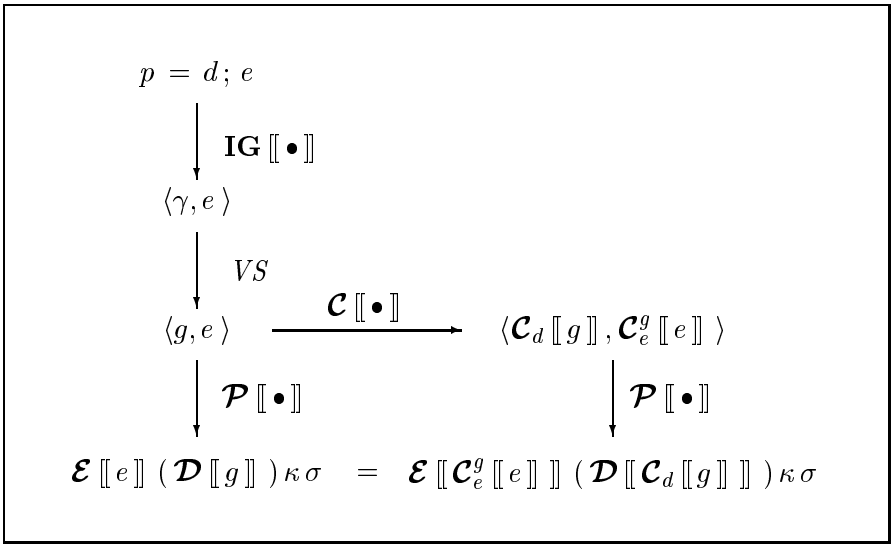


Abbildung 1.2: Semantik, Übersetzung und Korrektheit

Superframes $(A_1, \ldots, A_n)$ und die lokalen Instanzvariablenlisten $(y_1, \ldots, y_k)$ aus den in d vorkommenden Framedefinitionen:

$$\mathit{defframe}\ A\ (A_1, \ldots, A_n)\ (y_1, \ldots, y_k)$$

Kapitel 3 widmet sich der Definition der *statischen Semantik*. Dort definieren wir die Durchführung der Vererbung als Anwendung der Vererbungsstrategie VS auf den nunmehr abstrakten Deklarationsteil γ . Durch diese Transformation entsteht ein Programm, genauer der Deklarationsteil eines Programmes *ohne Vererbung*. Alle Framedefinitionen sind um die fehlenden Definitionen der *ererbten* Methoden ergänzt. Die Instanzvariablenlisten enthalten nunmehr vollständig alle Instanzvariablen. Die entstehenden Deklarationsteile bezeichnen wir auch als *Übersetzungszeitumgebungen*

$$\begin{aligned} g \ \in\ \mathit{CEnv} \ =_{\mathrm{Def}}\ & (\mathit{Ident} \xrightarrow{\mathit{fin}} \mathit{Abstraction}) \ \times \\ & (\mathit{Framename} \xrightarrow{\mathit{fin}} \mathit{InstVar}^*) \end{aligned}$$

Die Durchführung der Vererbung ist eine *semantikdefinierende* syntaktische Transformation. Bereits die Definition der *Wohlgeformtheit* von Programmen erfordert die Kenntnis aller Instanzvariablen und aller Methoden eines Frames. Als wesentliche Einschränkung fordern wir von *wohlgeformten* Programmen, daß nur *instanziierbare* Frames instanziiert werden. Dies insbesondere ist eine Eigenschaft, die erst nach Durchführung der Vererbung entschieden werden kann.

Programme ohne Vererbung enthalten nun nur noch Sprachbestandteile, die wir dem *Kern* der objektorientierten Sprache zugerechnet haben. In Kapitel 4 definieren wir für die Ausdrücke dieses Kerns eine denotationelle Continuation-Semantik $\mathcal{E}\,[\![\bullet]\!]$.

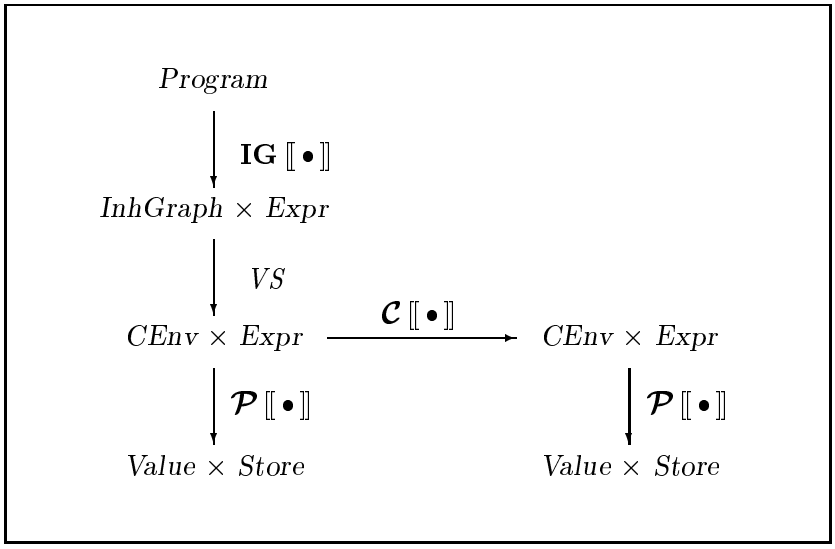


Abbildung 1.3: Syntaktische und semantische Bereiche der Programme

Ebenfalls in diesem Kapitel definieren wir einen Übersetzer $\mathcal{C}_e^g[\![\bullet]\!]$ für Ausdrücke, der die in ihnen enthaltenen Nachrichten in eine explizite endliche generische Methodenfunktionsauswahl übersetzt. In Kapitel 5 zeigen wir zunächst die Korrektheit der Übersetzung von Ausdrücken. Schließlich weiten wir die Definition der denotationellen Semantik und auch die des Übersetzers zunächst auf den Deklarationsteil $g \in \mathit{CEnv}$ und dann auf das ganze Programm $\langle g, e \rangle \in \mathit{CEnv} \times \mathit{Expr}$ aus und beweisen in Theorem (5.22) die Korrektheit der Übersetzung von Programmen.

Berücksichtigen wir einmal die Framedeklarationen nicht, dann entspricht der abstrakte Deklarationsteil eines Programmes ohne Vererbung einem im allgemeinen wechselseitig rekursiven Gleichungssystem, das aus Definitionen von Methoden und Funktionen besteht. Folgerichtig ist seine Semantik als der kleinste Fixpunkt eines Funktionals definiert, das Umgebungen auf Umgebungen abbildet. Die Semantik des gesamten Programmes ist dann die Semantik des Hauptprogrammausdruckes in der Umgebung, die als Semantik des Deklarationsteils des Programmes entsteht.

Die Ausweitung der Übersetzungsfunktion geschieht dadurch, daß wir den Übersetzer für Ausdrücke sowohl auf die im Deklaratonsteil vorhandenen Rümpfe der Methoden und Funktionen als auch auf den Hauptprogrammausdruck anwenden.

Aus dem ursprünglichen Programm $p = d; e$ ist nunmehr ein semantisch äquivalentes Programm entstanden, dessen Deklarationsteil nur noch aus wechselseitig rekursiven Funktionsdefinitionen besteht. Die in ihm vorkommenden Ausdrücke enthalten nur noch dynamische Instanziierungen *Record*-artiger Datentypen sowie lesende und schreibende Zugriffe auf die Komponentenvariablen und die endliche generische Methodenfunktionsauswahl. Natürlich kommen die Ausdrücke der applikativen Teilsprache ebenfalls vor.

Unterstellen wir eine korrekte Implementation der Scheme-artigen Zielsprache des Übersetzers, so stellt die vorliegende Arbeit ebenfalls eine als korrekt nachgewiesene Implementation für eine objektorientierte Sprache mit multipler Vererbung bereit, wobei die konkret benutzte Vererbungsstrategie als *Parameter* des Ansatzes eingeht. Sowohl der Übersetzer als auch die denotationelle Semantik hängen letztlich von der Vererbungsstrategie ab. Die denotationelle Semantik des Kerns unserer Sprache, die Definition des Übersetzers und auch die Überlegungen zur Korrektheit der Implementation bleiben von einer eventuellen Änderung der Vererbungsstrategie unberührt.

Ich danke Herrn Prof. Dr. Hans Langmaack für die Möglichkeit, auf dem interessanten Forschungsgebiet der Semantik und Übersetzung hybrider Programmiersprachen zu arbeiten. Herrn Dr. Dieter Ackermann danke ich für zahlreiche Diskussionen und Anregungen ebenso wie Herrn Dr. Friedemann Simon, der Vorabversionen dieser Arbeit behutsam gelesen und durch viele Anregungen und Vorschläge geholfen hat, sie zu verbessern.

Ein besonderer Dank gilt meiner Frau Heidemarie und meinem Sohn Stephan. Sie haben die Entstehung dieser Arbeit, wo immer es ihnen möglich gewesen ist, tatkräftig unterstützt.

2

Die Sprache

Objektorientierte Sprachen können als Erweiterungen prozeduraler oder imperativer Sprachen angesehen werden. Sie unterstützen Objekte als Datenabstraktionen mit lokalem Zustand und einer Schnittstelle zu benannten Operationen. Objekte besitzen einen assoziierten Objekttyp, als dessen Instanzen sie gebildet werden. Diese können Attribute von anderen Objekttypen *erben* [CW86]. Die Operationen werden meist *Methoden* oder *Behaviors* genannt, Objekttypen werden in der Regel als *Klassen* bezeichnet. In objektorientierten Wissensrepräsentationssprachen hat sich der Begriff *Frame* [Min80] durchgesetzt, den wir in dieser Arbeit benutzen werden. In den ersten objektorientierten Erweiterungen von Lisp heißen sie *Flavors* (Geschmack, Aroma), wahrscheinlich aufgrund der Vorliebe der Autoren von [Sym85] für Eis, insbesondere mit Vanille-*Geschmack*[1].

Nur die Methoden eines Objektes (s.u.) haben Zugriff auf den Zustand des Objektes, nur in ihnen können die Komponenten, die *Instanzvariablen* oder *Slots* (Schlitze) genannt werden, gelesen oder beschrieben werden. Diese Eigenschaft nennt man *Verkapselung*. *Vererbung* und *Verkapselung* sind entscheidende Merkmale objektorientierter Sprachen, die diese von anderen unterscheiden. Das Nachrichtensenden (engl. *message passing*) entspricht hingegen in den meisten objektorientierten Sprachen Funktionsaufrufen *generischer* Funktionen, die in Abhängigkeit von dem Empfänger eine der vorhandenen Methoden aufrufen. Der englische Begriff *message passing* wird in der Literatur häufig auch zur Bezeichnung eines Konzeptes parallelverarbeitender Rechnerarchitekturen verwendet. Deshalb weisen wir darauf hin, daß wir im Rahmen dieser Arbeit *sequentielle* objektorientierte Sprachen betrachten.

In diesem Kapitel definieren wir zunächst die Syntax einer hybriden objektorientierten Sprache (Abschnitt 2.1). Dabei diskutieren wir informell die Bedeutung insbesondere der objektorientierten Teile. Die Syntax definieren wir als abstrakte Syntax (vgl. [Sto77]), indem wir *syntaktische Bereiche* (Abschnitt 2.1.1) einführen und dann die Struktur der abstrakten Syntaxbäume der einzelnen syntaktischen Elemente durch *syntaktische Gleichungen* (Abschnitt 2.1.2) definieren. Abschnitt 2.2 zeigt zwei ausführliche Beispiele.

Um ein objektorientiertes Programm zu verstehen, benötigt man den Überblick über die Komponenten der einzelnen Frames, d.h. man muß die ihnen zugeordneten *Methoden* und

[1] Vanilla ist dort der Name der Wurzel des Vererbungsgraphen.

die *Instanzvariablen* kennen, die bei der Bildung von Instanzen den entstehenden Objekten zugeordnet werden. Die Komponenten werden mit der *Vererbungsstrategie* (vgl. Kapitel 3) aus denen der Frames und aller ihrer Superframes zusammengestellt. In Abschnitt 2.3 definieren wir *abstrakte Programme*, deren Deklarationsteil eine Repräsentation des *Vererbungsgraphen* eines Programmes darstellt.

Die Begriffe objektorientierte Programmierung und Vererbung tauchten zuerst in Simula67 [DN66] auf. Eine rein objektorientierte Sprache ist SMALLTALK [GR83], in der das Senden von Nachrichten und die damit verbundene, implizit durchgeführte Methodenauswahl die einzigen Elemente der Kontrollflußprogrammierung sind. Eine Nachricht hat die Bedeutung eines Funktionsaufrufes mit streng sequentiellem Kontrollfluß: Die Argumente werden in der einen, das Ergebnis in der anderen Richtung kommuniziert. Dies gilt ebenfalls für die bekannten objektorientierten Erweiterungen von Lisp, Flavors [Sym85] und NewFlavors [Sym86]. Sie enthalten multiple Vererbung und unterscheiden sich durch unterschiedliche Vererbungsstrategien (vgl. Kapitel 3). In CLOS [Kee89] sind Nachrichten sogar syntaktisch durch Aufrufe sog. *generischer Funktionen* ersetzt.

Unsere Sprache ist hybrid in dem Sinne, daß sie einen applikativen Kern besitzt, der um die Konzepte der objektorientierten Programmierung erweitert ist. Dies sind Deklarationen von *Frames* mit multipler Vererbung, die Definition von *Methoden* zu Nachrichtennamen sowie das Senden von Nachrichten an Objekte, die Bildung von Objekten als Instanzen von Frames sowie der lesende und schreibende Zugriff auf Instanzvariablen von Objekten.

Framedeklarationen und Methodendefinitionen bilden zusammen mit den Funktionsdefinitionen des applikativen Kerns den Deklarationsteil eines Programmes. Nachrichten und Instanziierungen sind die *Ausdrücke* der objektorientierten Erweiterung, die neben denen des applikativen Kerns sowohl in Funktions- und Methodendefinitionen wie auch im *Hauptprogrammausdruck* eines Programmes auftreten können. Lesende und schreibende Zugriffe auf Instanzvariablen dürfen nur in den Rümpfen von Methodendefinitionen auftauchen. Diese Eigenschaft werden wir, um die Syntax einfach zu halten, in der Definition der statischen Semantik (vgl. Kapitel 3) von wohlgeformten Programmen verlangen.

Der applikative Kern der Sprache enthält λ-Abstraktionen als Ausdrücke, bildet also eine funktionale Sprache mit höheren Funktionalen, da Abstraktionen sowohl als Argumente von Aufrufen als auch als Resultate von Ausdrücken vorkommen können. Die Sprache insgesamt ist aber nicht funktional, denn mit den Instanzvariablen von Objekten enthält sie das Konzept von Variablen, die durch Zuweisungen ihren Wert ändern können. Wir schließen nicht aus, daß auch Funktionen Werte von Instanzvariablen sein können, werden aber im folgenden motivieren, daß solche *funktionalen Komponenten* von Objekten nicht mit Methoden verwechselt werden dürfen.

Unser Ansatz hängt nicht essentiell davon ab, daß Funktionen als Daten in Programmen auftreten können. Stattdessen erlaubt diese Tatsache die Beschränkung auf null- oder einparametrige Funktionen und Methoden, da wir uneingeschränkt *currifizieren* können. Dies wiederum vereinfacht die Notation in der Definition der Semantik (vgl. Kapitel 4) und in den Beweisen im Kapitel 5.

Wie SMALLTALK und auch die objektorientierten Erweiterungen von Lisp ist unsere Sprache ungetypt in dem Sinne, daß Variablen und auch Instanzvariablen Daten unterschiedlichen Typs enthalten dürfen, und daß, wie in Lisp, die Zweige des Konditionals Ergebnisse verschiedener Typen liefern können. Damit ist einem Ausdruck sein Typ statisch nicht anzusehen.

2.1 Syntax

Im Sinne von [Sto77] definieren wir in diesem Abschnitt zunächst die abstrakte Syntax, indem wir syntaktische Bereiche angeben (Abschnitt 2.1.1) und durch syntaktische Gleichungen (Abschnitt 2.1.2) induktiv die Menge der syntaktisch korrekten abstrakten Syntaxbäume spezifizieren.

Dabei unterscheiden wir zwischen *Programmen*, *Deklarationen* und *Ausdrücken*. Programme bestehen aus einem *Deklarationsteil* und einem *Hauptprogramm*. Der Deklarationsteil setzt sich aus Frame–, Methoden– und Funktionsdefinitionen zusammen, an deren Reihenfolge wir keine besonderen Anforderungen stellen. Sie ist semantisch irrelevant, denn wir werden Mehrfachdefinitionen ausschließen. Ausdrücke treten in den Rümpfen der Funktions– und Methodendefinitionen sowie im Hauptprogramm auf. Das Hauptprogramm selbst ist ein Ausdruck.

Die Begriffe *Anweisung* oder *Prozedur* definieren wir nicht, obwohl einzelne Ausdrücke den Charakter von Anweisungen haben und damit Funktionen wie Prozeduren verwendet werden können. Wir orientieren uns an Sprachen wie Lisp oder Scheme. „Anweisungen“ haben einen Wert und sind somit ebenfalls Ausdrücke.

2.1.1 Syntaktische Bereiche

Die Menge der Identifikatoren ist in vier paarweise disjunkte lexikalische Bereiche aufgeteilt. Variablennamen stammen aus dem applikativen Kern. Namen von Funktionen und formale Parameternamen fassen wir in diesem Bereich zusammen. Framenamen, Instanzvariablennamen und Methodennamen sind diejenigen Identifikatoren, die durch die objektorientierte Erweiterung zusätzlich auftreten können:

Definition 2.1

Mit *Var*, *InstVar*, *Framename* und *Message* bezeichnen wir die paarweise disjunkten lexikalischen Bereiche der *Variablen-* bzw. *Funktionsnamen*, der *Instanzvariablennamen*, der *Framenamen* und der *Methodennamen*. Mit

$$\begin{array}{lcl} x, f & \in & \mathit{Var}, \\ y & \in & \mathit{InstVar}, \\ A & \in & \mathit{Framename}, \\ m & \in & \mathit{Message}, \end{array}$$

eventuell mit Indizes, bezeichnen wir typische Elemente dieser Bereiche. Methodennamen nennen wir auch *Nachrichtennamen*. □

Als Programmkonstanten lassen wir Zahlen, Wahrheitswerte und gerade die Standardfunktionen auf diesen Bereichen zu, die wir in den im folgenden angegebenen Beispielen benötigen. *true* und *false* sind die konstanten Wahrheitswerte. Als numerische Konstanten lassen wir der Einfachheit halber den Bereich $\mathbb{N}_0$ der natürlichen Zahlen zu und gehen davon aus, daß ein *Parser* die konkrete Dezimaldarstellung einer Zahl $n \in \mathbb{N}_0$ bereits in das entsprechende Element $n \in \mathbb{N}_0$ transformiert hat. Durch die objektorientierte Erweiterung erhalten wir keine zusätzlichen Programmkonstanten.

Definition 2.2

Der syntaktische Bereich *Const* der *Programmkonstanten* ist definiert durch

$$c \in Const =_{\text{Def}} \mathbb{N}_0 + \{ true, false \} + \{ +, -, *, mod, div, \leq, <, =, and, not, or \}$$

□

Man beachte, daß die genannten Standardfunktionen einparametrig sind, d.h., daß $+(3)$ die Funktion bedeutet, die 3 zu ihrem Argument addiert. In den Beispielen werden wir der Lesbarkeit halber Standardfunktionsaufrufe wiederum in Infixnotation schreiben, also z.B. $e_1 + e_2$ statt $+(e_1)(e_2)$.

Unter den Ausdrücken zeichnen wir den Bereich *Abstraction* der λ-Abstraktionen besonders aus. In den Deklarationsteilen der im Abschnitt 2.3 definierten *abstrakten Programme* kommen nur solche als Bindungen von Identifikatoren vor, und im Kapitel 5 werden wir ihnen neben ihrer Semantik als Ausdruck eine spezielle *Abstraktionssemantik*, nämlich die entsprechende Funktion, zuordnen.

Definition 2.3

Expr, *Abstraction*, *Declaration* und *Program* sind die syntaktischen Bereiche der *Ausdrücke*, der λ-*Abstraktionen*, der *Deklarationen* und der *Programme*. Typische Elemente dieser Bereiche benennen wir, eventuell mit Indizes, mit

$$\begin{array}{lcl} e & \in & Expr, \\ a & \in & Abstraction \subseteq Expr, \\ d & \in & Declaration, \\ p & \in & Program. \end{array}$$

□

Ausdrücke, Deklarationen und Programme werden wir im folgenden Abschnitt durch syntaktische Gleichungen induktiv definieren. Bereits an dieser Stelle wollen wir die syntaktischen Bereiche um den der *verallgemeinerten Identifikatoren* ergänzen. Dieser Bereich umfaßt die gewöhnlichen Variablennamen, mit denen formale Parameter und Funktionen benannt sind, und *Methodenfunktionsnamen*. Letztere sind Paare $\langle A, m \rangle$, bestehend aus einem Framenamen A und einem Methodennamen m. Mit ihnen werden wir in abstrakten Programmen die Methoden benennen (vgl. Abschnitt 2.3, speziell Definition 2.23).

Definition 2.4

Der syntaktische Bereich *Ident* ist definiert durch

$$I \in Ident =_{\text{Def}} Var + (Framename \times Message)$$

□

Die folgende Tabelle faßt die Definition der syntaktischen Bereiche noch einmal zusammen:

Syntaktische Bereiche

x, f	$\in$	Var
y	$\in$	$InstVar$
A	$\in$	$Framename$
m	$\in$	$Message$
c	$\in$	$Const =$
		$\mathbb{N}_0 + \{ true, false \} + \{ +, -, *, mod, div, \leq, <, =, and, not, or \}$
e	$\in$	$Expr$
a	$\in$	$Abstraction$
d	$\in$	$Declaration$
p	$\in$	$Program$
I	$\in$	$Ident = Var + (Framename \times Message)$

2.1.2 Deklarationen und Programme

Im folgenden führen wir die syntaktischen Gleichungen und Bezeichnungen für die einzelnen syntaktischen Bestandteile ein und werden ihre Bedeutung informell erläutern. Der dann folgende Abschnitt zeigt zwei Beispielprogramme. Die statische Semantik eines Programmes können wir an dieser Stelle noch nicht definieren, da die Wohlgeformtheit von Programmen wesentlich von der Strategie abhängt, mit der die Vererbung durchgeführt wird. Diese *Vererbungsstrategie* legt fest, welche Instanzvariablen und welche Methoden einem Frame tatsächlich zugeordnet sind. Es gibt durchaus verschiedene plausible Vererbungsstrategien, zumindest in dem hier allgemein vorliegenden Fall der multiplen Vererbung. Überlegungen hierzu und die Definition der statischen Semantik sind Gegenstand des Kapitels 3.

Definition 2.5

Einen abstrakten Syntaxbaum

$$p ::= d\,;e$$

bezeichnen wir als *Programm* mit *Deklarationsteil* d und *Hauptprogrammausdruck* oder *Hauptteil* e . □

Definition 2.6

Einen abstrakten Syntaxbaum

$$d \;::=\; \mathit{defframe}\ A\ A^*\ y^*$$

bezeichnen wir als *Framedeklaration* mit *Framenamen* A , *lokaler Superframeliste* A^* und *lokaler Instanzvariablenliste* y^* . □

Framedeklarationen bezeichnen wir auch als *Framedefinitionen*. Wir unterscheiden die Begriffe *Definition* und *Deklaration* nicht. Die Deklaration eines Frames entspricht der eines *Record*-ähnlichen Datentyps oder einer *Struktur* im Sinne von Lisp. Daten dieses Datentyps sind gerade die *Objekte*, die als *Instanzen* des Frames gebildet werden. Die lokale Instanzvariablenliste y^* nennt einige der Komponentenvariablen, die Instanzen besitzen, und die in objektorientierten Wissensrepräsentationssprachen auch als *Slots* (*Schlitze*) bezeichnet werden. Die restlichen Instanzvariablen ergeben sich durch *Vererbung* aus den in der lokalen Superframeliste A^* genannten Frames.

Definition 2.7

Als *Methodendefinition* zum Frame A bezeichnen wir

$$\begin{array}{lcl} d & ::= & \mathit{defmethod}\ A\ m\ x\ =\ e \quad \text{bzw.} \\ d & ::= & \mathit{defmethod}\ A\ m\ ()\ =\ e. \end{array}$$

m heißt *Methodenname*, x nennen wir *formalen Parameter* der Methode, e bezeichnen wir als deren *Rumpf*. □

Eine Framedeklaration legt die lokalen Instanzvariablen eines Frames fest. Die Methodendefinitionen zu einem Frame A bestimmen entsprechend die lokalen Methoden des Frames. Frames können sowohl Instanzvariablen als auch Methoden von ihren Superframes erben. Erst durch die *Vererbungsstrategie* sind die Instanzvariablen und Methodendefinitionen eines Frames bestimmt.

In Methodenrümpfen darf ein zusätzlicher „impliziter“ Parameter *self* frei vorkommen, zum einen wie eine gewöhnliche Variable, zum anderen in den weiter unten definierten Instanzvariablenreferenzen. Methoden werden durch Nachrichten an Objekte aufgerufen; *self* bezeichnet gerade dieses Empfängerobjekt. Wir behandeln *self* wie eine gewöhnliche Variable, denn eine Methodendefinition

$$\mathit{defmethod}\ A\ m\ x\ =\ e$$

entspricht der Definition eines Funktionals $\lambda\, \mathit{self}.\, \lambda\, x\, .\, e$ mit einem Namen, der aus dem Framenamen A und dem Methodennamen m besteht. Die Abstraktion $\lambda\, \mathit{self}.\, \lambda\, x\, .\, e$ bezeichnen wir als *Methode*. Durch Anwenden der Methode auf ein Objekt entsteht $\lambda\, x\, .\, e$ mit einer zusätzlichen Bindung von *self* an das Empfängerobjekt. Diese „Closure“ bezeichnen wir als *Methodenfunktion*.

Definition 2.8

Als Funktionsdefinition mit Funktionsnamen f und *Rumpf* e bezeichnen wir

$$d \quad ::= \quad \mathit{defun}\ f\ x\ =\ e \ \mid\ \mathit{defun}\ f\ ()\ =\ e.$$

x heißt *formaler Parameter*. □

Funktionen gehören dem applikativen Teil unserer Sprache an und erhalten die übliche Bedeutung, wie sie aus imperativen Programmiersprachen bekannt ist. Es sei darauf hingewiesen,

daß Funktionen i.a. nicht seiteneffektfrei sind, und daß parameterlose Funktionen nicht unbedingt konstant sind. Ihre Rümpfe können Nachrichten enthalten, die evtl. schreibende oder lesende Zugriffe auf Instanzvariablen durchführen. Wir benutzen den Begriff *Funktion* hier im Sinne von Lisp oder Scheme, nicht im Sinne funktionaler Sprachen.

Definition 2.9

Die *Sequenz*

$$d \ ::= \ d_1 \, ; \, d_2$$

der Deklarationen d_1 und d_2 läßt die sequentielle Kombination zweier und damit endlich vieler Deklarationen zu. □

Damit haben wir die Syntax von Programmen und Deklarationsteilen eingeführt. Die Deklarationsteile bestehen aus einer Sequenz von Deklarationen, deren Reihenfolge aber keine Rolle spielt.

2.1.3 Ausdrücke

Kommen wir nun zu den Ausdrücken, die in den Rümpfen der Funktions– oder Methodendefinitionen und im Hauptteil eines Programmes auftreten können.

Definition 2.10

$$\begin{array}{lcl} a & ::= & \lambda\, x \,.\, e \ \mid \ \lambda\, () \,.\, e \\ e & ::= & a \end{array}$$

$\lambda\, x \,.\, e$ und $\lambda\, () \,.\, e$ heißen *Abstraktionen*. Den Ausdruck e in ihnen nennen wir *Rumpf*, x heißt *formaler Parameter* der Abstraktion. □

Abstraktionen bedeuten Funktionen. Funktionen können also insbesondere als Argument in Applikationen, als Ergebnis von Funktionen und Methoden oder als Inhalt von Variablen auftreten. Außerdem erlauben sie *Blockschachtelung* (vgl. Definition 2.18 und Satz 4.19 im Kapitel 4).

Definition 2.11

$$e \quad ::= \quad e_0 \, (\, e_1 \,) \ \mid \ e_0 \, (\,)$$

bezeichnen wir als *Aufruf* oder *Applikation* des *Funktionsausdrucks* e_0 . e_1 heißt *Argument* des Aufrufes. □

Kaskadierte Aufrufe der Form $e_0 \, (\, e_1 \,) \cdots (\, e_n \,)$ müssen wir uns linksgeklammert denken, d.h. als $(\cdots (\, e_0 \, (\, e_1 \,) \,) \cdots) \, (\, e_n \,)$.

Definition 2.12

$$e \quad ::= \quad \mathit{if}\ e_0\ \mathit{then}\ e_1\ \mathit{else}\ e_2$$

nennen wir *bedingten Ausdruck* mit *Bedingung* e_0 , *then*-Teil e_1 und *else*-Teil e_2 . □

Alle Resultate von e_0 außer $\mathit{false} \in \mathit{Bool}$ werden als *wahr* angesehen. Die Ausdrücke e_1 und e_2 dürfen Resultate beliebigen, auch verschiedenen Typs liefern.

Wir haben bereits darauf hingewiesen, daß einzelne Ausdrücke den Charakter von Anweisungen haben. Deshalb macht es Sinn, diese hintereinander ausführen zu können:

Definition 2.13

$$e \quad ::= \quad e_1\,;\,e_2$$

nennen wir *Hintereinanderausführung* der Ausdrücke (oder Anweisungen) e_1 und e_2 . □

Die Hintereinanderausführung $e_1\,;\,e_2$ ist selbst wieder ein Ausdruck. Der Wert von e_1 wird ignoriert; $e_1\,;\,e_2$ hat als Wert den von e_2 . Natürlich ist die Bedeutung von $(\,e_1\,;\,e_2\,)\,;\,e_3$ gleich der von $e_1\,;\,(\,e_2\,;\,e_3\,)$ und wir schreiben stattdessen $e_1\,;\,e_2\,;\,e_3$. Die Assoziativität von „ ; “ läßt sich in der im Kapitel 4 definierten denotationellen Semantik leicht beweisen.

Wir haben damit die Ausdrücke des applikativen Kerns komplett eingeführt. Im folgenden widmen wir uns den Ausdrücken, die durch den objektorientierten Teil hinzukommen. Dies sind die Nachrichten, die lesenden und schreibenden Zugriffe auf Instanzvariablen sowie die Erzeugung neuer Objekte, die wir als Instanziierung bezeichnen.

Definition 2.14

$$e \quad ::= \quad e.m$$

heißt *Nachricht*. Den Ausdruck e nennen wir auch *Empfängerausdruck* und m heißt *Nachrichtenname* oder *Methodenname* der Nachricht. □

Häufig werden die Aufrufe $e_0.m\,(e_1)$ bzw. $e_0.m\,()$ von Methodenfunktionen als Nachrichten bezeichnet. Wir *currifizieren* auch hier und bestimmen mit einer Nachricht $e_0.m$ stattdessen die aufzurufende Methodenfunktion. $e_0.m\,(e_1)$ ist damit syntaktisch eine Applikation des Funktionsausdrucks $e_0.m$ auf das Argument e_1 . Die aufzurufende Methodenfunktion hängt auf zweierlei Weise von dem Empfängerobjekt, also der Bedeutung von e_0 ab:

- In einem Programm können zu einem Methodennamen m in verschiedenen Frames verschiedene Methoden definiert sein. Die Auswahl der richtigen hängt vom Frame (dem *Typ*) von e_0 ab.
- Freie Vorkommen von *self* in Methodenrümpfen bezeichnen das Empfängerobjekt selbst. Die dem Ausdruck $e_0.m$ zugeordnete Methodenfunktion enthält also eine Bindung von *self* an das Empfängerobjekt.

Die Variable *self* wird häufig als *Pseudovariable* oder *impliziter Parameter* bezeichnet. Wir werden bereits in diesem Kapitel im Abschnitt 2.3 den Methodennamen spezielle Abstraktionen der Form $\lambda\, self\, .\, \lambda\, x\, .\, e$ bzw. $\lambda\, self\, .\, \lambda\, ()\, .\, e$ zuordnen und diesen Parameter damit explizit sichtbar machen.

Definition 2.15

Einen abstrakten Syntaxbaum

$$e \quad ::= \quad \textit{instance-of}\ A$$

bezeichnen wir als *Instanziierung* des Frames A . □

Der Ausdruck (*instance-of* A) erzeugt ein neues Objekt als Instanz des Frames A . Dieses Objekt assoziiert alle Instanzvariablennamen von A mit neu allozierten Speicherplätzen. Außerdem enthält es den Framenamen A als Typkennzeichnung. Die Liste aller Instanzvariablen des Frames A sind erst nach Durchführung der Vererbung bekannt (vgl. Kapitel 3).

Die Bindung der Speicherplätze an Werte bildet den lokalen Zustand des Objektes. Er ist nur in den Methodenfunktionen des Objektes selbst sichtbar und kann somit nur durch das Senden von Nachrichten an das Objekt gelesen oder modifiziert werden. Diesen Effekt bezeichnet man häufig auch als *Verkapselung* (engl. *encapsulation*).

Definition 2.16

$$e \quad ::= \quad self\, .\, y \ \mid\ self\, .\, y\ :=\ e$$

heißen *lesende* bzw. *schreibende Instanzvariablenreferenz.* □

Wir werden in Kapitel 3 von statisch semantisch wohlgeformten Programmen verlangen, daß Instanzvariablenreferenzen nur in Methodenrümpfen vorkommen. Diese Eigenschaft hätten wir auch in der Syntax ausdrücken können; wir haben uns aber dafür entschieden, die syntaktischen Gleichungen möglichst einfach zu halten, um Definitionen, die induktiv über den syntaktischen Aufbau von Ausdrücken erfolgen, nicht zu verkomplizieren.

Da wir die Disjunktheit der lexikalischen Bereiche *Var* und *InstVar* gefordert haben, hätten wir bei der Definition der Instanzvariablenreferenzen auf die Nennung des „impliziten Parameters“ *self* verzichten können, wie dies beispielsweise in den Lisp-Erweiterungen Flavors oder NewFlavors geschieht. Da die Semantik aber von der Bindung von *self* abhängt (vgl. Kapitel 4), haben wir uns zugunsten der Lesbarkeit von Programmen für die oben definierte Syntax entschieden. Die Variable $self \in Var$ bezeichnet in diesen Ausdrücken den impliziten Parameter, der im Methodenrümpfen für den Empfänger der entsprechenden Nachricht steht.

Zusätzlich zu den bis jetzt definierten Ausdrücken können Konstanten c und gewöhnliche Variablen x als Ausdrücke vorkommen. Die folgende Tabelle faßt die Definition der syntaktischen Gleichungen noch einmal zusammen:

Syntaktische Gleichungen

$$
\begin{array}{lcl}
e & ::= & c \mid x \mid e_1 \,;\, e_2 \mid \mathit{if}\ e_0\ \mathit{then}\ e_1\ \mathit{else}\ e_2 \mid \\
& & a \mid e_0\,(e_1) \mid e_0\,() \mid \\
& & \mathit{self}.y \mid \mathit{self}.y := e \mid e.m \mid \mathit{instance\text{-}of}\ A \\
\\
a & ::= & \lambda x\,.\,e \mid \lambda()\,.\,e \\
\\
d & ::= & \mathit{deffrime}\ A\ A^*\ y^* \mid \\
& & \mathit{defmethod}\ A\ m\ x\ =\ e \mid \mathit{defmethod}\ A\ m\ ()\ =\ e \mid \\
& & \mathit{defun}\ f\ x\ =\ e \mid \mathit{defun}\ f\ ()\ =\ e \mid d_1\,;\,d_2 \mid \\
\\
p & ::= & d\,;e
\end{array}
$$

Bemerkung 2.17

Es ist klar, wann ein Ausdruck e_0 Teilausdruck eines Ausdrucks e , einer Deklaration d oder eines Programms p ist. Wir sagen dann auch, e_0 komme in e, d bzw. p vor.
Ebenso ist klar, wann eine (gewöhnliche) Variable in einem Ausdruck, einer Deklaration oder einem Programm *frei* vorkommt. Eine Variable x wird gerade in dem Ausdruck $\lambda x\,.\,e$ und in den Deklarationen $\mathit{defun}\ f\ x\ =\ e$ und $\mathit{defmethod}\ A\ m\ x\ =\ e$ gebunden.
Funktionsnamen f sind gewöhnliche Variablenidentifikatoren, die evtl. durch eine Funktionsdefinition $\mathit{defun}\ f\ x\ =\ e$ an eine Abstraktion $\lambda x\,.\,e$ gebunden sind.
Über die Bindungen von Nachrichtennamen machen wir keine Aussage. Die Bindungen für Instanzvariablenvorkommen werden in Abschnitt 3.4 im Kapitel 3 im Zusammenhang mit der Definition instanziierbarer Frames diskutiert.

Das Ergebnis des Hauptprogrammausdruckes zusammen mit dem Zustand nach Ausführung des Programmes stellt das Programmergebnis dar. Der Zustand enthält insbesondere die Werte aller Instanzvariablen der während der Ausführung des Programmes gebildeten Instanzen.

Um in den Beispielen eine etwas lesbarere Syntax zur Verfügung zu haben, führen wir nun noch einige syntaktische Erweiterungen ein:

Definition 2.18

$$
\begin{array}{lcl}
\mathit{let}\ x\ =\ e_1\ \mathit{in}\ e_2 & \equiv_{\mathrm{Def}} & (\lambda x\,.\,e_2)\,(e_1) \\
\mathit{defun}\ f\ (x_1,\ldots,x_n)\ =\ e & \equiv_{\mathrm{Def}} & \mathit{defun}\ f\ x_1\ =\ \lambda x_2.\ldots.\lambda x_n\,.e \\
\mathit{defmethod}\ A\ m\ (x_1,\ldots,x_n)\ =\ e & \equiv_{\mathrm{Def}} & \mathit{defmethod}\ A\ m\ x_1\ =\ \lambda x_2.\ldots.\lambda x_n\,.e \\
e_0\,(e_1,\ldots,e_n) & \equiv_{\mathrm{Def}} & e_0\,(e_1)\cdots(e_n) \\
 & \equiv_{\mathrm{Def}} & (\cdots(e_0\,(e_1))\cdots)\,(e_n)
\end{array}
$$

□

2.2 Beispiele

Das erste Programm, das wir in diesem Abschnitt als Beispiel angeben wollen, liefert als Ergebnis 42 . Wir werden dieses Programm bzw. Ausschnitte davon im folgenden immer wieder benutzen, um die definierten Begriffe oder die Eigenarten objektorientierter Programme am Beispiel zeigen zu können.

Beispiel 2.19

Das folgende Programm enthält multiple Vererbung und zwei Beispiele für nicht instanziierbare Frames, nämlich A und C (vgl. Abschnitt 3.4). Es ist zudem ein relativ kleines Beispiel für ein Programm, an dem die Unterschiede der in Abschnitt 3.2.1 definierten Vererbungsstrategien deutlich werden.

```
defframe A ( ) (x);
defmethod A set-x (n) = self.x := n;
defmethod A add ( ) = self.x + self.y;

defframe B (A) (y);
defmethod B set-y (n) = self.y := n;

defframe C (A) ( );
defmethod C add ( ) = self.y + self.x;

defframe D (B, C) ( );

(λ di .
      (di . set-x (17); di . set-y (25); di . add ( ))
) (instance-of D);
```

Die beiden Methodendefinitionen für *set-x* und *set-y* zu den Frames A bzw. B benötigen wir, da wir Instanzvariablenzugriffe außerhalb der Methodenrümpfe nicht zugelassen haben. Der Hauptteil des Programmes entspricht dem folgenden *Block*:

```
let (di = instance-of D) in
      di . set-x (17); di . set-y (25);
      di . add ( );
```

Die beiden Methodendefinitionen für *add* in A und C unterscheiden sich voneinander: Einmal wird y zu x addiert, einmal x zu y . Die beiden Methoden zu *add* in A und C enthalten Referenzen auf die Instanzvariable y , die weder in A noch in C lokal vorhanden ist. Die

Framedefinition von D enthält keine lokalen Komponenten, weder Methoden noch Instanzvariablen. D erhält seine Komponenten durch Vererbung von seinen Superframes B , C und A .

Beispiel 2.20

In einem zweiten Beispiel wollen wir ein Programm angeben, das mit der Methode des „Siebes des Eratostenes“ die Primzahlen zwischen 2 und 1000 bestimmt. Wir benutzen in diesem Programm intensiv die in Definition 2.18 vereinbarten syntaktischen Erweiterungen, also Funktions– und Methodendefinitionen mit mehr als nur einem Parameter und *let*-Blöcke.

```
defframe Fmesh ( ) ( p , next ) ;
defmethod Fmesh set-p ( n ) = self.p := n ;
defmethod Fmesh set-next ( mesh ) = self.next := mesh ;
defmethod Fmesh sieve ( n , sender ) =
      if ( n mod self.p ) = 0 then false
      else self . next . sieve ( n , self ) ;

defframe Flast ( ) ( ) ;
defmethod Flast sieve ( n , sender ) =
      let ( newmesh = instance-of Fmesh ) in
            sender . set-next ( newmesh ) ;
            newmesh . set-p ( n ) ;
            newmesh . set-next ( self ) ;

defun drive ( i , n , mesh ) =
      if ( i ≤ n ) then
            mesh . sieve ( i , mesh ) ; drive ( i + 1 , n , mesh )
      else false ;

let ( mesh = instance-of Fmesh ) in
      let ( last = instance-of Flast ) in
            mesh . set-p ( 2 ) ;
            mesh . set-next ( last ) ;
            drive ( 2 , 1000 , mesh ) ;
```

Das Programmergebnis ist eine linear verkettete Liste von Instanzen des Frames *Fmesh* , die mit der im Hauptprogrammausdruck erzeugten Instanz *mesh* beginnt und mit der ebenfalls dort erzeugten Instanz des Frames *Flast* endet. Diese Liste stellt das „Sieb“ dar. Die Instanzvariablen p der einzelnen „Maschen“ enthalten nacheinander die Primzahlen zwischen 2 und 1000.
Die Vielfachen einer bereits bestimmten Primzahl p „fallen durch die Masche“, die diese Primzahl in ihrer Instanzvariablen p hält. Die nächste Primzahl ist gerade dann gefunden, wenn die Instanz *last* die Nachricht *sieve* erhält. Es ist dann eine Zahl n gefunden, die sich durch keine der Primzahlen $< n$ ganzzahlig teilen läßt. Dazu ist es nötig, die Zahlen

nacheinander durch das „Sieb“ zu schicken. Dies leistet die Funktion *drive* allgemein für die Zahlen zwischen i und n.

Als letztes Beispiel wollen wir ein kleines Programmfragment angeben, das deutlich macht, daß einem Programm im allgemeinen statisch nicht anzusehen ist, welche Methode durch eine Nachricht aufgerufen wird. Dies ist eine Konsequenz aus der Tatsache, daß wir eine ungetypte Sprache im Sinne von Lisp oder Scheme definieren. Wir lassen zu, daß Variablen Daten unterschiedlichen Typs enthalten können. Ebenso lassen wir zu, daß die beiden Zweige des Konditionals Ergebnisse unterschiedlichen Typs liefern können.

Beispiel 2.21

Betrachten wir den folgenden Programmausschnitt:

$$
\begin{array}{l}
\mathit{defframe}\ A\ (\,)\ (\,);\\
\mathit{defmethod}\ A\ m(\,) = e_1\,;\\
\\
\mathit{defframe}\ B\ (\,)\ (\,);\\
\mathit{defmethod}\ B\ m(\,) = e_2\,;\\
\\
\qquad \vdots\\
\\
(\,\mathit{if}\ b\ \mathit{then}\ (\,\mathit{instance\text{-}of}\ A\,)\ \mathit{else}\ (\,\mathit{instance\text{-}of}\ B\,)\,).m(\,);
\end{array}
$$

Gehen wir davon aus, daß b ein Ausdruck ist, dessen Wert nicht bereits statisch bestimmbar und damit konstant ist, so läßt sich statisch nicht entscheiden, ob die Methode m von A oder von B aufgerufen wird.
Dieses Beispiel zeigt überdies, daß die *generische* Auswahl der richtigen Methode bereits in Programmen ohne Vererbung nötig ist. Das dargestellte Problem ist also unabhängig von der (Bedeutung der) Vererbung.

2.3 Abstrakte Programme

In diesem Abschnitt werden wir ein Programm, das bis jetzt als abstrakter Syntaxbaum der im Abschnitt 2.1 definierten Sprache vorliegen, in eine andere syntaktische Repräsentation überführen, die wir als *abstraktes Programm* bezeichnen. Dabei vereinheitlichen wir Funktions- und Methodendefinitionen, indem wir die entsprechenden Abstraktionen an ihre Namen binden. Die Bindung repräsentieren wir durch eine endliche Abbildung, genauso wie die in den Framedeklarationen spezifizierte Bindung der Framenamen an ihre lokalen Superframe- und Instanzvariablenlisten. Der in den Methodenrümpfen frei vorkommende implizite Parameter *self* wird zu einem expliziten Parameter der an den Methodenfunktionsnamen gebundenen Abstraktion. Die Ausdrücke des Programms übernehmen wir unverändert.

Definition 2.22

Den syntaktischen Bereich *InhGraph* der *abstrakten Deklarationsteile* oder *Vererbungsgraphen* definieren wir durch

$$\begin{aligned} \gamma \in \mathit{InhGraph} =_{\mathrm{Def}} \quad & (\mathit{Ident} \xrightarrow{\mathit{fin}} \mathit{Abstraction}) \times \\ & (\mathit{Framename} \xrightarrow{\mathit{fin}} \mathit{Framename}^*) \times \\ & (\mathit{Framename} \xrightarrow{\mathit{fin}} \mathit{InstVar}^*) \end{aligned}$$

□

Mit $A \xrightarrow{\mathit{fin}} B$ bezeichnen wir die Menge der endlichen Abbildungen im Bereich $A \longrightarrow B$, also derjenigen $f \in A \longrightarrow B$, für die $\{x \in A \mid f(x) \neq \bot\}$ endlich ist.

Wir definieren nun zunächst den zu dem Deklarationsteil d eines Programmes gehörenden abstrakten Deklarationsteil, um ihn dann mit dem Hauptprogrammausdruck zu dem abstrakten Programm zusammenzusetzen.

Definition 2.23

Für beliebige Deklarationen $d \in \mathit{Declaration}$ und $\gamma \in \mathit{InhGraph}$ definieren wir die Funktion

$$\mathbf{IG}[\![\bullet]\!] : \mathit{Declaration} \longrightarrow \mathit{InhGraph} \longrightarrow \mathit{InhGraph}$$

induktiv über den Aufbau von d durch:

$$\begin{aligned}
&\mathbf{IG}[\![d_1 ; d_2]\!]\gamma && =_{\mathrm{Def}} && \mathbf{IG}[\![d_2]\!](\mathbf{IG}[\![d_1]\!]\gamma) && (1)\\
&\mathbf{IG}[\![\mathit{defframe}\ A\ A^*\ y^*]\!]\gamma && =_{\mathrm{Def}} && \langle \gamma\downarrow_1 , \gamma\downarrow_2 [A \leftarrow A^*] , \gamma\downarrow_3 [A \leftarrow y^*] \rangle && (2)\\
&\mathbf{IG}[\![\mathit{defmethod}\ A\ m\ x = e]\!]\gamma && =_{\mathrm{Def}} && \langle \gamma\downarrow_1 [\langle A, m \rangle \leftarrow \lambda \mathit{self}.\lambda\, x\, .e] , \gamma\downarrow_2 , \gamma\downarrow_3 \rangle && (3)\\
&\mathbf{IG}[\![\mathit{defmethod}\ A\ m\ () = e]\!]\gamma && =_{\mathrm{Def}} && \langle \gamma\downarrow_1 [\langle A, m \rangle \leftarrow \lambda \mathit{self}.\lambda\, ()\, .e] , \gamma\downarrow_2 , \gamma\downarrow_3 \rangle && (4)\\
&\mathbf{IG}[\![\mathit{defun}\ f\ x = e]\!]\gamma && =_{\mathrm{Def}} && \langle \gamma\downarrow_1 [f \leftarrow \lambda\, x\, .e] , \gamma\downarrow_2 , \gamma\downarrow_3 \rangle && (5)\\
&\mathbf{IG}[\![\mathit{defun}\ f\ () = e]\!]\gamma && =_{\mathrm{Def}} && \langle \gamma\downarrow_1 [f \leftarrow \lambda\, ()\, .e] , \gamma\downarrow_2 , \gamma\downarrow_3 \rangle && (6)
\end{aligned}$$

□

In Gleichung (2) wird die Bindung der Superframeliste A^* an A in der zweiten Komponente, die der lokalen Instanzvariablen y^* an A in der dritten Komponente von γ nachgetragen. Die erste Komponente bleibt unverändert. In (3) und (4) wird in der ersten Komponente die Bindung der der Methodendefinition entsprechenden Abstraktion an den Methodenfunktionsnamen $\langle A, m \rangle$ nachgetragen. Dabei wird der implizite Parameter *self* des Methodenrumpfes ein gewöhnlicher Parameter dieser Abstraktion. Die zweite und dritte Komponente von γ bleiben, wie auch in (5) und (6), unverändert. In (5) und (6) wird die zugehörige Abstraktion an den Funktionsnamen gebunden.

Damit läßt sich der abstrakte Deklarationsteil eines Programmes $p = d; e$ als Anwendung von $\mathbf{IG}[\![\bullet]\!]$ auf den gesamten Deklarationsteil d von p und das an jeder Stelle undefinierte $\gamma_\bot$ definieren:

Definition 2.24

Sei $d \in \mathit{Declaration}$ der Deklarationsteil eines Programmes. Dann heißt

$$\gamma =_{\mathrm{Def}} \mathbf{IG}\,[\![\, d \,]\!]\;\gamma_\perp \in \mathit{InhGraph}$$

abstrakter Deklarationsteil zu d. Dabei ist $\gamma_\perp =_{\mathrm{Def}} \langle \lambda I \,.\, \perp, \lambda A \,.\, \perp, \lambda A \,.\, \perp \rangle \in \mathit{InhGraph}$ ein Tripel von an jeder Stelle undefinierten Abbildungen.

□

Beispiel 2.25

Für das Programm in Beispiel 2.19 erhalten wir den folgenden abstrakten Deklarationsteil:

$$\begin{array}{lll}
\langle\ \{ & \langle A, \mathit{add}\,\rangle & \mapsto\ \lambda\,\mathit{self}\,.\,\lambda\,(\,)\,.\,\mathit{self}.x + \mathit{self}.y\ , \\
& \langle A, \mathit{set\text{-}x}\,\rangle & \mapsto\ \lambda\,\mathit{self}\,.\,\lambda\,n\,.\,\mathit{self}.x := n\ , \\
& \langle B, \mathit{set\text{-}y}\,\rangle & \mapsto\ \lambda\,\mathit{self}\,.\,\lambda\,n\,.\,\mathit{self}.y := n\ , \\
& \langle C, \mathit{add}\,\rangle & \mapsto\ \lambda\,\mathit{self}\,.\,\lambda\,(\,)\,.\,\mathit{self}.y + \mathit{self}.x \quad \}\ , \\
\\
\{ & A & \mapsto\ (\,)\ , \\
& B & \mapsto\ (\,A\,)\ , \\
& C & \mapsto\ (\,A\,)\ , \\
& D & \mapsto\ (\,B\,,C\,) \quad \}\ , \\
\\
\{ & A & \mapsto\ (\,x\,)\ , \\
& B & \mapsto\ (\,y\,)\ , \\
& C & \mapsto\ (\,)\ , \\
& D & \mapsto\ (\,) \quad \}\ \ \rangle
\end{array}$$

Die Bindung der Framenamen an ihre lokalen Superframelisten in der zweiten Komponente von γ repräsentiert gerade die *Vererbungsrelation* des Programmes, die wir im folgenden Kapitel als Relation auf den Framenamen des Programmes definieren werden. Die Vererbungsrelation bildet mit den Framenamen des Programmes als Knotenmenge einen endlichen, gerichteten geordneten Graphen. Mit dem abstrakten Deklarationsteil haben wir eine Repräsentation dieses Vererbungsgraphen vorliegen, vervollständigt um die Annotation der Knoten mit den lokalen Instanzvariablenlisten und den lokalen Methodendefinitionen. Ein abstraktes Programm besteht nun aus diesem Vererbungsgraphen und dem Hauptprogrammausdruck:

Definition 2.26

Sei $p = d; e \in \mathit{Program}$ ein Programm. Sei $\gamma = \mathbf{IG}\,[\![\, d \,]\!]\;\gamma_\perp \in \mathit{InhGraph}$ der abstrakte Deklarationsteil zu d. Dann nennen wir

$$\langle \gamma, e\,\rangle \in \mathit{InhGraph} \times \mathit{Expr}$$

abstraktes Programm zu p.

□

Definition 2.27

Wir sagen von einem Programm $p = d; e$, es sei *ohne Doppeldeklarationen*, wenn je zwei Funktions- bzw. Methodendefinitionen oder Framedeklarationen paarweise verschiedene Namen haben. Für Funktionsdefinitionen und Framedeklarationen ist klar, wann sie verschiedene Namen haben. Zwei Methodendefinitionen *defmethod* $A\ m\ \ldots$ und *defmethod* $A_1\ m_1\ \ldots$ haben verschiedene Namen, falls $\langle A, m \rangle \neq \langle A_1, m_1 \rangle$ gilt. □

Das Verbot von Doppeldeklarationen in Programmen ist eine minimale Anforderung an die statische Semantik. Wir benötigen sie bereits hier, um sicherzugehen, daß beim Übergang zum abstrakten Programm keine relevante Information verlorengeht. Wir machen diesen Übergang nur für Programme, die keine Doppeldeklarationen enthalten und weisen Programme mit Doppeldeklarationen als statisch semantisch inkorrekt ab.

Bemerkung 2.28

Programme ohne Doppeldeklarationen werden bis auf die Reihenfolge der Deklarationen eindeutig dargestellt. Der abstrakte Deklarationsteil ist ein Tripel endlicher Abbildungen, da das ursprüngliche Programm nur endlich viele Deklarationen enthielt.

Es ist klar, daß wir das ursprüngliche Programm $p = d; e$ aus dem zugehörigen abstrakten Programm leicht wiedergewinnen können, natürlich nur bis auf die Reihenfolge der Deklarationen. Dazu brauchen wir einfach nur aus den Bindungen $A \mapsto (A_1, \ldots, A_n)$ in der zweiten Komponente und $A \mapsto (y_1, \ldots, y_k)$ in der dritten Komponente die entsprechenden Framedefinitionen

$$\mathit{deffrime}\ A\ (A_1, \ldots, A_n)\ (y_1, \ldots, y_k)$$

zu erzeugen. Aus den Bindungen $f \mapsto \lambda x.e$ bzw. $\langle A, m \rangle \mapsto \lambda \mathit{self}.\lambda x.e$ in der ersten Komponente von γ erhalten wir die ursprünglichen Funktions- bzw. Methodendefinitionen zurück:

$$\mathit{defun}\ f\ x\ =\ e \quad \text{bzw.}$$

$$\mathit{defmethod}\ A\ m\ x\ =\ e\,.$$

Unsere Vererbungsgraphen $\gamma \in \mathit{InhGraph}$ sind vergleichbar mit den Methodensystemen (engl. *method systems*) aus [CP89]. Sie sind dort für den Fall der einfachen Vererbung definiert, um den Zustand eines inkrementellen objektorientierten Systems inklusive der Beziehung von Instanzen zu ihren Frames zu beschreiben. Diese Beziehung ist ein Aspekt der dynamischen Semantik eines Programmes und somit für die Repräsentation der Syntax kompletter Programme irrelevant. In abstrakten Programmen ist stattdessen zusätzlich die für die Instanziierung notwendige Information über die lokalen Instanzvariablen eines Frames vorhanden.

Mit den abstrakten Programmen haben wir eine mathematische Notation für die Syntax von Programmen eingeführt, die es erlaubt, bestimmte syntaktische Bestandteile von Programmen durch die Anwendung von Abbildungen zu „selektieren“ und bestimmte Eigenschaften von Programmen „formaler“ auszudrücken.

Die in einem Programm zusätzlich zu den gewöhnlichen Funktionsdefinitionen auftretenden Methodendefinitionen beschreiben ebenfalls Funktionen, die allerdings im Unterschied zu den gewöhnlichen nicht durch die Nennung ihres Namens, sondern durch Nachrichten an Objekte aufgerufen werden. Eine Nachricht $e_0 \,.\, m$ bedeutet den Aufruf einer der im Programm zu m definierten Methoden.

Damit bilden die Funktions- und Methodendefinitionen des Deklarationsteils d eines Programmes $p = d; e$ ein i.a. wechselseitig rekursives Gleichungssystem, das wir gerade in der ersten Komponente $\gamma{\downarrow_1}$ des zu d gebildeten *abstrakten* Deklarationsteils $\gamma = \mathbf{IG}\,[\![\, d \,]\!]\,\gamma_\perp$ repräsentieren.

Bemerkung 2.29

Sei $p = d; e$ ein Programm und sei $\gamma = \mathbf{IG}\,[\![\, d \,]\!]\,\gamma_\perp$ der zugehörige abstrakte Deklarationsteil. Dann ist für $f \in \mathit{Var}$ die Aussage

$$(\gamma{\downarrow_1}\ f) \neq \perp$$

gleichbedeutend damit, daß in d zu f eine Funktionsdefinition vorhanden ist.
Ebenso bedeutet für einen Framenamen A und einen Methodennamen m die Aussage

$$(\gamma{\downarrow_1}\ \langle A, m \rangle) \neq \perp,$$

daß zu A und m eine Methode definiert ist, d.h. daß in d eine Methodendefinition der Form *defmethod* $A\ m\ x = e$ oder *defmethod* $A\ m\ () = e$ vorkommt.

Die zweite und dritte Komponente von γ repräsentieren zusammen gerade die Framedeklarationen des Programms. Die zweite Komponente enthält die den Frames zugeordneten lokalen Superframelisten, während die dritte Komponente ihnen ihre lokalen Instanzvariablenlisten zuordnet.

Bemerkung 2.30

Ist $\gamma = \mathbf{IG}\,[\![\, d \,]\!]\,\gamma_\perp$ der abstrakte Deklarationsteil eines Programmes $p = d; e$, so gilt für jeden Framenamen A

$$(\gamma{\downarrow_2}\ A) \neq \perp \qquad \text{genau dann, wenn} \qquad (\gamma{\downarrow_3}\ A) \neq \perp.$$

Dies ist genau dann der Fall, wenn in d eine Framedeklaration

$$\mathit{defframe}\ A\ (\, A_1\,,\, \ldots\,,\, A_n\,)\ (\, y_1\,,\, \ldots\,,\, y_k\,)$$

für A vorkommt $(\, n, k \geq 0\,)$. Für eine Instanzvariable $y \in \mathit{InstVar}$ ist also die Aussage „ y kommt in $(\gamma{\downarrow_3}\ A)$ vor “ gleichbedeutend damit, daß y eine lokal zu A definierte Instanzvariable ist.

Die Menge der Framename der in d deklarierten Frames ist gerade

$$\{\, A \in \mathit{Framename} \mid (\gamma{\downarrow_2}\ A) \neq \perp \},$$

und, um noch ein Beispiel zu nennen, die Menge der Methodennamen, zu denen in d eine Methode definiert ist, können wir durch

$$\{\, m \in \mathit{Message} \mid \exists\, A \in \mathit{Framename} : (\gamma{\downarrow_1}\ \langle A, m \rangle) \ \neq\ \bot \}$$

notieren.

Es ist klar, daß wir durch eine leichte Modifikation der Abbildung $\mathbf{IG}\,[\![\,\bullet\,]\!]$ ebenso Programme mit einer SMALLTALK-ähnlicheren Syntax in die hier definierten abstrakten Programme überführen können. Dort werden die lokalen Methoden einer Klasse wie die lokalen Instanzvariablen syntaktisch innerhalb der Klassendefinition angegeben.

3

Statische Semantik und Vererbung

Es ist klar, daß nicht jedes Programm, das syntaktisch korrekt ist im Sinne der Definitionen aus Kapitel 2, auch ein vernünftiges oder *wohlgeformtes* Programm darstellt. Vielen syntaktisch richtigen Programmen wollen oder können wir keine dynamische Semantik zuordnen. Es hat sich als zweckmäßig herausgestellt, die Definition der Kontextbedingungen, die wohlgeformte Programme zusätzlich zu ihrer syntaktischen Korrektheit erfüllen müssen, von der Definition der Syntax zu trennen, denn die meisten lassen sich in der üblichen kontextfreiartigen Beschreibung der Syntax gar nicht oder nur mit erheblichem Aufwand ausdrücken. Sprachdefinitionen werden durch diese Trennung wesentlich verständlicher. Es können verschiedene, jeweils adäquate, mathematische Methoden zur Beschreibung unterschiedlicher Aspekte benutzt werden.

Eines der zentralen Merkmale unseres Ansatzes zur Beschreibung der Semantik objektorientierter Sprachen ist, daß wir, eben aus diesem Grund, auch die Definition der Vererbung von der der dynamischen Semantik trennen. Die Vererbung führen wir mittels einer syntaktischen Transformation explizit durch und erhalten so ein Programm ohne Vererbung. Das geschieht, indem jedem Frame die fehlenden Definitionen der ererbten Methoden sowie die vollständige Liste seiner Instanzvariablen hinzugefügt wird.

Ein wichtiger Begriff, der für die Durchführung der Vererbung benötigt wird, ist der des *Vererbungsgraphen eines Frames*, den wir im Abschnitt 3.1 definieren. Die Frames bilden die Knotenmenge des Vererbungsgraphen, und die *Vererbungsrelation* definiert die Kanten. Im Spezialfall der Programme mit *einfacher* Vererbung legt die Vererbungsrelation selbst bereits eine lineare Anordnung der Superframes fest. Ein Frame A *erbt* eine Methode m oder Instanzvariable y von B , falls B der *speziellste* (kleinste) Frame mit dieser Komponente in der linearen Anordnung ist. Gleichnamige Komponenten speziellerer Frames *verschatten* also die der allgemeineren.

Im Fall *multipler* Vererbung müssen wir zusätzlich zu der Verschattungsregel eine Regel angeben, die die Konflikte löst, die durch die Vererbung gleichnamiger Komponenten aus mehreren Superframes entstehen, die ihrerseits nicht in der Vererbungsrelation zueinander stehen. Dabei gibt es nun verschiedene plausible Möglichkeiten, sog. *Vererbungsstrategien*, die wir im Abschnitt 3.2 beschreiben. Sie basieren häufig auf topologischen Sortierungen oder *Linearisierungen* der Menge der Superframes eines Frames und führen so, für jeden

Frame des Programms einzeln, die Lösung der Konflikte auf die Anwendung der Verschattungsregel bei einfacher Vererbung zurück. Das heißt nicht, daß auf diese Weise ein Programm mit multipler Vererbung systematisch in eines mit einfacher Vererbung transformiert werden kann, da durch die Linearisierung Frames zueinander in Beziehung gesetzt werden, die im ursprünglichen Programmm nicht in der Vererbungsrelation standen, und das im allgemeinen sogar auf unterschiedliche Weise, je nachdem für welchen Frame die Linearisierung gerade bestimmt wird.

Schließlich führen wir dann im Abschnitt 3.3 die Vererbung von Methoden und Instanzvariablen explizit durch und erzeugen so ein Programm ohne Vererbung. Dabei setzen wir eine Vererbungsstrategie als konkret gegeben voraus. Diese bewirkt also eine *semantikdefinierende* syntaktische Transformation, denn zwei verschiedene Vererbungsstrategien können letztlich für ein und dasselbe Programm durchaus zu zwei semantisch unterschiedlichen Programmen ohne Vererbung führen.

Gerade in Wissensrepräsentationssprachen drückt die Vererbungsrelation häufig eine inhaltliche Beziehung zwischen den Objektklassen eines Expertensystems aus, die sich in den sehr technisch definierten Vererbungsstrategien der zugrundeliegenden objektorientierten Sprachen nicht immer adäquat wiederfindet. Der in der vorliegenden Arbeit gewählte Ansatz zur Beschreibung der Semantik und auch der Übersetzung objektorientierter Sprachen läßt auf einfache Weise das Experimentieren mit unterschiedlichen Vererbungsstrategien zu, da die konkret benutzte Vererbungsstrategie als „Parameter“ unseres Ansatzes in die Definition der dynamischen Semantik und der Übersetzung eingeht (vgl. Kapitel 4).

Eine wesentliche statisch semantische Einschränkung an wohlgeformte Programme können wir erst nach Durchführung der Vererbung definieren. Kommt in einer Methode m eines Frames A eine Instanzvariablenreferenz $self.y$ oder $self.y := e$ vor, so ist nicht in jedem Fall y auch Instanzvariable von A, auch nicht nach Durchführung der Vererbung. In Abschnitt 3.4 definieren wir *instanziierbare* Frames und werden in Abschnitt 3.5 von wohlgeformten Programmen u.a. verlangen, daß nur solche tatsächlich instanziiert werden. Es ist klar, daß wir dazu alle Instanzvariablen und Methoden aller Frames kennen müssen. Die Wohlgeformtheit von Programmen hängt also ebenfalls von der verwendeten Vererbungsstrategie ab.

Dies ist der Grund dafür, daß wir der Definition der statischen Semantik und der dazu notwendigen expliziten Durchführung der Vererbung ein ganzes Kapitel widmen.

3.1 Vererbungsrelation und Vererbungsgraph

Die zentralen Begriffe in diesem Abschnitt sind die *Vererbungsrelation* sowie der *Vererbungsgraph* eines Frames A in einem Programm $p = \langle \gamma, e \rangle$.

Wir definieren sie für abstrakte Programme, werden sie jedoch häufig anhand der entsprechenden konkreten Programme motivieren. Abstrakte Programme, also die Paare $\langle \gamma, e \rangle$, die aus einem abstrakten Deklarationsteil und dem Hauptprogrammausdruck bestehen, haben wir im vorigen Kapitel durch die Anwendung der Funktion $\mathbf{IG}\,[\![\bullet]\!]$ aus konkreten Programmen $p = d\,;\, e$ gewonnen.

Zunächst definieren wir den Vererbungsgraphen des Programmes, der aus der Menge der Framenamen des Programmes durch die *Vererbungsrelation* gebildet wird. Die Vererbungsrelation gibt die Beziehung zwischen einem Frame und seinen direkten Superframes wieder,

den Frames also, die in der Superframeliste einer Framedefinition

$$\mathit{defframe}\ A\ (\ A_1,\ \ldots\ ,A_n\ )\ (\ y_1,\ \ldots\ ,y_k\ )$$

auftreten. Der Vererbungsgraph ist ein endlicher gerichteter Graph, da in einem Programm nur endlich viele Framedefinitionen vorkommen. Durch die Reihenfolge der Frames in der Liste $(\ A_1,\ \ldots\ ,A_n\ )$ ist die Menge der direkten Superframes zudem total geordnet.

Definition 3.1

Sei $p = \langle \gamma, e \rangle \in \mathit{InhGraph} \times \mathit{Expr}$ ein abstraktes Programm. Dann bezeichnen wir mit $\mathit{Frames}(p)$ die Menge der Frames, genauer der Framenamen in p.

$$\mathit{Frames}(p) =_{\mathrm{Def}} \{\, A \in \mathit{Framename} \mid (\gamma{\downarrow_2}\ A) \neq \bot \}$$

$\mathit{Frames}(p)$ ist gerade die endliche Menge der Namen der in dem Programm p deklarierten Frames. Wir definieren die *Vererbungsrelation*

$$\longrightarrow_p\ \subseteq \mathit{Frames}(p) \times \mathit{Frames}(p) \quad \text{durch}$$

$$A \longrightarrow_p B \iff_{\mathrm{Def}} B \text{ kommt in } (\gamma{\downarrow_2}\ A) \text{ vor.}$$

Wir nennen dann B *direkten Superframe* von A. Mit $\longrightarrow_p^*$ bezeichnen wir die transitiv reflexive Hülle von $\longrightarrow_p$. B heißt *Superframe von* A, falls $A \longrightarrow_p^* B$ gilt. Die Menge der *Superframes von* A ist also

$$\mathit{Supers}_p(A) =_{\mathrm{Def}} \{\, B \in \mathit{Framename} \mid A \longrightarrow_p^* B \}.$$

□

Man beachte, daß A selbst Superframe von A ist, also $A \in \mathit{Supers}_p(A)$ gilt. Damit ist für jeden Frame A in $\mathit{Frames}(p)$ die Menge $\mathit{Supers}_p(A)$ eine endliche nichtleere Menge von Framenamen. Man beachte ebenfalls, daß Framenamen, die zwar in der Superframeliste einer Framedeklaration auftauchen, für die es aber keine Framedeklaration in dem Programm gibt, nicht Element der Menge $\mathit{Frames}(p)$ sind. Sie stehen damit mit keinem Frame in p in der Vererbungsrelation. Wir ignorieren sie einfach. Eine Sprachimplementation sollte in diesem Fall natürlich eine Warnung erzeugen. Es ist leicht, eine entsprechende Einschränkung an wohlgeformte Programme zu formulieren.

Definition 3.2

Sei $\mathit{Frames}(p)$ die Menge der Framenamen eines abstrakten Programmes p und sei $\longrightarrow_p\ \subseteq \mathit{Frames}(p) \times \mathit{Frames}(p)$ die Vererbungsrelation in p. Dann bezeichnen wir mit

$$\mathbf{VG}(p)\ =_{\mathrm{Def}}\ (\ \mathit{Frames}(p)\ ,\ \longrightarrow_p\ )$$

den *Vererbungsgraphen von* p. □

Der *Vererbungsgraph* von p ist also der endliche gerichtete Graph, den die Menge $Frames(p)$ als Knotenmenge zusammen mit der Vererbungsrelation $\longrightarrow_p$ bildet. Wir wollen nun, wie dies allgemein üblich ist, im folgenden nur azyklische Vererbungsgraphen betrachten. Von wohlgeformten Programmen (vgl. Abschnitt 3.5) verlangen wir, daß ihr Vererbungsgraph azyklisch ist, d.h. daß für kein Frame A in p $A \longrightarrow_p^+ A$ gilt.

Tatsächlich sind wir aber nicht an dem, im allgemeinen unzusammenhängenden Vererbungsgraphen des gesamten Programmes interessiert, sondern an dem zusammenhängenden Teilgraphen, der mit einem Frame A gerade alle Superframes von A enthält. Diesen bezeichnen wir als *Vererbungsgraphen von* A . Er enthält alle Frames, aus denen A potentiell Instanzvariablen und Methoden erbt.

Definition 3.3

Sei p abstraktes Programm und sei $Supers_p(A)$ für einen Framenamen A in p die Menge der Superframes von A. Dann heißt

$$\mathbf{VG}_p(A) =_{\text{Def}} (\, Supers_p(A)\,,\, \longrightarrow_p |_{Supers_p(A)\ \times\ Supers_p(A)}\,)$$

Vererbungsgraph von A in p . □

Der Vererbungsgraph $\mathbf{VG}_p(A)$ eines Frames A ist also der endliche gerichtete zusammenhängende Teilgraph von $\mathbf{VG}(p)$, der alle Superframes von A durch die Vererbungsrelation zueinander in Beziehung setzt. Mit $\mathbf{VG}(p)$ ist natürlich auch $\mathbf{VG}_p(A)$ azyklisch und durch die Reihenfolge der Framenamen in den direkten Superframelisten $(\gamma\downarrow_2\ A)$ geordnet.

Beispiel 3.4

Betrachten wir den folgenden Ausschnitt p aus dem Programm in Beispiel 2.19 (vgl. Kapitel 2, Seite 26), bei dem wir den Hauptprogrammausdruck weggelassen haben:

$$\begin{array}{l} \mathit{defframe}\ A\ (\,)\ (x)\,; \\ \\ \mathit{defframe}\ B\ (A)\ (y)\,; \\ \\ \mathit{defframe}\ C\ (A)\ (\,)\,; \\ \mathit{defmethod}\ C\ \mathit{add}\,(\,) = \mathit{self}.y + \mathit{self}.x\,; \\ \\ \mathit{defframe}\ D\ (B\,, C)\ (\,)\,; \end{array}$$

Der Vererbungsgraphen $\mathbf{VG}(p)$, der in diesem Beispiel gleichzeitig der Vererbungsgraph $\mathbf{VG}_p(D)$ ist, läßt sich wie in Abbildung 3.1 graphisch veranschaulichen.
Wir können uns an diesem kleinen Programmbeispiel bereits ein Problem klar machen, das im Zusammenhang mit Vererbung immer wieder auftaucht: Die Bindung von Identifikatoren, in unserem Fall der Instanzvariablen y in der Methode add des Frames C , ist dem Originalprogramm in der Regel nicht einfach anzusehen. Die Methode add enthält eine Referenz auf eine Instanzvariable y , der Frame C hat aber gar keine Instanzvariable y . Es stellt sich

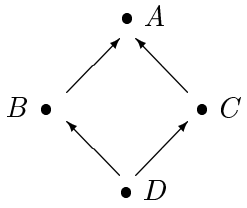


Abbildung 3.1: Vererbungsgraph $\mathbf{VG}(p) = \mathbf{VG}_p(D)$

die Frage, ob die Methode *add* denn überhaupt eine sinnvolle Bedeutung hat. Die Antwort ist positiv, zumindest wenn wir eine Vererbungsstrategie unterstellen, durch die der Frame D sowohl die Methode *add* von C als auch die Instanzvariable y von B erbt. Dann ist D *instanziierbar* (vgl. Abschnitt 3.4), und Nachrichten *add* an Instanzen von D haben eine durchaus vernünftige Bedeutung. Einen Frame wie C bezeichnet man auch als *Mixin*, da er selbst nicht instanziierbar ist, aber durch den Vererbungsmechanismus Komponenten der instanziierbaren Frames beisteuert.

Wir haben den Bereich der abstrakten Deklarationsteile *InhGraph* genannt, da ein solches

$$\begin{aligned} \gamma \in \mathit{InhGraph} =_{\text{Def}} \quad & (\mathit{Ident} \xrightarrow{\mathit{fin}} \mathit{Abstraction}) \times \\ & (\mathit{Framename} \xrightarrow{\mathit{fin}} \mathit{Framename}^*) \times \\ & (\mathit{Framename} \xrightarrow{\mathit{fin}} \mathit{InstVar}^*) \end{aligned}$$

gerade die Repräsentation eines Vererbungsgraphen darstellt, in der die Knoten zusätzlich mit den lokalen Instanzvariablenlisten und den lokal zu einem Frame definierten Methoden annotiert sind.

Definition 3.5

Sei $p = \langle \gamma, e \rangle \in \mathit{InhGraph} \times \mathit{Expr}$ ein abstraktes Programm. Wir sagen, p *habe einfache Vererbung*, falls für alle Frames $A \in \mathit{Frames}(p)$ die Superframelisten $(\gamma{\downarrow_2}\ A)$ höchstens einelementig sind. □

Wir sprechen also von Programmen mit einfacher Vererbung, wenn jeder Frame höchstens einen Superframe hat. In anderen objektorientierten Sprachen, zum Beispiel in den objektorientierten Erweiterungen Flavors oder NewFlavors von Lisp, spielt ein spezieller, interner Frame die Rolle einer Wurzel des Vererbungsgraphen, wird also implizit als Superframe all derjenigen Frames definiert, die eine leere Superframeliste haben. Programme dieser Sprachen haben einfache Vererbung, wenn jeder Frame *genau einen* Superframe besitzt. In diesem Fall haben alle Programme einen zusammenhängenden Vererbungsgraphen, und im Spezialfall von Programmen mit einfacher Vererbung bildet dieser Graph einen *Baum*. Daß dies auch

in unserer Sprache für die interessanten zusammenhängenden Teilgraphen von $\mathbf{VG}(p)$ gilt, wollen wir im folgenden bemerken. Sei dazu

$$\mathit{Subframes}_p(A) =_{\text{Def}} \{ B \in \mathit{Framename} \mid B \longrightarrow_p^* A \}$$

die Menge aller Subframes von A in p.

Bemerkung 3.6

Sei $p = \langle \gamma, e \rangle \in \mathit{InhGraph} \times \mathit{Expr}$ ein abstraktes Programm mit einfacher Vererbung. Dann ist für jeden Frame A in p der zusammenhängende Teilgraph

$$(\mathit{Subframes}_p(A) , \longrightarrow_p^{-1})$$

des Vererbungsgraphen $\mathbf{VG}(p)$ ein Baum mit Wurzel A.

Natürlich folgt daraus umgekehrt sofort, daß für jeden Frame A in p der Vererbungsgraph $\mathbf{VG}_p(A)$ eine Kette ist, d.h. daß die Menge $\mathit{Supers}_p(A)$ bereits durch die Vererbungsrelation $\longrightarrow_p$ *linear geordnet* ist. Dies ist eine entscheidende Eigenschaft, die Programme mit einfacher Vererbung von solchen mit multipler Vererbung unterscheidet. Die Vererbungsrelation $\longrightarrow_p$ ist dem Programm syntaktisch anzusehen, da die Superframes in den Framedefinitionen genannt sind. Im Fall einfacher Vererbung lassen sich also die Komponenten, d.h. die Instanzvariablen und Methoden eines Frames, auf sehr einfache Weise induktiv über den syntaktischen Aufbau des Programmes definieren: Ein Frame A hat eine Komponente y oder m, falls y oder m in A lokal definiert ist, oder falls der evtl. vorhandene Superframe die Komponente y oder m hat. Damit läßt sich Vererbung *denotationell* definieren [Red88] [CP89] [Kam88]. Unabhängig voneinander haben [Red88] und [CP89] für SMALLTALK-ähnliche Sprachen mit einfacher Vererbung denotationelle Semantiken angegeben, die das eben beschriebene Verfahren zur Bestimmung der Komponenten eines Frames, genauer der Komponenten von Instanzen, durch Fixpunktbildung modellieren. Noch deutlicher wird dies in [Kam88], ebenfalls am Beispiel einer denotationellen Semantik für SMALLTALK.

Im Fall multipler Vererbung ist dies nicht so einfach möglich. Es tritt ein Konflikt auf, wenn mehr als ein Superframe die gleiche Komponente „hat“. Die naive Lösung in diesem Fall, nämlich das Heranziehen der lokalen Ordnung der direkten Superframes zur Auflösung des Konfliktes, führt im allgemeinen zu dem unerwünschten Effekt, daß Komponenten *spezi-ellerer* Frames durch die *allgemeinerer* verschattet werden können. Wir werden im folgenden Abschnitt ein Beispiel angeben, wo dies der Fall ist. Diese Problematik hat zur Diskussion unterschiedlicher *Vererbungsstrategien* im Fall der multiplen Vererbung geführt [Sym86], von denen wir zwei im folgenden Abschnitt exemplarisch definieren wollen.

3.2 Vererbungsstrategie

Zum Verständnis objektorientierter Programme ist es nötig, *alle* Komponenten, also alle Instanzvariablen und Methoden aller Frames zu kennen. Wir haben im vorigen Abschnitt motiviert, daß diese im Spezialfall einfacher Vererbung bereits festliegen. Die Begriffe aus dem vorigen Abschnitt reichen dann aus, um die Instanzvariablen und Methoden eines Frames zu

bestimmen. Kern dieser Aussage ist die Tatsache, daß die Vererbungsrelation $\longrightarrow_p$ in diesem Fall die Menge $Supers_p(A)$ bereits linear anordnet. Dadurch ist die einzig vernünftige Strategie zur Bestimmung der Komponenten eines Frames dadurch festgelegt, daß Komponenten *speziellerer* Frames diejenigen *allgemeinerer* verschatten.

Definition 3.7

Sei $p = \langle \gamma, e \rangle$ ein abstraktes Programm, sei A ein Frame in p. Wir sagen, A *besitzt* eine Instanzvariable y *lokal*, falls y in der Liste $(\gamma\downarrow_3\ A)$ vorkommt, falls also in p zu A direkt eine Instanzvariable y definiert ist. □

Dies ist genau dann der Fall, wenn y in der direkten Instanzvariablenliste $(y_1, \ldots, y_k)$ der Framedefinition

$$defframe\ A\ (A_1, \ldots, A_n)\ (y_1, \ldots, y_k)$$

von A vorkommt. Analog definieren wir, wann A eine Methode zu einem Methodennamen $m \in Message$ lokal besitzt. Dies ist gerade dann der Fall, wenn eine Methodendefinition

$$defmethod\ A\ m\ x\ =\ e$$

in p vorkommt, eventuell auch ohne formalen Parameter:

Definition 3.8

Sei $p = \langle \gamma, e \rangle$ ein abstraktes Programm, sei A ein Frame in p. Sei $m \in Message$. Wir sagen, A *besitzt lokal eine Methode* $\lambda self.\lambda x.e$ bzw. $\lambda self.\lambda().e$ *zu* m, falls

$$(\gamma\downarrow_1\ \langle A, m \rangle)\ =\ \lambda self.\lambda x.e \quad \text{bzw.}$$

$$(\gamma\downarrow_1\ \langle A, m \rangle)\ =\ \lambda self.\lambda().e$$

gilt. □

Im Fall einfacher Vererbung, allgemeiner sogar immer dann, wenn eine lineare Anordnung der Menge $Supers_p(A)$ vorliegt, können wir nun definieren, wann ein Frame eine Instanzvariable oder Methode besitzt. Dazu dient die folgende Definition, die das Vorhandensein einer linearen Anordnung von $Supers_p(A)$ voraussetzt.

Definition 3.9

Sei $p = \langle \gamma, e \rangle$ ein abstraktes Programm, sei A ein Frame in p. Sei ferner $Supers_p(A) = \{A_1, \ldots, A_n\}$ die Menge der Superframes von A und $\langle A_{i_1}, \ldots, A_{i_n} \rangle$ eine lineare Anordnung von $Supers_p(A)$ derart, daß $A_{i_1} = A$ gilt.
Wir sagen dann, A *besitzt eine Instanzvariable* y bzw. *eine Methode* a *zu* m, falls ein kleinster (*speziellster*) Frame A_{i_k} in der linearen Anordnung existiert, der y bzw. a zu m lokal besitzt. □

Da wir vorausgesetzt haben, daß A selbst kleinster Frame in der linearen Anordnung von $Supers_p(A)$ ist, besitzt A natürlich alle seine direkten Komponenten, also die direkt zu A definierten Instanzvariablen und Methoden. Im Fall einfacher Vererbung liegt eine natürliche lineare Anordnung von $Supers_p(A)$ vor, falls $\mathbf{VG}_p(A)$ azyklisch ist. Hier greift unsere Voraussetzung an den Vererbungsgraphen $\mathbf{VG}(p)$ aus dem vorigen Abschnitt. Bei einfacher Vererbung reicht somit Definition 3.9 zur Bestimmung der Komponenten der Frames aus.

Um dasselbe Verfahren auch im Falle multipler Vererbung zur Bestimmung der Instanzvariablen und Methoden eines Frames verwenden zu können, benötigen wir auch in diesem Fall eine plausible lineare Anordnung der Superframes $Supers_p(A)$, die sich an der Vererbungsrelation orientiert. In den objektorientierten Erweiterungen Flavors und NewFlavors von Lisp und damit auch in BABYLON werden topologische Sortierverfahren benutzt, um eine *Linearisierung* des Vererbungsgraphen $\mathbf{VG}_p(A)$ zu bestimmen. Diesem Vorgehen widmen wir uns im folgenden Abschnitt.

3.2.1 Linearisierung des Vererbungsgraphen

Wir wollen nun zwei Beispiele topologischer Sortierungen angeben, die die Menge der Superframes von A linear anordnen. Beide basieren letztlich auf der Besuchsreihenfolge der Knoten in $\mathbf{VG}_p(A)$, die wir dadurch erhalten, daß dieser Graph zunächst in die Tiefe und dann von links nach rechts (*depth first left to right*) traversiert wird. Da $\mathbf{VG}_p(A)$ kein Baum, sondern im allgemeinen ein azyklischer gerichteter Graph ist, treten in dieser Besuchsreihenfolge Frames mehrfach auf. Die beiden *Linearisierungen*, die wir im folgenden angeben werden, unterscheiden sich dadurch, daß im Falle mehrfach auftretender Frames andere Vorkommen gestrichen werden. Sie stammen aus den Definitionen der objektorientierten Erweiterungen Flavors [Sym85] bzw. NewFlavors [Sym86] von Lisp, die Implementierungssprachen von BABYLON sind. Ihre Vererbungsstrategie bestimmt damit jeweils auch die Vererbungsstrategie und damit die Semantik von BABYLON.

Das erste Linearisierungsverfahren bestimmt die lineare Anordnung $\mathbf{Lin}_p(A)$ des Vererbungsgraphen $\mathbf{VG}_p(A)$ durch die *depth-first-left-to-right*-Besuchsreihenfolge der Knoten in $\mathbf{VG}_p(A)$ *ohne Wiederholungen*. Dies entspricht dem folgenden Vorgehen:

- Wir traversieren $\mathbf{VG}_p(A)$, startend bei A , zunächst in die Tiefe und dann jeweils von links nach rechts, also *depth-first-left-to-right*, und notieren die Knoten in der Reihenfolge, in der wir sie besuchen.

- Treten in der dabei enstehenden Liste von Frames einzelne Frames mehrfach auf, so lassen wir nur das jeweils linkeste Vorkommen stehen, wir streichen also die *wiederholt* besuchten Knoten.

Da $\mathbf{VG}_p(A)$ zusammenhängend ist, erhalten wir so tatsächlich eine lineare Anordnung aller Superframes von A .

Beispiel 3.10

Betrachten wir wiederum einen Ausschnitt p aus unserem Beispielprogramm.

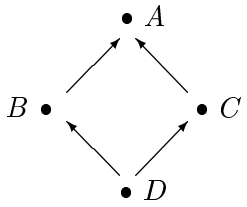
```
defframe A ( ) (x);
defmethod A set-x n = self.x := n;
defmethod A add ( ) = self.x + self.y;

defframe B ( A ) (y);
defmethod B set-y n = self.y := n;

defframe C ( A ) ( );
defmethod C add ( ) = self.y + self.x;

defframe D ( B , C ) ( );
```

Den Vererbungsgraphen $\mathbf{VG}(p) = \mathbf{VG}_p(D)$ geben wir der Deutlichkeit halber noch einmal an:



Wir wollen die Linearisierung $\mathbf{Lin}_p(D)$ des Vererbungsgraphen $\mathbf{VG}_p(D)$ bestimmen. Die *depth-first-left-to-right*-Besuchsreihenfolge der Knoten dieses Graphen ergibt die Knotenliste $\langle D, B, A, C, A \rangle$. Streichen wir nun das rechte Vorkommen von A , so erhalten wir als Linearisierung $\mathbf{Lin}_p(D) = \langle D, B, A, C \rangle$.
Sowohl A als auch C besitzen lokal eine Methode *add* . Die beiden Methoden unterscheiden sich. Einmal wird y zu x und einmal x zu y addiert. Betrachten wir nun die Linearisierung $\mathbf{Lin}_p(D)$ des Vererbungsgraphen $\mathbf{VG}_p(D)$, so stellen wir fest, daß dort A vor C auftritt, daß also, von D aus gesehen, A *spezieller* als C ist. Damit erbt D also *add* von A , die Methode *add* von C wird durch die aus A verschattet.

Das Problem entsteht, weil die Strategie zur Bestimmung von $\mathbf{Lin}_p(D)$ die durch die Vererbungsrelation in $\mathbf{VG}_p(D)$ gegebene Ordnung nicht erhält. Dort nämlich taucht C vor A auf, ist also *spezieller* als A .

In der Definition von NewFlavors [Sym86] ist daher eine andere Strategie zur Linearisierung von $\mathbf{VG}_p(A)$ festgelegt worden, die wir als zweites Beispiel anführen wollen. Sie basiert ebenfalls auf einer topologischen Sortierung $\mathbf{Lin}_p(A)$ des Vererbungsgraphen $\mathbf{VG}_p(A)$, diesmal allerdings auf eine Weise, die die durch die Vererbungsrelation gegebene Ordnung erhält. Da die lokale Ordnung der direkten Superframes ebenfalls erhalten werden soll, ist mit dieser Strategie nunmehr nicht mehr jeder Vererbungsgraph *linearisierbar*, die Vererbungsstrategie sortiert dort Programme mit nicht linearisierbarem Vererbungsgraphen als *nicht wohlgeformt* aus.

In [Sym86] ist in den beiden folgenden Regeln formuliert, welchen Anforderungen die Linearisierung $\mathbf{Lin}_p(A)$ genügen soll:

- Sind B und C Superframes von A, für die $B \longrightarrow_p^* C$ gilt, so tritt B in der linearen Anordnung $\mathbf{Lin}_p(A)$ vor C auf. Die durch die Vererbungsrelation gegebene Ordnung der Superframes von A wird also erhalten.

- Tritt B in der Superframeliste von A vor C auf, so auch in $\mathbf{Lin}_p(A)$. Die durch die lokale Ordnung der Superframelisten gegebene Ordnung wird also ebenfalls erhalten.

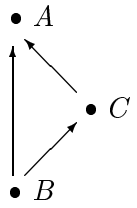
Der Fall, daß verschiedene lineare Anordnungen möglich sind, die den beiden ersten Regeln entsprechen, tritt häufig auf. Es ist ebenfalls klar, daß nicht für jeden Vererbungsgraphen eine lineare Anordnung existiert, die diesen beiden Regeln genügt. Beispiele dafür lassen sich leicht konstruieren:

Beispiel 3.11

Als Beispiel für einen nichtlinearisierbaren Vererbungsgraphen betrachten wir die Framedefinitionen

defframe A ( ) ( );
defframe C (A) ( );
defframe B (A , C) ( );

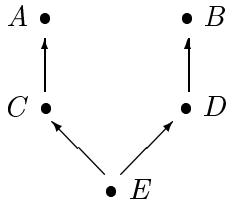
und den daraus entstehenden Vererbungsgraphen $\mathbf{VG}_p(B)$



Die möglichen linearen Anordnungen von $\mathbf{VG}_p(B)$, deren kleinstes Element B ist, sind $\langle B, A, C \rangle$ und $\langle B, C, A \rangle$. Die erste widerspricht Regel 1 und die zweite Regel 2.

Beispiel 3.12

Auch für die Existenz verschiedener möglicher linearer Anordnungen wollen wir ein kleines Beispiel angeben. Der Vererbungsgraph



läßt drei verschiedene lineare Anordnungen von $\mathbf{VG}_p(E)$ zu, die den oben genannten Regeln entsprechen. Dies sind $\langle E, C, A, D, B \rangle$, $\langle E, C, D, A, B \rangle$ und $\langle E, C, D, B, A \rangle$. Die *depth-first-left-to-right*-Strategie liefert in diesem Beispiel, ebenso wie die im folgenden definierte Strategie aus [Sym86], die erste Linearisierung $\langle E, C, A, D, B \rangle$.

Die Situation ist in [Sym86] dadurch noch etwas komplizierter, daß dort lokale Anordnungen der direkten Superframes explizit spezifiziert werden können, die von der durch die Reihenfolge ihres Auftretens gegebenen abweichen. In dem hier vorliegenden einfacheren Fall können wir $\mathbf{Lin}_p(A)$ wiederum durch eine topologische Sortierung von $\mathbf{VG}_p(A)$ bestimmen:

- Wir traversieren $\mathbf{VG}_p(A)$, startend bei A , zunächst in die Tiefe und dann jeweils von links nach rechts, also *depth-first-left-to-right*, und notieren die Knoten in der Reihenfolge, in der wir sie besuchen.

- Treten in der dabei enstehenden Liste von Frames einzelne Frames mehrfach auf, so lassen wir nur das jeweils *rechteste* Vorkommen stehen, wir streichen also die *vorherigen* Besuche allgemeinerer Frames.

Das so definierte Verfahren bestimmt also wieder eine, eindeutig bestimmte, lineare Anordnung $\mathbf{Lin}_p(A)$ aller Superframes von A . Man beachte, daß diese Sortierung auch für „nicht linearisierbare“ Vererbungsgraphen funktioniert. Sie bestimmt dann aber eine Anordnung der Superframes von A , die den Anforderungen der beiden oben genannten Regeln nicht entspricht. Die lokale Ordnung in den Superframelisten wird nicht erhalten. In dem Beispiel 3.11 liefert dieses Verfahren die Linearisierung $\mathbf{Lin}_p(B) = \langle B, C, A \rangle$, denn wir erhalten als Besuchsreihenfolge der Knoten $\langle B, A, C, A \rangle$. Auf ähnliche Weise ist die Vererbungsstrategie auch in [KS87] definiert.

In unserem Beispielprogramm erhalten wir nun für $\mathbf{VG}_p(D)$ aus der Besuchsreihenfolge $\langle D, B, A, C, A \rangle$ die Linearisierung $\langle D, B, C, A \rangle$, in der C vor A auftritt. D erbt also nun die Methode für *add* aus dem *spezielleren* Frame C .

3.2.2 Vernünftige Vererbungsstrategien

Wir haben im letzten Abschnitt zwei Beispiele von Vererbungsstrategien beschrieben, die auf topologischen Sortierungen der Menge $Supers_p(A)$ basieren. Die durch die Sortierung erzeugte lineare Anordnung $\mathbf{Lin}_p(A)$ der Menge $Supers_p(A)$ haben wir im Sinne von Definition 3.9 benutzt, um diejenigen Instanzvariablen und Methoden zu bestimmen, die ein Frame in p *besitzt*.

Dies sind nicht die einzigen denkbaren Vererbungsstrategien. So wird beispielsweise speziell für BABYLON-Frames in [Bre87] eine Vererbungsstrategie durch prädikatenlogische Formeln spezifiziert. Die Situation, die die zusätzlichen Konflikte bei multipler Vererbung hervorruft, daß nämlich ein Frame eine Komponente aus zwei verschiedenen, nicht zueinander in Vererbungsrelation stehenden Frames erben soll, wird dort als *mehrdeutig* (engl. *ambiguous*) bezeichnet, und der Frame erbt *keine* der beiden Komponenten. Auch dies ist eine durchaus plausible Strategie, die den weiter unten definierten Anforderungen an eine *vernünftige* Vererbungsstrategie genügt.

Diesem Begriff wenden wir uns nun zu. Wir abstrahieren wieder von den konkreten Vererbungsstrategien des vorigen Abschnittes und definieren zunächst, was wir im Rahmen dieser Arbeit unter einer *Vererbungsstrategie* verstehen wollen. Letztlich kommt es uns darauf an, daß wir, gegeben ein abstrakter Deklarationsteil $\gamma \in \mathit{InhGraph}$ eines Programmes $p = \langle \gamma, e \rangle$, auf vernünftige Weise in endlicher Zeit zu jedem Frame $A \in \mathit{Frames}(p)$ alle Instanzvariablen von A und alle zu A in γ definierten Methoden bestimmen können. Wir definieren deshalb zunächst ganz allgemein eine *Vererbungsstrategie* als Paar effektiv berechenbarer Abbildungen, die dies für die Instanzvariablen bzw. die Methoden leisten:

Definition 3.13

Als *Vererbungsstrategie* bezeichnen wir ein Paar $\langle \mathit{instvars}, \mathit{methods} \rangle$ effektiv berechenbarer Abbildungen

$$\begin{aligned} \mathit{instvars} &\in \mathit{InhGraph} \longrightarrow (\, \mathit{Framename} \xrightarrow{\mathit{fin}} \mathit{InstVar}^* \,) \\ \mathit{methods} &\in \mathit{InhGraph} \longrightarrow (\, (\, \mathit{Framename} \times \mathit{Message} \,) \xrightarrow{\mathit{fin}} \mathit{Abstraction} \,), \end{aligned}$$

die zu jedem $\gamma \in \mathit{InhGraph}$ und zu jedem Frame A die Liste $(\mathit{instvars}\ \gamma\ A)$ der Instanzvariablen von A und die Bindung $(\mathit{methods}\ \gamma\ \langle A, m \rangle)$ für die Methodenfunktionsnamen $\langle A, m \rangle$ liefern. □

Natürlich wollen wir nicht jedes beliebige Paar $\langle \mathit{instvars}, \mathit{methods} \rangle$ von Abbildungen als Vererbungsstrategie zulassen. Im folgenden motivieren wir deshalb Einschränkungen an *vernünftige Vererbungsstrategien*, die dann in Definition 3.14 zusammengefaßt werden.

Zunächst wollen wir verlangen, daß die Vererbungsstrategie keine zusätzlichen, in γ nicht bereits vorhandenen Frames erzeugt. Dies ist eine Einschränkung an die Abbildung *instvars*. Jeder Frame A, für den $(\mathit{instvars}\ \gamma\ A)$ definiert ist, soll bereits Frame des ursprünglichen Programmes gewesen sein. Umgekehrt wollen wir sicher gehen, daß *instvars* auch für jeden Frame $A \in \mathit{Frames}(p)$ eine gültige Instanzvariablenliste definiert. Diese kann unter Umständen leer sein, sie soll jedenfalls nicht undefiniert sein. Fassen wir diese beiden Einschränkungen zusammen, so verlangen wir:

- Die Instanzvariabalenliste $(\mathit{instvars}\ \gamma\ A) \in \mathit{InstVar}^*$ ist genau dann definiert, wenn $A \in \mathit{Frames}(p)$ gilt. $(\mathit{instvars}\ \gamma)$ definiert also eine endliche Abbildung, deren Urbild genau die Menge $\mathit{Frames}(p)$ ist. Ist $(\mathit{instvars}\ \gamma\ A) = y^*$ definiert, so enthält y^* keine Instanzvariable mehrfach.

Ebenfalls soll ausgeschlossen werden, daß die Vererbungsstrategie Methoden zu nicht vorhandenen Frames oder syntaktisch falsche Methoden erzeugt. Beide Forderungen definieren Einschränkungen an die erlaubten Abbildungen *methods*:

- Die Abbildung $(\mathit{methods}\ \gamma)$ ist nur für endlich viele $\langle A, m \rangle$ definiert. Außerdem verlangen wir, daß $A \in \mathit{Frames}(p)$ ist, falls $(\mathit{methods}\ \gamma\ \langle A, m \rangle)$ definiert ist, und daß die an $\langle A, m \rangle$ gebundene Abstraktion von der Form $\lambda\, \mathit{self}.\, \lambda\, x.\, e$ oder $\lambda\, \mathit{self}.\, \lambda\, ().\, e$ ist.

Man beachte, daß wir nicht verlangen, daß zu dem Methodennamen m in γ bereits eine Methodendefinition vorhanden war. Wir lassen also zu, daß die Vererbungsstrategie dem

Programm Methodendefinitionen hinzufügt. So erlauben wir beispielsweise die implizite Definition von Methoden für den lesenden und schreibenden Zugriff auf Instanzvariablen, denn die Vererbungsstrategie könnte für Instanzvariablen y eines Frames A die Methodendefinitionen

$$\begin{aligned} &\mathit{defmethod}\ A\ y\ (\,)\ =\ \mathit{self}.y\,; \\ &\mathit{defmethod}\ A\ \textit{set-y}\ x\ =\ \mathit{self}.y\ :=\ x\,; \end{aligned}$$

implizit hinzufügen, so daß

$$(\ \mathit{methods}\ \gamma\ \langle A, \textit{set-y}\,\rangle\ )\ =\ \lambda\,\mathit{self}\,.\,\lambda\,x\,.\,\mathit{self}.y := x$$

definiert sein kann, obwohl in γ selbst keine Bindung für eine Methode mit dem Namen *set-y* vorhanden war.

Im vorigen Abschnitt haben wir von *Linearisierungen* $\mathbf{Lin}_p(A)$ des Vererbungsgraphen $\mathbf{VG}_p(A)$ verlangt, daß der Frame A selbst jeweils der kleinste (speziellste) Frame in der durch $\mathbf{Lin}_p(A)$ gegebenen linearen Anordnung der Superframes von A sein soll. Dies führt dazu, daß A alle seine *lokalen* Instanzvariablen und Methoden tatsächlich *besitzt*. Es ist plausibel, dies auch allgemein von vernünftigen Vererbungsstrategien zu verlangen. Wir fordern also:

- Besitzt A eine Instanzvariable y lokal, so kommt diese Instanzvariable auch in der Liste $(\mathit{instvars}\ \gamma\ A)$ vor.
- Besitzt A lokal eine Methode zu m, so ist $(\mathit{methods}\ \gamma\ \langle A, m\,\rangle)$ definiert.

Beschränken wir uns auf die in dieser Arbeit definierte Sprache, so ist es durchaus vernünftig, zusätzlich zu verlangen, daß die Abstraktionen $(\mathit{methods}\ \gamma\ \langle A, m\,\rangle)$ und $(\gamma{\downarrow_1}\ \langle A, m\,\rangle)$ gleich sind. Wir sollen dies aber nicht fordern, um nicht von vornherein weitergehende Sprachbestandteile wie z.B. die *Methodenkombination* auszuschließen. Sprachen wie Flavors oder auch BABYLON erlauben die Definition sog. *Dämonen* (engl. *demons*) zu einem Methodennamen m, die in einer durch die Vererbungsstrategie bestimmten Reihenfolge *vor* oder *nach* der *Primärmethode* ausgeführt werden. Die Rümpfe der durch die Durchführung der Vererbung entstehenden *kombinierten* Methodenfunktionen unterscheiden sich dann von denen der ursprünglich im Programm vorhanden Methoden. Nähere Informationen zur Behandlung der Methodenkombination in dem hier vorgestellten Ansatz finden sich im Kapitel 8 in [CdV89] oder auch in [AGS88].

Definition 3.14 faßt nun die erwähnten Anforderungen an die Abbildungen *instvars* und *methods* zusammen. *Vernünftige Vererbungsstrategien* sind gerade solche, die alle o.a. Forderungen erfüllen.

Definition 3.14

Sei $vs = \langle \mathit{instvars}, \mathit{methods}\,\rangle$ eine Vererbungsstrategie. Sei $p = \langle \gamma, e\,\rangle$ ein abstraktes Programm. Dann heißt vs *vernünftig*, falls alle der folgenden Bedingungen erfüllt sind:

(1) $(\mathit{instvars}\ \gamma\ A)$ ist definiert $\iff$ $A \in \mathit{Frames}(p)$.

(2) $(\mathit{instvars}\ \gamma\ A)$ enthält, falls definiert, keine Instanzvariable mehrfach.

(3) $(methods\ \gamma)$ ist für höchstens endlich viele $\langle A, m \rangle$ definiert. Ist $(methods\ \gamma\ \langle A, m \rangle)$ für ein $\langle A, m \rangle$ definiert, dann gilt $A \in Frames(p)$ und $(methods\ \gamma\ \langle A, m \rangle)$ ist eine Abstraktion der Form $\lambda\, self\,.\, \lambda\, x\,.\, e$ oder $\lambda\, self\,.\, \lambda\,(\,)\,.\, e$ für ein $e \in Expr$.

(4) Besitzt A eine Instanzvariable y lokal, so kommt y in $(instvars\ \gamma\ A)$ vor.

(5) Besitzt A lokal eine Methode zu m , so ist $(methods\ \gamma\ \langle A, m \rangle)$ definiert.

□

Die Vererbungsstrategien aus dem vorigen Abschnitt, deren Kern durch die Bestimmung einer Linearisierung $\mathbf{Lin}_p(A)$ des Vererbungsgraphen $\mathbf{VG}_p(A)$ gegeben war, lassen sich als *vernünftige Vererbungsstrategien* im Sinne von Definition 3.14 definieren. Wir benutzen dazu $\mathbf{Lin}_p(A)$ und Definition 3.9, um jedem Frame A die Liste der Instanzvariablen zuzuordnen, die A *besitzt*. Jedem Methodenfunktionsnamen $\langle A, m \rangle$ ordnen wir die Methode $a \in Abstraction$ zu, die A zu m *besitzt*.

Sei also $p = \langle \gamma, e \rangle$ abstraktes Programm und sei für jeden Frame $A \in Frames(p)$ die *Linearisierung* $\mathbf{Lin}_p(A)$ des Vererbungsgraphen $\mathbf{VG}_p(A)$ von A gegeben. Mit $\mathbf{Lin}_p(A)$ liegt eine lineare Anordnung der Menge $Supers_p(A)$ der Superframes von A vor. Unter diesen Voraussetzungen spezifiziert Definition 3.9, wann A eine Instanzvariable y *besitzt*. Damit läßt sich die Abbildung

$$instvars \ \in \ InhGraph \longrightarrow (\, Framename \xrightarrow{fin} InstVar^* \,)$$

durch

$$instvars\ \gamma\ A\ =_{\mathrm{Def}} \begin{cases} y^* & \text{, falls } A \text{ ein Frame in } \gamma \text{ und } y^* \text{ die Liste der Instanzvariablen ist, die } A \textit{ besitzt}, \\ \bot & \text{, sonst} \end{cases}$$

definieren. Ist A ein Frame in γ , der keine Instanzvariable besitzt, so meint $(instvars\ \gamma\ A)$ die leere Instanzvariablenliste $\langle\,\rangle$. Sei ferner $m \in Message$ ein Methodenname. Dann spezifiziert Definition 3.9 ebenfalls, wann A eine Methode $\lambda\, self\,.\, \lambda\, x\,.\, e \in Abstraction$ oder $\lambda\, self\,.\, \lambda\,(\,)\,.\, e \in Abstraction$ zu m *besitzt*. Die Abbildung

$$methods \ \in \ InhGraph \longrightarrow (\,(\, Framename \times Message\,) \xrightarrow{fin} Abstraction\,)$$

können wir also wie folgt angeben:

$$methods\ \gamma\ \langle A, m \rangle\ =_{\mathrm{Def}} \begin{cases} a & \text{, falls } A \text{ zu } m \text{ die Methode } a \in Abstraction \textit{ besitzt}, \\ \bot & \text{, sonst} \end{cases}$$

Für Framenamen $A \notin Frames(p)$ bleibt $(instvars\ \gamma\ A)$ undefiniert. Das gleiche gilt für $(methods\ \gamma\ \langle A, m \rangle)$, falls A in γ keine Methode zu m *besitzt*.

Wir wollen uns zum Abschluß dieses Abschnittes klar machen, daß die durch diese beiden Abbildungen gegebene Vererbungsstrategie $\langle instvars, methods \rangle$ tatsächlich vernünftig

ist. Zunächst stellen wir fest, daß in beiden im vorigen Abschnitt definierten Linearisierungsstrategien die Linearisierung $\mathbf{Lin}_p(A)$ effektiv bestimmbar ist, so daß entscheidbar ist, ob ein Frame A eine Instanzvariable y oder eine Methode zu m *besitzt*. Damit sind *instvars* und *methods* effektiv berechenbar. $\langle \mathit{instvars}, \mathit{methods} \rangle$ ist also eine Vererbungsstrategie.

Ebenso schnell wird klar, daß sowohl $(\mathit{instvars}\ \gamma)$ als auch $(\mathit{methods}\ \gamma)$ endliche Abbildungen sind, die für A bzw. $\langle A, m \rangle$ höchstens dann definiert sein können, wenn $A \in \mathit{Frames}(p)$ gilt, denn nur solche Frames *besitzen* Instanzvariablen oder Methoden. Daß in $(\mathit{instvars}\ \gamma\ A)$ keine Instanzvariablen mehrfach vorkommen sollen, versteht sich von selbst. Da $(\mathit{methods}\ \gamma\ \langle A, m \rangle)$, falls definiert, einer bereits im ursprünglichen Programm vorhandenen Methode entspricht, sind somit die Bedingungen (1), (2) und (3) aus Definition 3.14 erfüllt.

Daß die Bedingungen (4) und (5) ebenfalls gelten, ist klar, denn wir haben sie ja gerade gefordert, um die speziellen Eigenschaften der auf Linearisierungen von $\mathbf{VG}_p(A)$ basierenden Vererbungsstrategien auch im allgemeinen Fall zu erhalten.

3.3 Durchführung der Vererbung

In diesem Abschnitt definieren wir nun die syntaktische Transformation, die durch „Anwendung der Vererbungsstrategie“ aus einem abstrakten Programm $p = \langle \gamma, e \rangle$ ein abstraktes Programm ohne Vererbung erzeugt, indem die Instanzvariablenlisten der Framedefinitionen vervollständigt und die fehlenden Definitionen der ererbten Methoden hinzugefügt werden. Wir haben im vorigen Abschnitt ein Paar $\langle \mathit{instvars}, \mathit{methods} \rangle$ von Abbildungen

$$\begin{aligned} \mathit{instvars} &\in \mathit{InhGraph} \longrightarrow (\mathit{Framename} \xrightarrow{\mathit{fin}} \mathit{InstVar}^*) \\ \mathit{methods} &\in \mathit{InhGraph} \longrightarrow ((\mathit{Framename} \times \mathit{Message}) \xrightarrow{\mathit{fin}} \mathit{Abstraction}), \end{aligned}$$

als *Vererbungsstrategie* bezeichnet und definiert, wann eine Vererbungsstrategie *vernünftig* ist. Wir setzen nun das Vorhandensein einer vernünftigen Vererbungsstrategie voraus. Sie bestimmt zu den Frames des Programmes die Liste aller Instanzvariablen und alle Methoden. Der abstrakte Deklarationsteil γ des Programms p haben wir durch die Anwendung der Abbildung $\mathbf{IG}\,[\![\,\bullet\,]\!]$ aus dem konkreten Programm gewonnen. (vgl. Kapitel 2). Wir erinnern uns an den Bereich von γ :

$$\begin{aligned} \gamma \in \mathit{InhGraph} =_{\mathrm{Def}} \quad & (\mathit{Ident} \xrightarrow{\mathit{fin}} \mathit{Abstraction}) \times \\ & (\mathit{Framename} \xrightarrow{\mathit{fin}} \mathit{Framename}^*) \times \\ & (\mathit{Framename} \xrightarrow{\mathit{fin}} \mathit{InstVar}^*) \end{aligned}$$

Programme ohne Vererbung wollen wir nun als Paare $\langle g, e \rangle$ repräsentieren, wobei wir $g \in \mathit{CEnv}$ auch als Übersetzungszeitumgebung bezeichnen:

Definition 3.15

Ein Element $g \in \mathit{CEnv}$ in dem Bereich

$$\mathit{CEnv} =_{\mathrm{Def}} (\mathit{Ident} \xrightarrow{\mathit{fin}} \mathit{Abstraction}) \times (\mathit{Framename} \xrightarrow{\mathit{fin}} \mathit{InstVar}^*).$$

nennen wir *Übersetzungszeitumgebung*. □

Wir können g wiederum als abstrakten Deklarationsteil des Programmes $\langle g, e \rangle$ auffassen. Dies rechtfertigt die Bezeichnung *abstraktes Programm ohne Vererbung* für $\langle g, e \rangle$ (s.u.).

Definition 3.16

Sei $p = \langle \gamma, e \rangle$ ein abstraktes Programm. Sei ferner $\langle instvars, methods \rangle$ eine vernünftige Vererbungsstrategie. Dann definieren wir die *Anwendung der Vererbungsstrategie* auf γ als Abbildung

$$VS \ : \ InhGraph \longrightarrow CEnv$$

wie folgt:

$$\begin{array}{lll} VS\ \gamma & =_{\mathrm{Def}} & \langle\, \lambda I \,.\, I \in Var \ \rightarrow \ (\gamma\ I)\,, \\ & & \qquad (\, methods\ \gamma\ I\,)\,, \\ & & (\, instvars\ \gamma\,)\,\rangle \end{array}$$

□

Die zweite Komponente von $(\ VS\ \gamma\ )$ ist also gerade durch die Abbildung *instvars* bestimmt, die zu jedem Frame die Liste aller Instanzvariablen liefert. In der ersten Komponente bleiben die Bindungen an gewöhnliche Variablen, also die aus den Funktionsdefinitionen des originalen Programmes enstandenen Bindungen, unverändert, während die Abbildung *methods* die Bindungen für die Methodenfunktionsnamen $\langle A, m \rangle$ bestimmt.

Wir führen die Vererbung nun durch, indem wir die Übersetzungszeitumgebung g durch Anwendung der Vererbungsstrategie auf den abstrakten Deklarationsteil γ bestimmen. Der Hauptteil e des Programmes bleibt unverändert.

Definition 3.17

Als *abstrakte Programme ohne Vererbung* bezeichnen wir Paare $\langle g, e \rangle \in CEnv \times Expr$ mit Übersetzungszeitumgebungen

$$\begin{array}{lcll} g & \in & CEnv & =_{\mathrm{Def}} \quad (\, Ident \xrightarrow{fin} Abstraction\,) \ \times \\ & & & \qquad (\, Framename \xrightarrow{fin} InstVar^*\,) \end{array}$$

als abstraktem Deklarationsteil. □

Ein abstraktes Programm ohne Vererbung entsteht nun gerade aus $p = \langle \gamma, e \rangle$ durch Anwendung der Vererbungsstrategie auf den Deklarationsteil γ .

Definition 3.18

Sei $p = \langle \gamma, e \rangle$ ein abstraktes Programm, und sei $\langle instvars, methods \rangle$ eine vernünftige Vererbungsstrategie. Dann erhalten wir durch *Anwendung der Vererbungsstrategie* auf p das abstrakte Programm ohne Vererbung

$$\langle (VS\ \gamma), e \rangle \ \in CEnv \times Expr$$

□

Es ist klar, daß wir aus $g \in CEnv$ wiederum konkrete Deklarationen rekonstruieren können, indem wir aus den Bindungen der ersten Komponente von g die entsprechenden Methoden- und Funktionsdefinitionen erzeugen. Aus den Bindungen der zweiten Komponente können wir entsprechende Framedefinitionen rekonstruieren, die nun nur noch leere Superframelisten haben. Wir können uns das durch Anwendung der Vererbungsstrategie erzeugte Programm auf diese Weise veranschaulichen.

Beispiel 3.19

Aus dem Beispielprogramm des vorigen Abschnittes entsteht nun die folgende Übersetzungszeitumgebung, wenn wir die Vererbungsstrategie benutzen, die durch die zweite im Abschnitt 3.2 definierte Linearisierung entsteht:

$$
\begin{array}{lll}
\langle\ \{ & \langle A, \mathit{set\text{-}x}\,\rangle & \mapsto\ \lambda\, \mathit{self}\,.\,\lambda\, n\,.\,\mathit{self}.x := n\ , \\
& \langle A, \mathit{add}\,\rangle & \mapsto\ \lambda\, \mathit{self}\,.\,\lambda\,()\,.\,\mathit{self}.x + \mathit{self}.y\ , \\
& \langle B, \mathit{set\text{-}y}\,\rangle & \mapsto\ \lambda\, \mathit{self}\,.\,\lambda\, n\,.\,\mathit{self}.y := n\ , \\
& \langle B, \mathit{set\text{-}x}\,\rangle & \mapsto\ \lambda\, \mathit{self}\,.\,\lambda\, n\,.\,\mathit{self}.x := n\ , \\
& \langle B, \mathit{add}\,\rangle & \mapsto\ \lambda\, \mathit{self}\,.\,\lambda\,()\,.\,\mathit{self}.x + \mathit{self}.y\ , \\
& \langle C, \mathit{add}\,\rangle & \mapsto\ \lambda\, \mathit{self}\,.\,\lambda\,()\,.\,\mathit{self}.y + \mathit{self}.x\ , \\
& \langle C, \mathit{set\text{-}x}\,\rangle & \mapsto\ \lambda\, \mathit{self}\,.\,\lambda\, n\,.\,\mathit{self}.x := n\ , \\
& \langle D, \mathit{set\text{-}x}\,\rangle & \mapsto\ \lambda\, \mathit{self}\,.\,\lambda\, n\,.\,\mathit{self}.x := n\ , \\
& \langle D, \mathit{set\text{-}y}\,\rangle & \mapsto\ \lambda\, \mathit{self}\,.\,\lambda\, n\,.\,\mathit{self}.y := n\ , \\
& \langle D, \mathit{add}\,\rangle & \mapsto\ \lambda\, \mathit{self}\,.\,\lambda\,()\,.\,\mathit{self}.y + \mathit{self}.x \quad \}\ ,
\end{array}
$$

$$
\begin{array}{lll}
\{ & A & \mapsto\ (x)\ , \\
& B & \mapsto\ (x, y)\ , \\
& C & \mapsto\ (x)\ , \\
& D & \mapsto\ (x, y) \quad \}\ \ \rangle
\end{array}
$$

Diese entspricht dem folgenden konkreten Deklarationsteil:

$$
\begin{array}{l}
\mathit{deffrane}\ A\ (\,)\ (x)\,; \\
\mathit{defmethod}\ A\ \mathit{set\text{-}x}\,(\,n\,)\ =\ \mathit{self}.x\ :=\ n\,; \\
\mathit{defmethod}\ A\ \mathit{add}\,(\,)\ =\ \mathit{self}.x\ +\ \mathit{self}.y\,;
\end{array}
$$

$$
\begin{array}{l}
\mathit{deffrane}\ B\ (\,)\ (x,y)\,; \\
\mathit{defmethod}\ B\ \mathit{set\text{-}y}\,(\,n\,)\ =\ \mathit{self}.y\ :=\ n\,; \\
\mathit{defmethod}\ B\ \mathit{set\text{-}x}\,(\,n\,)\ =\ \mathit{self}.x\ :=\ n\,; \\
\mathit{defmethod}\ B\ \mathit{add}\,(\,)\ =\ \mathit{self}.x\ +\ \mathit{self}.y\,;
\end{array}
$$

$$
\begin{array}{l}
\mathit{deffrane}\ C\ (\,)\ (x)\,; \\
\mathit{defmethod}\ C\ \mathit{add}\,(\,)\ =\ \mathit{self}.y\ +\ \mathit{self}.x\,; \\
\mathit{defmethod}\ C\ \mathit{set\text{-}x}\,(\,n\,)\ =\ \mathit{self}.x\ :=\ n\,;
\end{array}
$$

$$
\begin{array}{l}
\mathit{deffrane}\ D\ (\,)\ (x,y)\,; \\
\mathit{defmethod}\ D\ \mathit{set\text{-}x}\,(\,n\,)\ =\ \mathit{self}.x\ :=\ n\,; \\
\mathit{defmethod}\ D\ \mathit{set\text{-}y}\,(\,n\,)\ =\ \mathit{self}.y\ :=\ n\,; \\
\mathit{defmethod}\ D\ \mathit{add}\,(\,)\ =\ \mathit{self}.y\ +\ \mathit{self}.x\,;
\end{array}
$$

Benutzen wir stattdessen die Vererbungsstrategie auf der Basis des ersten Linearisierungsverfahrens, so erbt D die Methode zu add aus A, erhält also die Methodendefinition

$$
\mathit{defmethod}\ D\ \mathit{add}\,(\,)\ =\ \mathit{self}.x\ +\ \mathit{self}.y\,;
$$

3.4 Instanziierbare Frames

In diesem Abschnitt definieren wir eine Einschränkung an *wohlgeformte* Programme (vgl. Abschnitt 3.5), die im Zusammenhang mit der vernünftigen Verwendung von Instanzvariablenreferenzen steht.

In Beispiel 3.4 haben wir ein Programm gesehen, in dem speziell der Frame C eine Methode add mit einer lesenden Referenz $self.y$ auf die in C lokal nicht vorhandene Instanzvariable y besitzt. In Beispiel 3.19 wird deutlich, daß dies für den Frame C auch nach Durchführung der Vererbung so bleibt, falls wir eine der dort genannten Vererbungsstrategien verwenden.

In D dagegen ist die Methode add durchaus vernünftig, da Instanzen von D eine Instanzvariable y besitzen.

Wir wollen die Fehlersituation verhindern, die in einem Programm eventuell dadurch auftritt, daß durch eine Nachricht eine Methode aufgerufen wird, die eine Referenz auf eine in dem Empfängerobjekt nicht vorhandene Instanzvariable durchführt. Ob dieser Fehler in einem Programm tatsächlich auftritt, können wir natürlich statisch nicht entscheiden, denn wir wissen nicht, ob eine Methode, die ihn durch eine Nachricht an ein spezielles Objekt potentiell hervorruft, auch tatsächlich mit einem entsprechenden Empfängerobjekt aufgerufen wird.

Allerdings können wir nach Durchführung der Vererbung eine hinreichende Bedingung angeben und für das entstandene Programm ohne Vererbung auch entscheiden, die diesen

Fehler verhindert. Wir kennen nunmehr alle Instanzvariablen und Methoden aller Frames und können verlangen, daß nur Instanzen von Frames gebildet werden, in deren Methoden höchstens Referenzen auf ebenfalls in dem Frame vorhandene Instanzvariablen vorkommen. Frames, die diese Eigenschaft besitzen, werden wir *instanziierbar* nennen. Wir werden, um auf das Beispiel zurückzukommen, sehen, daß C ein nicht instanziierbarer, D aber ein instanziierbarer Frame ist.

In Programmen ohne Vererbung kann auf sehr einfache Weise eine *Bindungsfunktion* zwischen den *angewandten Vorkommen* $self.y$ oder $self.y := e$ der Instanzvariablen in Methodenrümpfen und ihren *definierenden Vorkommen* in den Instanzvariablenlisten der Frames definiert werden[1]. Wir werden diese Bindungsfunktion hier nicht explizit definieren, da wir den Begriff *Vorkommen eines Ausdrucks* im Rahmen dieser Arbeit nicht formalisiert haben. Hätten wir dies getan, so wäre ein Frame gerade dann instanziierbar, wenn die Bindungsfunktion des Programmes, eingeschränkt auf die angewandten Instanzvariablenvorkommen in den Methoden des Frames, total definiert ist. Wir betrachten dabei natürlich das durch Anwendung der als vernünftig vorausgesetzten Vererbungsstrategie entstandene Programm ohne Vererbung.

Im folgenden setzen wir voraus, daß $vs = \langle instvars, methods \rangle$ eine vernünftige Vererbungsstrategie ist und daß $\langle g, e \rangle = \langle (VS\ \gamma), e \rangle$ durch Anwendung dieser Vererbungsstrategie auf $p = \langle \gamma, e \rangle$ entsteht. Dabei ist $VS \in InhGraph \longrightarrow CEnv$ die auf vs beruhende syntaktische Transformation gemäß Definition 3.16.

Definition 3.20

Sei $p = \langle \gamma, e \rangle$ ein abstraktes Programm. Sei $A \in Frames(p)$ und sei $\langle g, e \rangle$ das durch Anwendung von VS auf p entstandene abstrakte Programm $\langle (VS\ \gamma), e \rangle \in CEnv \times Expr$. Dann definieren wir mit

$$Methods_p(A) =_{\mathrm{Def}} \{ a \in Abstraction \mid \exists\, m \in Message : (g{\downarrow_1}\ \langle A, m \rangle) = a \}$$

die Menge *aller Methoden von* A in p. □

Die Menge $Methods_p(A)$ ist eine endliche Menge von Abstraktionen (oder Methoden) der Form $\lambda self.\lambda x.e$ bzw. $\lambda self.\lambda().e$. Völlig analog können wir zu einem abstrakten Programm $\langle g, e \rangle$ ohne Vererbung die Menge $Methods_{\langle g,e \rangle}(A)$ der zu A in g vorhandenen Methoden definieren. In einem beliebigen abstrakten Deklarationsteil g entsprechen allerdings die an $\langle A, m \rangle$ gebundenen Abstraktionen nicht unbedingt Methoden der o.a. Form. In den Rümpfen der Abstraktionen in $Methods_p(A)$ bzw. $Methods_{\langle g,e \rangle}(A)$ kommen eventuell Instanzvariablenreferenzen der Form $self.y$ oder $self.y := e$ vor, und wir verlangen von einem *instanziierbaren* Frame A nun, daß die Instanzvariablen y Instanzvariablen von A sind:

[1] Im allgemeinen betrachtet man in imperativen Programmen die *Bindungsrelation* zwischen angewandten und definierenden Identifikatorvorkommen und verlangt von statisch semantisch korrekten Programmen, daß diese Relation eine total definierte Funktion ist, daß also jedes angewandte Vorkommen an genau ein definierendes *gebunden* ist. Wir können von einer Bindungsfunktion sprechen, da wir uns auf Instanzvariablen beschränken und eine vernünftige Vererbungsstrategie nur Instanzvariablenlisten erzeugt, in denen keine der Instanzvariablen mehrfach vorkommt.

Definition 3.21

Sei $p = \langle \gamma, e \rangle$ ein abstraktes Programm. Sei $A \in \mathit{Frames}(p)$ und sei $\langle g, e \rangle$ das durch Anwendung von VS auf p entstandene abstrakte Programm $\langle (VS\ \gamma), e \rangle \in \mathit{CEnv} \times \mathit{Expr}$. Sei ferner $\mathit{Methods}_p(A)$ die Menge aller Methoden von A in p.
Dann heißt A *instanziierbar*, falls für jede Instanzvariable y, die im Rumpf einer Methode $a \in \mathit{Methods}_p(A)$ vorkommt, gilt, daß y auch Instanzvariable von A ist, genauer, daß y in der Instanzvariablenliste $(g{\downarrow_2}\ A)$ auftritt. □

Es ist klar, wann ein Frame in einem abstrakten Programm $\langle g, e \rangle$ instanziierbar ist. Natürlich hängt die Eigenschaft der Frames des Programmes p, instanziierbar oder nicht instanziierbar zu sein, von der konkret verwendeten Vererbungsstrategie VS ab, denn wir haben diese Eigenschaft anhand des durch Anwendung von VS entstandenen Programmes ohne Vererbung definiert.

Wir haben bereits in Beispiel 2.19 im Kapitel 2 behauptet, daß das dort angegebene Programm zwei instanziierbare und zwei nicht instanziierbare Frames enthält. Dies wollen wir uns nun klarmachen.

Beispiel 3.22

Betrachten wir dazu das in Beispiel 3.19 angegebene Programm ohne Vererbung, das wir aus unserem ursprünglichen Programm durch Anwendung der Vererbungsstrategie gemäß [Sym86] gewonnen haben. Durch

$$\begin{array}{l} \mathit{defframe}\ A\ (\,)\ (x)\,; \\ \mathit{defmethod}\ A\ \mathit{add}\,(\,) = \mathit{self}.x + \mathit{self}.y\,; \end{array}$$

ist klar, daß A ein nicht instanziierbarer Frame ist, denn die Methode add enthält eine Referenz auf die in A nicht vorhandene Instanzvariable y. Ebenso ist C nicht instanziierbar, denn C enthält ebenfalls keine Instanzvariable y, obwohl in der Methode add eine Referenz auf y vorhanden ist:

$$\begin{array}{l} \mathit{defframe}\ C\ (\,)\ (x)\,; \\ \mathit{defmethod}\ C\ \mathit{add}\,(\,) = \mathit{self}.y + \mathit{self}.x\,; \end{array}$$

Die beiden anderen Frames sind instanziierbar, denn es gibt überhaupt nur Instanzvariablenreferenzen auf x und y, und sowohl B als auch D enthalten beide Instanzvariablen:

$$\begin{array}{l} \mathit{defframe}\ B\ (\,)\ (x, y)\,; \\ \mathit{defmethod}\ B\ \mathit{set\text{-}y}\,(\,n\,) = \mathit{self}.y := n\,; \\ \mathit{defmethod}\ B\ \mathit{set\text{-}x}\,(\,n\,) = \mathit{self}.x := n\,; \\ \mathit{defmethod}\ B\ \mathit{add}\,(\,) = \mathit{self}.x + \mathit{self}.y\,; \end{array}$$

$$\begin{array}{l} \mathit{defframe}\ D\ (\,)\ (x, y)\,; \\ \mathit{defmethod}\ D\ \mathit{set\text{-}x}\,(\,n\,) = \mathit{self}.x := n\,; \\ \mathit{defmethod}\ D\ \mathit{set\text{-}y}\,(\,n\,) = \mathit{self}.y := n\,; \\ \mathit{defmethod}\ D\ \mathit{add}\,(\,) = \mathit{self}.y + \mathit{self}.x\,; \end{array}$$

Betrachten wir nun auch noch den Hauptprogrammausdruck, den wir in all den bisher durchgeführten Transformationen unverändert gelassen haben:

$$(\,\lambda\, di\,.\\ \qquad (\,di\,.\,set\text{-}x\,(\,17\,)\,;\;\; di\,.\,set\text{-}y\,(\,25\,)\,;\;\; di\,.\,add\,(\,)\,)\\ )\,(\,instance\text{-}of\;\; D\,)\,;$$

Im gesamten entstandenen Programm kommt nur die Instanziierung $(\,instance\text{-}of\;\; D\,)$ im Hauptprogramm vor. Es werden also nur Instanzen instanziierbarer Frames gebildet.

3.5 Wohlgeformte Programme

Zum Abschluß dieses Kapitels können wir nun endlich definieren, wann ein Programm *wohlgeformt* oder *statisch semantisch korrekt* ist. Wir tun dies zunächst für abstrakte Programme $\langle g, e\,\rangle$ ohne Vererbung, denn die wesentlichen Bedingungen lassen sich erst nach Durchführung der Vererbung definieren. Dies liegt insbesondere daran, daß wir allgemein zugelassen haben, daß eine Vererbungsstrategie die Rümpfe der in dem ursprünglichen Programm vorhandenen Methoden modifizieren darf, so daß die in dem Programm vorkommenden Ausdrücke erst nach Durchführung der Vererbung bekannt sind.

Natürlich nennen wir das ursprüngliche Programm mit Vererbung gerade dann *wohlgeformt*, wenn das ihm durch Anwendung der Vererbungsstrategie zugeordnete Programm ohne Vererbung wohlgeformt ist.

Im folgenden werden wir die einzelnen Bedingungen an wohlgeformte Programme angeben und diskutieren und sie schließlich in Definition 3.23 zusammenfassen. Sei dazu $\langle g, e_0\,\rangle \in CEnv \times Expr$ ein abstraktes Programm ohne Vererbung.

Zunächst verlangen wir, daß Methodennamen $\langle A, m\,\rangle$ auch tatsächlich an Methoden der Form $\lambda\, self\,.\,\lambda\, x\,.\,e$ bzw. $\lambda\, self\,.\,\lambda\,(\,)\,.\,e$ gebunden sind. Wir fordern also

- Ist $(g{\downarrow_1}\;\langle A, m\,\rangle)$ definiert, dann ist der Wert von $(g{\downarrow_1}\;\langle A, m\,\rangle)$ eine Methode der Form $\lambda\, self\,.\,\lambda\, x\,.\,e$ oder $\lambda\, self\,.\,\lambda\,(\,)\,.\,e$.

Instanzvariablenreferenzen der Form $self\,.\,y$ bzw. $self\,.\,y\;:=\;e$ haben nur in Rümpfen von Methoden eine vernünftige Bedeutung. Wir lassen sie deshalb nur in Methodenrümpfen zu. Zudem wollen wir verlangen, daß *self* an diesen Stellen auch wirklich das Empfängerobjekt der entsprechenden Nachricht bezeichnet, daß *self* also an der Stelle der Instanzvariablenreferenz $self\,.\,y$ bzw. $self\,.\,y\;:=\;e$ frei in dem Methodenrumpf vorkommt und syntaktisch verschieden von dem eventuellen Parameter x der Methode ist.

- Instanzvariablenreferenzen treten nur in Methodenrümpfen auf, und dort auch nur an Stellen, an denen *self* frei ist.

Die nächste Anforderung, die wir an wohlgeformte Programme stellen, betrifft die Instanziierungen $(instance\text{-}of\;\; A)$, die in dem Programm vorkommen. Wir haben im vorigen Abschnitt den Begriff eines instanziierbaren Frames definiert und verlangen nun, daß nur solche instanziiert werden:

- Kommt eine Ausdruck $(\mathit{instance\text{-}of}\ A)$ in g oder im Hauptprogramm e_0 vor, so ist A instanziierbarer Frame in $\langle g, e_0 \rangle$.

Wir lassen zu, daß das Programm freie gewöhnliche Variablen enthält. Es ist also nicht möglich, jeden gewöhnlichen Funktionsaufruf der Form $f(e_1)$ bzw. $f()$ mit $f \in \mathit{Var}$ auf korrekte Parameterübergabe zu prüfen. Wir können dies ebenfalls für Nachrichten der Form $e_1 . m(e_2)$ oder $e_1 . m()$ nicht tun, da wir die aufgerufene Methodenfunktion im allgemeinen nicht kennen. Wir wollen aber dennoch wenigstens verlangen, daß die „bekannten" gewöhnlichen Funktionen $f \in \mathit{Var}$ korrekt aufgerufen werden, und daß es überhaupt eine Methodendefinition zu m gibt, wenn eine Nachricht $e_1 . m$ in dem Programm vorkommt.

Diese beiden Anforderungen entsprechen den Punkten (4), (5) und (6) der nun folgenden Definition *wohlgeformter* Programme.

Definition 3.23

Sei $\langle g, e_0 \rangle \in \mathit{CEnv} \times \mathit{Expr}$ ein Programm ohne Vererbung. Dann heißt $\langle g, e_0 \rangle$ *wohlgeformt*, wenn die folgenden Bedingungen erfüllt sind:

(1) Ist $(g{\downarrow_1}\ \langle A, m \rangle)$ definiert, dann ist $(g{\downarrow_1}\ \langle A, m \rangle)$ eine Methode der Form $\lambda\, \mathit{self} . \lambda\, x . e$ oder $\lambda\, \mathit{self} . \lambda\, () . e$ für einen Ausdruck $e \in \mathit{Expr}$ und $x \in \mathit{Var}$.

(2) Instanzvariablenreferenzen $\mathit{self}.y$ bzw. $\mathit{self}.y := e$ treten nur in Methodenrümpfen $(g{\downarrow_1}\ \langle A, m \rangle) = \lambda\, \mathit{self} . \lambda\, x . e_1$ bzw. $(g{\downarrow_1}\ \langle A, m \rangle) = \lambda\, \mathit{self} . \lambda\, () . e_1$ auf, und zwar derart, daß self frei in $\lambda\, x . e_1$ bzw. in $\lambda\, () . e_1$ ist.

(3) Kommt eine Ausdruck $\mathit{instance\text{-}of}\ A$ in g oder im Hauptprogramm e_0 vor, so ist A instanziierbarer Frame in $\langle g, e_0 \rangle$.

(4) Kommt ein (*gewöhnlicher*) Funktionsaufruf $f(e)$ mit $f \in \mathit{Var}$ in g oder e_0 vor, und ist $(g{\downarrow_1}\ f)$ definiert, so gilt $(g{\downarrow_1}\ f) = \lambda\, x . e_1$ für einen Ausdruck $e_1 \in \mathit{Expr}$ und $x \in \mathit{Var}$.

(5) Kommt ein (*gewöhnlicher*) Funktionsaufruf $f()$ mit $f \in \mathit{Var}$ in g oder e_0 vor, und ist $(g{\downarrow_1}\ f)$ definiert, so gilt $(g{\downarrow_1}\ f) = \lambda\, () . e_1$ für einen Ausdruck $e_1 \in \mathit{Expr}$.

(6) Kommt eine Nachricht $e_1 . m$ in g oder e_0 vor, so existiert ein $A \in \mathit{Framename}$ derart, daß $(g{\downarrow_1}\ \langle A, m \rangle)$ definiert ist.

□

Wie bereits angekündigt, definieren wir nun, daß ein Programm $p = \langle \gamma, e \rangle$ wohlgeformt ist, wenn das ihm durch Anwendung der Vererbungsstrategie zugeordnete Programm ohne Vererbung wohlgeformt ist. Die Forderung (1) in Definition 3.23 ist für Programme, die durch Anwendung einer vernünftigen Vererbungsstrategie aus einem Programm $\langle \gamma, e \rangle$ entstehen, unnötig, denn wir haben in Definition 3.14 gefordert, daß die Abbildung *methods* einer vernünftigen Vererbungsstrategie nur Methoden der geforderten Form erzeugt. Wir verlangen zusätzlich, daß der Vererbungsgraph $\mathbf{VG}(p)$ azyklisch ist.

Definition 3.24

Sei $p = \langle \gamma, e \rangle$ ein abstraktes Programm. Sei $\langle g, e \rangle$ das durch Anwendung von VS auf p entstandene abstrakte Programm $\langle (VS\ \gamma), e \rangle \in CEnv \times Expr$, wobei VS gemäß Definition 3.16 die zu einer vernünftigen Vererbungsstrategie $\langle instvars, methods \rangle$ gehörende syntaktische Transformation ist. Sei $\mathbf{VG}(p)$ der Vererbungsgraph von p .
Dann heißt p *wohlgeformt* oder *statisch semantisch korrekt*, falls $\mathbf{VG}(p)$ azyklisch und $\langle g, e \rangle$ wohlgeformt ist. □

Damit haben wir die Definition der statischen Semantik von Programmen abgeschlossen. Die beiden folgenden Kapitel widmen sich der Definition der dynamischen Semantik, der Definition eines Übersetzers und dem Beweis der Korrektheit des Übersetzers. Dabei betrachten wir im folgenden nur noch abstrakte Programme ohne Vererbung, von denen wir stillschweigend voraussetzen, daß sie wohlgeformt sind. Sowohl die dynamische Semantik von Programmen als auch der Übersetzer hängen von der in diesem Kapitel definierten expliziten Durchführung der Vererbung durch die Anwendung einer vernünftigen Vererbungsstrategie ab. Insofern geht die konkret benutzte Vererbungsstrategie als „Parameter“ in den im Rahmen dieser Arbeit vorgeschlagenen Ansatz zur Beschreibung und korrekten Implementation objektorientierter Sprachen mit multipler Vererbung ein.

4

Dynamische Semantik und Übersetzung

Das nun folgende Kapitel widmet sich der dynamischen Semantik von Ausdrücken unserer objektorientierten Sprache und der Definition eines Übersetzers, der insbesondere Nachrichten in „generische Funktionsaufrufe“ übersetzt.

Im folgenden Abschnitt beschäftigen wir uns zunächst mit den mathematischen Grundlagen der hier verwendeten denotationellen Semantik und erläutern dabei unsere Notation. Der dann folgende zweite Abschnitt definiert eine denotationelle Continuation-Semantik. Zunächst geben wir die semantischen Bereiche an. Es handelt sich um reflexive Bereiche, da wir alle Funktionen als Daten erster Ordnung zulassen. Anschließend definieren wir das semantische Funktional $\mathcal{E}[\![\bullet]\!]$ induktiv über den syntaktischen Aufbau von Ausdrücken durch Angabe der entsprechenden semantischen Gleichungen.

Der dritte Abschnitt erweitert unsere Sprache um einen Ausdruck, der die endliche generische Methodenfunktionsauswahl gestattet. In Anlehnung an Lisp nennen wir ihn *typecase*, obwohl die Semantik spezieller definiert wird: *typecase* e *of* $A_1 : e_1, \ldots, A_n : e_n$ selektiert nicht nur anhand des Typs von e, sondern wir gehen davon aus, daß die Bedeutung des selektierten e_i eine Methode ist, die wir gleichzeitig auf die Bedeutung von e anwenden. Ist ein *typecase*-Ausdruck wohldefiniert, so liefert er eine Methodenfunktion.

Der letzte Abschnitt dieses Kapitels schließlich definiert einen Übersetzer induktiv über den Aufbau von Ausdrücken, der insbesondere Nachrichten in entsprechende endliche Methodenselektionen überführt. Dazu fassen wir das abstrakte Programm, das wir im vorigen Kapitel durch die Anwendung der Vererbungsstrategie aus dem Vererbungsgraphen des Originalprogrammes gewonnen haben, tatsächlich als Übersetzungszeitumgebung auf. Der Übersetzer ist also nicht inkrementell, sondern benötigt die Information über alle zu einem Frame gehörenden Methoden.

Sprachen wie Lisp bieten Datentypen, die sich zur *Repräsentation* von Objekten eignen und die Programmierung der lesenden und schreibenden Zugriffe erlauben. Durch die Repräsentation der Framenamen als Lisp-Daten läßt sich auch die generische Methodenauswahl mit geeigneten Lisp-Funktionen *programmieren*. Tatsächlich erzeugt der bereits erwähnte BABYLON-Übersetzer [Goe89a] konkret derartige Repräsentationen und übersetzt Instanzierungen und Zugriffe auf Instanzvariablen in entsprechende Funktionsaufrufe des Laufzeit-

systems.

Wir wollen in dieser Arbeit bewußt darauf verzichten, den applikativen Kern unserer Sprache um einen derartigen *generischen Datentyp* zu erweitern, mit dessen Hilfe die Repräsentation von Objekten und die *Programmierung* der generischen Methodenauswahl möglich würde. Wir halten das dynamische Bilden von Instanzen und eben die generische Methodenauswahl für elementare Bestandteile objektorientierter Sprachen, die diese von applikativen oder imperativen Sprachen unterscheiden.

4.1 Grundlagen

Um die dynamische Semantik der entstandenen abstrakten Programme ohne Vererbung zu beschreiben, benutzen wir eine *denotationelle Continuation*-Semantik. Die denotationelle Beschreibung von Programmiersprachen geht auf D. Scott und Ch. Strachey zurück. Den syntaktischen Bestandteilen von Programmen werden Elemente, sog. *Denotationen*, in geeigneten mathematischen Bereichen zugeordnet. Meist geschieht diese Zuordnung strukturell, d.h. induktiv über den syntaktischen Aufbau der Programme. Inzwischen hat sich herausgestellt, daß die *Domain-Theorie* von Scott [Sco81] genügend reichhaltig ist, um auch imperativen Sprachkonstrukten und rekursiv definierten Datentypen denotationell eine Bedeutung zuordnen zu können.

Sie basiert auf vollständigen partiellen Ordnungen (c.p.o's) bzw. vollständigen Verbänden zusammen mit ihren stetigen Funktionen. Die Lösung von rekursiven Gleichungen wird durch kleinste Fixpunkte stetiger Funktionen eindeutig möglich, eine Konsequenz aus dem Fixpunkttheorem von Tarski [Tar55]. Wir werden in der vorliegenden Arbeit spezielle vollständige Verbände verwenden, um die für unsere objektorientierte Sprache nötigen semantischen Bereiche zu definieren, die sog. *stetigen* vollständigen Verbände mit *abzählbarer Basis* [Sto77], die in der Literatur häufig auch *ω-algebraische vollständige Verbände* genannt werden. Unter ihnen gibt es einen ausgezeichneten universellen Bereich, in den sich jeder andere Bereich als Unterbereich einbetten läßt [Sco76], die Potenzmenge $P\omega$ der natürlichen Zahlen. Spezielle stetige Funktionen, sog. *Retrakte*, haben stetige vollständige Verbände als Bilder und bilden selbst einen vollständigen Verband, in dem die üblichen Bereichskonstruktoren $\longrightarrow$ (stetige Funktionen), $\times$ (Kreuzprodukt), $+$ (separierte Summe), $[\,]_\perp$ ($\perp$-Erweiterung) und auch $[\,]^*$ (endliche Listen) stetige Funktionen sind [Sco76]. Damit lassen sich *reflexive Bereiche* (*reflexive domains*) durch rekursive Gleichungssysteme definieren und die Fixpunkttheorie sichert die Existenz einer eindeutig bestimmten kleinsten Lösung.

Bei der Beschreibung der imperativen Sprachbestandteile wie der Zuweisung oder der Hintereinanderausführung von Anweisungen hat sich die Technik der *Continuations* durchgesetzt, die ursprünglich zur Beschreibung der denotationellen Semantik einer Sprache mit Sprüngen in [SW74] vorgeschlagen wurde. Einer Anweisung wird nicht die zugehörige Zustandstransformation an sich zugeordnet – ein Element des Bereiches $Store \longrightarrow Store$, sondern die Wirkung der Anweisung auf die ihr folgende Zustandstransformation, die *Continuation*. Dies ist ein Element des Bereiches $(Store \longrightarrow Store) \longrightarrow (Store \longrightarrow Store)$. Im Falle eines Sprunges oder einfach in einer Fehlersituation kann nun die ursprüngliche Continuation verworfen und durch eine andere ersetzt werden.

In Lisp-artigen Sprachen und auch in der hier definierten Sprache wird zwischen Anwei-

sungen und Ausdrücken nicht unterschieden. Auch Anweisungen haben Werte, und Ausdrücke können Zustandstransformationen (Seiteneffekte) bewirken. Im allgemeinen hängt die einem Ausdruck folgende Zustandstransformation auch von dem Wert des Ausdruckes ab, die Bedeutung des Ausdrucks also von einer *Ausdrucks-Continuation* der Form $(Value \times Store) \longrightarrow Answer$. Dabei bezeichnet *Answer* den Bereich der Programmergebnisse, in unserem Fall den Bereich $Value \times Store$. Es ist üblich, diesen Bereich *currifiziert* aufzuschreiben, also als $\kappa \in Value \longrightarrow Store \longrightarrow Answer$. Ausdrucks-Continuations beschreiben „die Semantik des Restes des Programmes in Abhängigkeit von dem Wert des Ausdrucks“. Der Einfachheit halber werden wir Ausdrucks-Continuations im folgenden auch als Continuations bezeichnen. Um einen Eindruck zu geben, sei an dieser Stelle bereits der Typ des semantischen Funktionals $\mathcal{E}[\![\bullet]\!]$ angegeben, auch wenn die genaue Struktur von Umgebungen noch nicht bekannt ist: Für einen Ausdruck $e \in Expr$ ist

$$\mathcal{E}[\![e]\!] : Env \longrightarrow ECont \longrightarrow Cont\,.$$

Um uns ein wenig mit der Notation vertraut zu machen, wollen wir an dieser Stelle zwei der in den folgenden Abschnitten definierten semantischen Gleichungen diskutieren. Überraschend einfach ist die Definition der Hintereinanderausführung $e_1\,;\,e_2$: Wir wenden die Bedeutung von e_1 auf die Continuation an, die wir erhalten, indem wir die Bedeutung von e_2 auf die ursprüngliche Continuation anwenden. Dabei wird der Wert von e_1 ignoriert.

$$\mathcal{E}[\![e_1\,;\,e_2]\!]\,\rho\kappa = \mathcal{E}[\![e_1]\!]\,\rho\,\{\lambda\nu\,.\,\mathcal{E}[\![e_2]\!]\,\rho\kappa\}$$

Continuations schreiben wir in $\{\ldots\}$, um die Lesbarkeit der semantischen Gleichungen zu verbessern. Mit κ bzw. ρ bezeichnen wir üblicherweise Continuations bzw. Umgebungen.

Fragen wir uns nun, um ein zweites Beispiel zu geben, was die Bedeutung einer λ-Abstraktion $\lambda x\,.\,e$ ist. An die Continuation wird ein „funktionaler Wert“ übergeben; dies ist eine Funktion, die an den jeweiligen Aufrufstellen mit einem Argument und einer dort aktuell gültigen Continuation versorgt wird. Die Umgebung, in der dann schließlich die Bedeutung von e bestimmt wird, wird aber an der Stelle der Definition „eingefroren“, um statische Identifikatorbindung (engl. *static scoping*) zu gewährleisten. Der Zusatz $(\ldots\ in\ Value)$ leistet die nötige Inklusion in den Bereich *Value*, in dem die funktionalen Werte nur einen Summanden darstellen. Die Modifikation der Umgebung ρ an der Stelle x um das Argument ν' notieren wir $\rho\,[x \leftarrow \nu']$.

$$\mathcal{E}[\![\lambda x\,.\,e]\!]\,\rho\kappa = \kappa\,(\lambda\nu'.\,\lambda\kappa'.\,\lambda\sigma'.\,\mathcal{E}[\![e]\!]\ \rho\,[x \leftarrow \nu']\ \kappa'\,\sigma'\ \ in\ Value)$$

Der funktionale Wert, der einer λ-Abstraktion entspricht, ist also Element des Bereiches $Value \longrightarrow ECont \longrightarrow Cont$, und er hängt von der Definitionszeitumgebung $\rho \in Env$ ab. Eine λ-Abstraktion bedeutet die Anwendung der Continuation κ auf diesen Wert. Wir haben die Continuation $\kappa = \mathcal{E}[\![e]\!]\ \rho\,[x \leftarrow \nu']\ \kappa'$ in der Form $\lambda\sigma'\,.\,\kappa\,\sigma'$ geschrieben. Mit $\sigma \in Store$ bezeichnen wir üblicherweise Zustände.

Denotationelle Continuation-Semantiken mit reflexiven semantischen Bereichen haben sich bei der Beschreibung höherer imperativer Sprachen wie beispielsweise Scheme [Abe91] quasi als Standard durchgesetzt. Sie werden eingehend in [Sto77] diskutiert. Wir wollen im folgenden die mathematischen Grundlagen bereitstellen, um eine solche Semantik definieren zu können.

4.1.1 Domains

Unter einer *partiell geordneten* Menge verstehen wir eine Menge M mit einer reflexiven, antisymmetrischen und transitiven Ordnungsrelation $\sqsubseteq_M \subseteq M \times M$. Sind $x, y \in M$ und ist $A \subseteq M$, so schreiben wir $x \sqsubseteq_M y$ für $\langle x, y \rangle \in \sqsubseteq_M$ und $x \sqsubseteq_M A$, falls für alle $a \in A$ $x \sqsubseteq_M a$ gilt. Gilt für alle $a \in A$ $a \sqsubseteq_M x$, so schreiben wir auch $A \sqsubseteq_M x$. In diesem Fall heißt x *obere Schranke* von A . Besitzt die Menge der oberen Schranken von A ein kleinstes Element $\sqcup A$, so nennen wir es *kleinste obere Schranke* von A . Falls $\sqcup A$ existiert, so ist es eindeutig bestimmt. Völlig analog definieren wir die *größte untere Schranke* $\sqcap A$ von A.

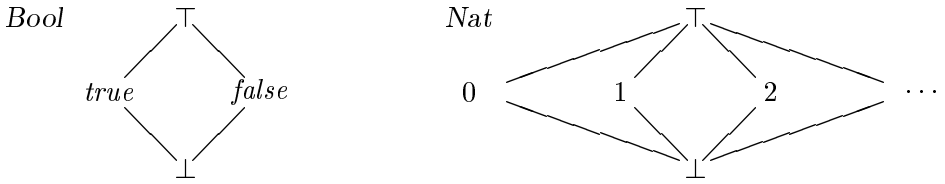
Definition 4.1

Eine partiell geordnete Menge D heißt *Verband*, falls für jede nichtleere, endliche Teilmenge $X \subseteq D$ die kleinste obere Schranke $\sqcup X$ und die größte untere Schranke $\sqcap X$ existieren. D heißt *vollständig*, falls $\sqcup X$ und $\sqcap X$ für jede Teilmenge $X \subseteq D$ existieren. □

Jede Menge M läßt sich zu einem *flachen* vollständigen Verband erweitern, indem ein kleinstes Element $\bot$ und ein größtes Element $\top$ hinzugefügt werden. Die Elemente von M bleiben unvergleichbar.

Definition 4.2

Die Bereiche *Bool* und *Nat* definieren wir als flache vollständige Verbände, ausgehend von den Mengen $\{true, false\}$ bzw. $\mathbb{N}_0$.



□

Die Potenzmenge 2^M einer Menge M ist ein Beispiel für einen Bereich, der mit seiner natürlichen Ordnung $\subseteq$ einen vollständigen Verband bildet. Ist X eine Menge von Teilmengen von M , also $X \subseteq 2^M$, so ist $\cup X$ die geforderte kleinste obere Schranke von X . Die Existenz der größten unteren Schranke folgt in vollständigen Verbänden D allgemein aus der Existenz der kleinsten oberen Schranke. Man mache sich dazu klar, daß $\sqcap X = \sqcup \{ d \in D \mid d \sqsubseteq_D X \}$ ist. In 2^M ist $\sqcap X$ gerade $\cap X$. Die leere Menge $\emptyset$ und M selbst sind kleinstes bzw. größtes Element in 2^M . Ein sehr wichtiger Bereich ist die Potenzmenge $P\omega =_{\text{Def}} 2^\omega$ der natürlichen Zahlen $\omega = \mathbb{N}_0$.

Die partielle Ordnung in einem vollständigen Verband D repräsentiert den „Grad der Definiertheit“ der Elemente in D . $\bot_D$ und $\top_D$ nennen wir auch *unechte* Elemente.

Definition 4.3

Sei D ein vollständiger Verband. Eine Teilmenge $E \subseteq D$ heißt *Basis* von D , falls

(a) jedes Element $d \in D$ kleinste obere Schranke der Menge der Basiselemente $e \sqsubseteq d$ ist, also für alle $d \in D$ gilt: $d = \sqcup \{ e \in E \mid e \sqsubseteq d \}$ und

(b) für jede endliche Teilmenge $E' \subseteq E$ die kleinste obere Schranke von E' ein Basiselement ist, d.h. $\sqcup E' \in E$ gilt.

□

Basiselemente haben „endlichen Charakter". Sie sind diejenigen Elemente, die wir im Speicher eines Rechners tatsächlich repräsentieren wollen. Wir verlangen deshalb, daß ein semantischer Bereich eine abzählbare Basis hat. Alle anderen Elemente von D können wir höchstens *approximieren*, d.h. durch eine sukzessive Berechnung der Basiselemente ein jeweils genügend definiertes Basiselement finden. Da nicht jede abzählbare Menge auch rekursiv aufzählbar ist, gelingt dies nicht in jedem Fall. Es gibt *nicht berechenbare* Elemente.

Teilmengen $S \subseteq D$, deren endliche Teilmengen $E \subseteq S$ eine kleinste obere Schranke in S haben, nennen wir *gerichtet*. Eine Basis von D ist also eine gerichtete Teilmenge von D derart, daß jedes Element von D kleinste obere Schranke einer Menge von Basiselementen von D ist.

Definition 4.4

Sei D ein vollständiger Verband. Seien $x, y \in D$. x heißt *essentiell kleiner* als y (in Zeichen: $x \ll y$), falls für jede gerichtete Teilmenge $S \subseteq D$ aus $y \sqsubseteq \sqcup S$ bereits $x \sqsubseteq z$ für eine $z \in S$ folgt. Einen vollständigen Verband, in dem sich jedes Element als kleinste obere Schranke der Menge der essentiell kleineren Elemente darstellen läßt, nennen wir *stetig*. D heißt also *stetig*, falls für alle $d \in D$ $d = \sqcup \{ x \in D \mid x \ll d \}$ gilt. □

Stetige vollständige Verbände mit abzählbarer Basis bezeichnen wir im folgenden als *semantische Bereiche* oder *Domains*.

4.1.2 Stetige Funktionen

Seien C und D vollständige Verbände. Diejenigen Funktionen von C nach D , die mit den partiellen Ordnungen $\sqsubseteq_C$ und $\sqsubseteq_D$ verträglich sind, nennen wir *monoton*. Eine Funktion $f \in C \longrightarrow D$ heißt also *monoton*, falls aus $x \sqsubseteq_C y$ folgt, daß $f(x) \sqsubseteq_D f(y)$ gilt. Ist f monoton, so ist das Bild $f(S) =_{\text{Def}} \{ f(s) \mid s \in S \}$ einer gerichteten Teilmenge $S \subseteq C$ eine gerichtete Teilmenge von D . Hat f überdies die Eigenschaft, daß für jede gerichtete Teilmenge $S \subseteq C$ die kleinste obere Schranke von S auf die kleinste obere Schranke der Menge der Bilder von S unter f abgebildet wird, so nennen wir f *stetig*.

Definition 4.5

Sind C, D vollständige Verbände und ist $f \in C \longrightarrow D$, so heißt f *stetig*, falls f monoton ist und für alle gerichteten Teilmengen $S \subseteq C$ gilt: $f(\sqcup S) = \sqcup f(S)$. □

Die Identität ist eine stetige Funktion, alle konstanten Funktionen sind stetig. Mit stetigen Funktionen $f \in C \longrightarrow D$ und $g \in B \longrightarrow C$ ist auch die Komposition $f \circ g \in B \longrightarrow D$ stetig. Generell ist jede in LAMBDA[1] definierbare Funktion von $P\omega$ nach $P\omega$ stetig (vgl. [Sco76] und [Sto77]). Da sich jeder semantische Bereich, also jeder stetige vollständige Verband mit abzählbarer Basis, in $P\omega$ einbetten läßt, genügt es, sich auf solche Funktionen zu beschränken. Zu den in LAMBDA definierbaren Funktionen gehört beispielsweise Mc'Carthy's Conditional $x \longrightarrow y, z$. λ-Abstraktion und Funktionsapplikation sind ebenfalls LAMBDA-definierbar.

Theorem 4.6 (Fixpunkttheorem von Tarski) *Ist D ein vollständiger Verband und ist f eine stetige Funktion von D nach D , so besitzt f einen kleinsten Fixpunkt fix f und*

$$fix\ f = \bigsqcup_{i=0}^{\infty} \{f^i(\bot)\}$$

Mehr noch, falls D ein semantischer Bereich ist, ist $fix \in (D \longrightarrow D) \longrightarrow D$ selbst stetig und LAMBDA-definierbar. Man betrachte dazu den wohlbekannten **Y**-Kombinator

$$\mathbf{Y} =_{\mathrm{Def}} \lambda f\,.\,(\,\lambda x\,.\,(f\ (x\ x))\,)\,(\,\lambda x\,.\,(f\ (x\ x))\,)\,.$$

Es ist also gerechtfertigt, die λ-Notation zur Definition von semantischen Gleichungen zu benutzen, da wir den Bereich der stetigen Funktionen nicht verlassen.

4.1.3 Reflexive Bereiche

Eine stetige Funktion heißt *Retraktion*, falls $f \circ f = f$ gilt. Retraktionen sind stetige Funktionen, die semantische Bereiche definieren. Ihr Bild ist gleich der Menge ihrer Fixpunkte, unter denen *fix* f das kleinste Element ist. Das folgende Theorem zeigt, daß die Bilder oder Fixpunktmengen von Retraktionen, die wir im folgenden *Retrakte* nennen wollen, stetige vollständige Verbände sind. Beweise finden sich beispielsweise in [Sco76] oder in [Sto77].

Theorem 4.7

Die Menge der Fixpunkte einer stetigen Funktion f bildet einen vollständigen Verband. Ist f eine Retraktion, so ist dieser Verband stetig.

Es ist damit klar, daß die Menge der Retraktionen selbst einen vollständigen Verband bildet, da jede Retraktion Fixpunkt der stetigen Funktion $\lambda f\,.\,f \circ f$ ist. Dieser Verband ist im allgemeinen nicht stetig. [Sco76] zeigt, daß der Verband der sog. Abschluß-Operationen (engl. *closure operation*) einen stetigen Verband bildet. Abschluß-Operationen sind Retraktionen, die größer oder gleich der Identität sind. [Sto77] zeigt, daß sich *Nat* und *Bool* in $P\omega$ als Retrakte definieren lassen. Ebenso lassen sich mit stetigen Operationen $\longrightarrow$, $\times$ und $+$ aus Retraktionen für A und B die Retraktionen für $A \longrightarrow B, A \times B$ und $A + B$ bilden. $A \longrightarrow B$ bezeichnet den Bereich der stetigen Funktionen von A nach B und ist, um ein Beispiel zu nennen, gleich dem Bild der stetigen Funktion $\lambda f\,.\,B \circ f \circ A$. $A \times B$ ist das Kreuzprodukt, $A + B$ die separierte Summe von A und B . Ebenso lassen sich der Bereich A^* der endlichen Listen von Elementen aus A und der Bereich $A_\bot =_{\mathrm{Def}} \{\bot\} \cup \{\,\langle 0, a\,\rangle \mid a \in A\}$ bilden.

[1] LAMBDA [Sco76] [Sto77] ist eine Notation (Sprache) zur Definition stetiger berechenbarer Funktionen von $P\omega$ nach $P\omega$. LAMBDA-Ausdrücke dienen zur Definition der Semantik des λ-Kalküls in dessen $P\omega$-Modell.

Semantische Bereiche, die durch rekursive Gleichungen mit den genannten Operationen definiert sind, existieren also als eindeutig bestimmte kleinste Fixpunkte stetiger Funktionen, die Tupel von Retraktionen auf Tupel von Retraktionen abbilden.

4.2 Semantik von Objekten

Objekte sind Daten. Sie können als solche Inhalt von Variablen, auch Instanzvariablen, und Ergebnis von Ausdrücken sein. Sie assoziieren Instanzvariablennamen (Slots) mit Speicherplätzen. Da Objekte ihren Framenamen (als Typ) beinhalten müssen, definieren wir Objekte als Paare, bestehend aus ihrem Framenamen und einem *Record* bzw. einer *Struktur* im Sinne von [Sto77]:

Definition 4.8

Objekte sind Paare, bestehend aus ihrem Framenamen und einer Zuordnung von Speicherplätzen zu ihren Instanzvariablen.

$$o \in \mathit{Objekt} =_{\mathrm{Def}} \mathit{Framename} \times (\mathit{InstVar} \longrightarrow \mathit{Location})$$

□

Dabei sind die Bereiche *Framename*, *InstVar*, *Location* wie auch die Bereiche *Message* und *Var* paarweise disjunkte flache Bereiche. Man beachte, daß die Objekte selbst keine Methoden beinhalten. Die Methoden haben wir vielmehr den Frames zugeordnet. Die Methode zu einem Methodennamen $m \in \mathit{Message}$ und einem Objekt $o \in \mathit{Object}$ erhalten wir als Bedeutung von $\langle A, m \rangle$ in der Umgebung. Dabei ist A der Framename, also die erste Komponente von o. Doch dazu später. Definieren wir zunächst Umgebungen:

Definition 4.9

Umgebungen sind Paare von Abbildungen, die Identifikatoren Werte und Framenamen Listen von Instanzvariablen zuordnen:

$$\rho \in \mathit{Env} =_{\mathrm{Def}} (\mathit{Ident} \longrightarrow \mathit{Value}) \times (\mathit{Framename} \longrightarrow \mathit{InstVar}^*)$$

□

Wir erinnern uns daran, daß *Ident* den Bereich der Variablenidentifikatoren $x \in \mathit{Var}$ und die verallgemeinerten Methodenidentifikatoren $\langle A, m \rangle \in \mathit{Framename} \times \mathit{Message}$ umfaßt. Den Bereich *Value*, der die elementaren Datenbereiche *Bool* und *Nat* sowie den Bereich der Funktionen und den Bereich der Objekte umfaßt, definieren wir später.

Speicherplätze bekommen erst im Zusammenhang mit einem *Zustand* $\sigma \in \mathit{Store}$ eine Bedeutung:

Definition 4.10

Den semantischen Bereich *Store* definieren wir als Paar von Abbildungen, die Speicherplätzen Werte und eine Markierung zuordnen, die besagt, ob der Speicherplatz benutzt ist oder nicht:

$$\sigma \in Store =_{\text{Def}} (Location \longrightarrow Value) \times (Location \longrightarrow Bool)$$

Gleichzeitig setzen wir die Existenz einer Funktion

$$new \in Store \longrightarrow Location \quad \text{voraus} \quad \text{mit} \quad (\sigma\downarrow_2 \ (new\ \sigma)) = false\,.$$

Falls kein neuer Speicherplatz $\alpha \in Location$ mit $(\sigma\downarrow_2\ \alpha) = false$ existiert, verlangen wir $(new\ \sigma) = \bot_{\text{Location}}$. Wir definieren die Hilfsfunktion $extend \in Location \longrightarrow Store \longrightarrow Store$ durch

$$extend\,\alpha\,\sigma =_{\text{Def}} \langle \sigma\downarrow_1\,,\ \sigma\downarrow_2\ [\alpha \leftarrow true]\ \rangle.$$

□

Gemäß der eingangs in diesem Kapitel gemachten Bemerkungen über Continuations definieren wir nun den semantischen Bereich *ECont* der Ausdrucks-Continuations und den Bereich *Cont* der Continuations durch

$$\begin{array}{rcll} \kappa \in ECont & = & Value \longrightarrow Cont \\ Cont & = & Store \longrightarrow Answer \\ Answer & = & Value \times Store \end{array}$$

Wir sind damit soweit, daß wir die Semantik von lesenden und schreibenden Zugriffen auf Instanzvariablen von Objekten definieren können. Ein lesender Zugriff $self.y$ liefert einen Wert an die Continuation, nämlich den Inhalt des Speicherplatzes, der in dem Objekt, das durch *self* bezeichnet wird, an y gebunden ist. Der Zustand bleibt unverändert. Ein schreibender Zugriff $self.y := e$ liefert den Wert von e und modifiziert den Zustand an eben diesem Speicherplatz, indem der Wert von e „dort hineingeschrieben wird“. Dies geschieht dadurch, daß wir die ursprüngliche Continuation auf den modifizierten Zustand anwenden, also sozusagen „den Rest des Programmes“ in dem modifizierten Zustand berechnen.

Definition 4.11

Die *Semantik von Ausdrücken* definieren wir durch das semantische Funktional

$$\mathcal{E}\,[\![\,e\,]\!] : Env \longrightarrow ECont \longrightarrow Cont\,.$$

□

Definition 4.12

$$\mathcal{E}\,[\![\,self.y\,]\!]\,\rho\kappa\ =$$
$$\quad \mathcal{E}\,[\![\,self\,]\!]\ \rho\,\{\lambda\nu_0\,.\,\nu_0 \in Object \to \lambda\sigma\,.\,\kappa\,(\sigma\ (\nu_0\downarrow_2\ y))\,\sigma\,,\ wrong\}$$

$$\mathcal{E}\,[\![\,self.y\ :=\ e\,]\!]\,\rho\kappa\ =$$
$$\quad \mathcal{E}\,[\![\,self\,]\!]\ \rho\,\{\lambda\nu_0\,.\,\nu_0 \in Object \to \mathcal{E}\,[\![\,e\,]\!]\ \rho\,\{\lambda\nu\,.\,\lambda\sigma\,.\,\kappa\nu\ \sigma\,[(\nu_0\downarrow_2\ y) \leftarrow \nu]\,\}\,,\ wrong\}$$

□

Ist die Bedeutung von *self* kein Objekt, so liefern unsere semantischen Gleichungen die total undefinierte Continuation $wrong =_{\text{Def}} \lambda\sigma\,.\,\bot_{\text{Answer}}$, d.h. sie brechen das Programm mit einem Fehler ab.

Die Modifikation des Zustandes $\sigma \in Store$, die wir in der semantischen Gleichung für $self.y := e$ der Einfachheit halber $\sigma\,[(\nu_0{\downarrow_2}\ y) \leftarrow \nu]$ geschrieben haben, ist tatsächlich eine Modifikation der ersten Komponente von σ. Wir werden auch im folgenden der Lesbarkeit halber häufig $\sigma\,[\alpha \leftarrow \nu]$ anstelle von $\langle\, \sigma{\downarrow_1}\,[\alpha \leftarrow \nu], \sigma{\downarrow_2}\,\rangle$ schreiben. Ebenfalls werden wir häufig die Projektion auf die richtige Komponente nicht explizit notieren, wenn diese durch den Zusammenhang eindeutig festliegt. Wir schreiben also einfach $(\sigma\ \alpha)$ anstelle von $(\sigma{\downarrow_1}\ \alpha)$. Das gleiche gilt für Zugriffe auf Umgebungen $\rho \in Env$.

Sehen wir uns die erste Gleichung der obigen Definition etwas genauer an: Ist $\nu_0 \in Object$, so ist $\nu_0{\downarrow_2} \in InstVar \longrightarrow Location$. Durch die Transformation des ursprünglichen Vererbungsgraphen in eine Übersetzungszeitumgebung, also ein Programm ohne Vererbung, können wir bereits durch die statische Semantik sicherstellen, daß ν_0 dann auch eine Instanzvariable y hat. Damit ist $(\nu_0{\downarrow_2}\ y) \in Location$ und damit $(\sigma\ (\nu_0{\downarrow_2}\ y)) \in Value$. Schließlich ist mit $\kappa \in ECont$ $(\kappa\,(\sigma\ (\nu_0{\downarrow_2}\ y))\,\sigma) \in Store$ und damit $\lambda\,\sigma\,.\,(\kappa\,(\sigma\ (\nu_0{\downarrow_2}\ y))\,\sigma) \in Cont$. Da *wrong* ebenfalls in *Cont* liegt, ist also der gesamte Ausdruck $\{\lambda\,\nu_0\,.\,\ldots\}$ ein Element von *ECont*. Die Gleichung ist wohlgetypt.

Ähnliche Überlegungen zeigen diese Tatsache auch für die restlichen semantischen Gleichungen, die wir in diesem Kapitel definieren werden. Der Leser möge sich im einzelnen davon überzeugen.

Zur Bildung einer Instanz definieren wir eine kleine Hilfsfunktion. In der Umgebung ist zu den Framenamen die jeweilige Liste der Instanzvariablennamen notiert. Diese Instanzvariablen werden mit neu allozierten Speicherplätzen assoziiert. Die entstehende Abbildung wird zusammen mit dem Framenamen als Objekt an die Continuation übergeben.

Definition 4.13

$$make\text{-}instance : Framename \longrightarrow (InstVar \longrightarrow Location) \longrightarrow InstVar^* \longrightarrow ECont \longrightarrow Cont$$

$$\begin{aligned}
&make\text{-}instance\ A\,\eta\,\langle\rangle\,\kappa\,\sigma\ =_{\text{Def}}\ \kappa\,(\langle A, \eta\,\rangle\ in\ Value)\,\sigma\\
&make\text{-}instance\ A\,\eta\,\langle y, y^*\,\rangle\,\kappa\,\sigma\ =_{\text{Def}}\\
&\quad(\lambda\,\alpha\,.\,make\text{-}instance\ A\ \eta\,[y \leftarrow \alpha]\ y^*\,\kappa\,(extend\,\alpha\,\sigma)\,)\,(new\ \sigma)
\end{aligned}$$

□

Die Funktion *make-instance* ist primitiv rekursiv über der (endlichen) Liste $y^* \in InstVar^*$. Mit dieser Hilfsfunktion ist die Bedeutung von *instance-of* A wie folgt definiert:

Definition 4.14

$$\mathcal{E}\,[\![\,instance\text{-}of\ A\,]\!]\,\rho\kappa =_{\text{Def}} make\text{-}instance\ A\,(\lambda\,y\,.\,\bot_{\text{Location}})\,(\rho{\downarrow_2}\ A)\,\kappa$$

□

Wir sehen, daß Objekte ganz natürlich wie *Record*-artige Daten repräsentiert sind. Der einzige Unterschied ist, daß sie zusätzlich den Namen des Frames beinhalten, dessen direkte Instanz sie sind. Insbesondere sind die Methoden nicht Bestandteil der Objekte. Natürlich können Instanzvariablen wiederum beliebige Daten enthalten, z.B. wieder Objekte oder auch Funktionen. Derartige „funktionale Komponenten" von Objekten dürfen wir aber nicht mit Methoden verwechseln; sie erlauben beispielsweise i. a. nicht die Referenz auf das Objekt selbst. Unterscheiden wir in einer Programmiersprache begrifflich zwischen *Programm* und *Daten*, so rechnen wir die Objekte eindeutig den Daten zu.

4.3 Semantik von Ausdrücken

Zur Definition der Semantik der restlichen in unserer Sprache vorkommenden Ausdrücke fehlt uns nun zunächst die Festlegung einiger weiterer semantischer Bereiche. Wir haben bereits zu Beginn dieses Kapitels motiviert, wie der Bereich der Funktionen auszusehen hat. Tatsächlich haben wir null– und einstellige Funktionen zugelassen, die natürlich in verschiedenen Bereichen modelliert werden. Zudem wollen wir sicherstellen, daß nicht terminierende Funktionen dennoch wohldefinierte semantische Elemente sind.

4.3.1 Semantik von Funktionen

Es läßt sich zeigen, daß das Modell des getypten λ-Kalküls, das unserer Semantik letztlich zugrundeliegt, *extensional* ist, d.h. daß zwei stetige Funktionen f und g bereits dann gleich sind, wenn sie für alle Argumente gleiche Werte liefern. Das hätte aber zur Folge, daß $\lambda\nu\,.\,\bot$ bereits gleich $\bot$ wäre. Um dies zu vermeiden, *liften* wir die funktionalen Bereiche. Dies verkompliziert unsere Notation ein wenig. Wir legen aber gleichzeitig fest, daß wir für $\lambda a\,.\,b \in [A \longrightarrow B]$ auch $\lambda a\,.\,b$ für das Element $\langle 0, \lambda a\,.\,b\,\rangle \in [A \longrightarrow B]_\bot$ schreiben. Die Stellen, an denen wir dies tun, sind aus dem Zusammenhang klar zu erkennen.

Definition 4.15

$$\begin{array}{rcl} \varphi \in FVal & = & FunVal + CFunVal \\ FunVal & = & [Value \longrightarrow ECont \longrightarrow Cont]_\bot \\ CFunVal & = & [ECont \longrightarrow Cont]_\bot \end{array}$$

□

Der Bereich *Value* der denotierten Werte ist nun die separierte Summe all derjenigen Bereiche, die wir als *Daten* auffassen wollen. Dazu gehören in der hier vorgestellten Sprache auch die Funktionen. Dies führt letztlich dazu, daß die semantischen Bereiche reflexive Bereiche sind, da ihre Definition ein wechselseitig rekursives Gleichungssystem bildet.

Definition 4.16

$$\nu \in Value = Nat + Bool + FVal + Object$$

□

Damit ist auch klar, wie die semantischen Gleichungen für $\lambda x\,.\,e$ und $\lambda()\,.\,e$ aussehen müssen. Sie übergeben die entsprechenden Elemente in *FunVal* bzw. in *CFunVal* an die Continuation. Es sei hier nocheinmal darauf hingewiesen, daß das „Einfrieren" der Umgebung $\rho \in Env$ zum Zeitpunkt der Funktionsdefinition statische Identifikatorbindung hervorruft.

Definition 4.17

Die semantischen Gleichungen[2] für $\lambda x\,.\,e$ und $\lambda()\,.\,e$ definieren wir wie folgt:

$$\begin{array}{lll} \mathcal{E}[\![\lambda x\,.\,e]\!]\,\rho\kappa & =_{\text{Def}} & \kappa\,(\lambda\nu'.\,\lambda\kappa'.\,\lambda\sigma'.\,\mathcal{E}[\![e]\!]\;\rho[x \leftarrow \nu']\;\kappa'\,\sigma'\;\mathit{in\ Value}) \\ \mathcal{E}[\![\lambda()\,.\,e]\!]\,\rho\kappa & =_{\text{Def}} & \kappa\,(\lambda\kappa'.\,\lambda\sigma'.\,\mathcal{E}[\![e]\!]\;\rho\kappa'\,\sigma'\;\mathit{in\ Value}) \end{array}$$

□

Folgerichtig müssen bei der Anwendung $e_0\,(\,e_1\,)$ das Argument und die aktuell gültige Continuation an die Funktion übergeben werden. Da e_0 ein beliebiger Ausdruck sein kann, muß natürlich zusätzlich sichergestellt sein, daß e_0 tatsächlich eine Funktion, in diesem Fall in *FunVal*, geliefert hat. In den beiden folgenden semantischen Gleichungen haben wir der Lesbarkeit halber die Projektion auf den tatsächlichen Bereich der Funktionen unterdrückt.

Definition 4.18

Für die Funktionsapplikationen $e_0\,(\,e_1\,)$ und $e_0\,()$ legen wir die folgenden semantischen Gleichungen fest:

$$\begin{array}{l} \mathcal{E}[\![e_0\,(\,e_1\,)]\!]\,\rho\kappa\ =_{\text{Def}} \\ \qquad \mathcal{E}[\![e_0]\!]\;\rho\{\lambda\nu_0\,.\,\nu_0 \in \mathit{FunVal} \rightarrow \mathcal{E}[\![e_1]\!]\;\rho\{\lambda\nu_1\,.\,\nu_0\,|_{\mathit{FunVal}}\;\nu_1\,\kappa\}, \mathit{wrong}\} \end{array}$$

$$\mathcal{E}[\![e_0\,()]\!]\,\rho\kappa \quad =_{\text{Def}} \quad \mathcal{E}[\![e_0]\!]\;\rho\{\lambda\nu_0\,.\,\nu_0 \in \mathit{CFunVal} \rightarrow \nu_0\,|_{\mathit{CFunVal}}\;\kappa, \mathit{wrong}\}$$

□

An dieser Stelle soll eine kleine Beispielrechnung ein Gefühl für den Umgang mit den bis jetzt definierten semantischen Gleichungen vermitteln. Wir haben im Kapitel 2 *Blöcke* als syntaktische Erweiterung unserer Sprache durch

$$\mathit{let}\ x\ =\ e_1\ \mathit{in}\ e_2\ \equiv_{\text{Def}}\ (\lambda x.e_2)(e_1)$$

definiert. Im folgenden rechtfertigen wir den Verzicht auf eine zusätzliche semantische Gleichung für Blöcke, indem wir zeigen, daß die Bedeutung von $(\lambda x.e_2)(e_1)$ tatsächlich der „natürlichen" Semantik von $\mathit{let}\ x\ =\ e_1\ \mathit{in}\ e_2$ entspricht. Letztere läßt sich intuitiv wie folgt angeben: Wir haben die Bedeutung von e_2 in einer Umgebung zu bestimmen, in der x zusätzlich an den Wert von e_1 gebunden ist. Man beachte, daß dabei „freie" Variablenvorkommen in e_1 außerhalb des *let*-Blocks gebunden werden und somit *rekursive* Definitionen durch *let* *nicht* möglich sind. Im Stile der hier vorliegenden Continuation-Semantik würden wir

$$\mathcal{E}[\![\,\mathit{let}\ x\ =\ e_1\ \mathit{in}\ e_2\,]\!]\ \rho\kappa\ =_{\text{Def}}\ \mathcal{E}[\![e_1]\!]\;\rho\{\lambda\nu_1\,.\,\mathcal{E}[\![e_2]\!]\;\rho[x \leftarrow \nu_1]\;\kappa\}$$

definiert haben. Der nun folgende Satz zeigt den Kern der im vorigen Absatz enthaltenen Behauptung.

[2]Natürlich sind die Funktionen $\lambda\alpha.(f\ \alpha)$ und f gleich in dem Bereich $[A \longrightarrow B]$ der stetigen Funktionen von A nach B. Wir bevorzugen in den semantischen Gleichungen häufig die erste Schreibweise.

Satz 4.19

Für beliebige $\rho \in Env$ *und* $\kappa \in ECont$ *gilt*

$$\mathcal{E}[\![(\lambda x\,.\,e_1)(e_2)]\!]\,\rho\kappa \;=\; \mathcal{E}[\![e_2]\!]\,\rho\{\lambda\nu_2\,.\,\mathcal{E}[\![e_1]\!]\;\rho[x \leftarrow \nu_2]\;\kappa\}\,.$$

Beweis:

$\mathcal{E}[\![(\lambda x\,.\,e_1)(e_2)]\!]\,\rho\kappa$

$(\lambda x\,.\,e_1)(e_2)$ ist eine Funktionsapplikation mit Funktion $\lambda x\,.\,e_1$ und Argument e_2 . Deshalb erhalten wir mit der semantischen Gleichung für Funktionsapplikationen

$= \mathcal{E}[\![\lambda x\,.\,e_1]\!]\,\rho\{\lambda\nu_0\,.\,\nu_0 \in \mathit{FunVal} \rightarrow \mathcal{E}[\![e_2]\!]\,\rho\{\lambda\nu_2\,.\,\nu_0\,|_{\mathit{FunVal}}\,\nu_2\,\kappa\}\,,\ \mathit{wrong}\}$

Wenden wir nun die semantische Gleichung für Abstraktionen an, so haben wir die Continuation $\{\lambda\nu_0\,.\,\nu_0 \in \mathit{FunVal} \rightarrow \ldots\}$ auf die Funktion (φ *in Value*) anzuwenden, wobei $\varphi = \lambda\nu'.\,\lambda\kappa'.\,\lambda\sigma'.\,\mathcal{E}[\![e_1]\!]\;\rho[x \leftarrow \nu']\;\kappa'\,\sigma'$ ist. Da (φ *in Value*) $\in \mathit{FunVal}$ und überdies (φ *in Value*) $|_{\mathit{FunVal}} = \varphi$ gilt, erhalten wir

$= \mathcal{E}[\![e_2]\!]\,\rho\{\lambda\nu_2\,.\,(\lambda\nu'.\,\lambda\kappa'.\,\lambda\sigma'.\,\mathcal{E}[\![e_1]\!]\;\rho[x \leftarrow \nu']\;\kappa'\,\sigma')\,\nu_2\,\kappa\}$

Wenden wir nun innen die Funktion φ auf ν_2 und κ an, so ergibt sich

$= \mathcal{E}[\![e_2]\!]\,\rho\{\lambda\nu_2\,.\,\lambda\sigma'\,.\,\mathcal{E}[\![e_1]\!]\;\rho[x \leftarrow \nu_2]\;\kappa\,\sigma'\}$
$= \mathcal{E}[\![e_2]\!]\,\rho\{\lambda\nu_2\,.\,\mathcal{E}[\![e_1]\!]\;\rho[x \leftarrow \nu_2]\;\kappa\}$

qed.

Mit der o.a. Semantik von *let*-Blöcken können wir nun die folgende Aussage als Korollar aus diesem Satz formulieren:

$$\mathcal{E}[\![let\ x\ =\ e_1\ in\ e_2]\!] = \mathcal{E}[\![(\lambda x.e_2)\,(e_1)]\!]$$

In dem Beweis von Satz (4.19) wird ebenfalls deutlich, wie durch eine Continuation-Semantik der Kontrollfluß innerhalb eines Programmausdrucks modelliert wird: Liefert ein Teilausdruck „elementar“ ein *Datum*, wie z.B. Konstanten, Variablen oder auch $\lambda x\,.\,e_1$ in unserem Beispiel, so wird die Continuation auf dieses Datum angewendet, und der Kontrollfluß „schreitet“ auf diese Weise „voran“.

4.3.2 Semantik von Nachrichten

Der nun folgende Abschnitt widmet sich der Semantik von Nachrichten. Wie bereits an anderer Stelle bemerkt, fassen wir bereits $e.m$ als Nachricht auf und nicht erst die Anwendung

$e.m\,(e_1)$ oder $e.m\,()$. Diese Ausdrücke bedeuten in unserer Sprache einfach die Anwendung von $e.m$ im Sinne einer Funktionsapplikation.

Was bedeutet nun $e.m$? Angenommen, e bezeichnet tatsächlich ein Objekt, das Empfängerobjekt der Nachricht, so beinhaltet dieses Objekt seinen Framenamen A . Zu A und m ist in unserem Programm eventuell eine Methode definiert. Diese Methode $\lambda\, self\,.\,\lambda\, x\,.\,e$ oder $\lambda\, self\,.\,\lambda\,()\,.\,e$ haben wir in dem nunmehr vorliegenden abstrakten Programm ohne Vererbung dem verallgemeinerten Identifikator $\langle A, m\,\rangle$ zugeordnet. Wir gehen also davon aus, daß die Bedeutung dieser Methode in der Umgebung $\rho \in \mathit{Env}$ ebenfalls dem Identifikator $\langle A, m\,\rangle$ zugeordnet ist. Durch Anwenden der Methode auf das Empfängerobjekt erhalten wir die Methodenfunktion, also die Bedeutung von $\lambda\, x\,.\,e$ bzw. von $\lambda\,()\,.\,e$, eben mit der zusätzlichen Bindung der Variablen *self* an das Empfängerobjekt.

Definition 4.20

Die Semantik einer Nachricht $e.m$ ist durch die folgende semantische Gleichung gegeben:

$$\begin{aligned}&\boldsymbol{\mathcal{E}}\,[\![\,e.m\,]\!]\,\rho\kappa\ =_{\mathrm{Def}}\\ &\qquad \boldsymbol{\mathcal{E}}\,[\![\,e\,]\!]\,\rho\{\lambda\,\nu_0\,.\,\nu_0\ \in\ \mathit{Object} \to (\rho{\downarrow_1}\ \langle \nu_0{\downarrow_1}\,, m\,\rangle)\,|_{\mathit{FunVal}}\ \nu_0\,\kappa\,,\ \mathit{wrong}\}\end{aligned}$$

□

Die Bedeutung einer Nachricht läßt sich deshalb so verhältnismäßig einfach definieren, weil wir durch die Durchführung der Vererbung aus dem originalen Programm ein Programm erzeugt haben, in dem zu A und m genau dann eine Methode existiert, wenn $\langle A, m\,\rangle$ an eine solche gebunden ist. In dem originalen Programm ist dies natürlich nicht allgemein der Fall, da Frames Methoden erben können.

Um die Definition der Semantik zunächst abzuschließen, haben wir noch vier weitere semantische Gleichungen sowie die Semantik von Konstanten zu definieren. In der folgenden Definition geben wir die Bedeutung der Ausdrücke *if* e_0 *then* e_1 *else* e_2 und $e_1\,;\,e_2$ an. Speziell für das Conditional *if* sei bemerkt, daß jedes Element außer *false* als wahr bewertet wird, wie dies in Lisp-artigen Sprachen üblich ist.

Definition 4.21

$$\boldsymbol{\mathcal{E}}\,[\![\,\mathit{if}\ e_0\ \mathit{then}\ e_1\ \mathit{else}\ e_2\,]\!]\,\rho\kappa\ \ =_{\mathrm{Def}}\ \ \boldsymbol{\mathcal{E}}\,[\![\,e_0\,]\!]\,\rho\{\lambda\,\nu_0\,.\,\nu_0 = \mathit{false} \to \boldsymbol{\mathcal{E}}\,[\![\,e_2\,]\!]\,\rho\kappa\,,\ \boldsymbol{\mathcal{E}}\,[\![\,e_1\,]\!]\,\rho\kappa\}$$

$$\boldsymbol{\mathcal{E}}\,[\![\,e_1\,;\,e_2\,]\!]\,\rho\kappa\ \ =_{\mathrm{Def}}\ \ \boldsymbol{\mathcal{E}}\,[\![\,e_1\,]\!]\,\rho\{\lambda\,\nu\,.\,\boldsymbol{\mathcal{E}}\,[\![\,e_2\,]\!]\,\rho\kappa\}$$

□

Gewöhnliche Variablen, und dazu zählen wir auch die Variable *self* , erhalten ihre Werte in der Umgebung $\rho \in \mathit{Env}$. Man vergleiche dazu die semantische Gleichung für $\lambda\, x\,.\,e$. Wir brauchen sie nicht im Speicher zu allozieren, da in unserer Sprache keine Seiteneffekte auf gewöhnliche Variablen möglich sind.

Definition 4.22

$$\boldsymbol{\mathcal{E}}\,[\![\,x\,]\!]\,\rho\kappa\ \ =_{\mathrm{Def}}\ \ \lambda\,\sigma\,.\,\kappa\,(\rho{\downarrow_1}\ \ x)\,\sigma$$

□

4.3.3 Semantik von Konstanten

Wir haben in Kapitel 2 eine ganze Reihe von Konstanten syntaktisch in Programmen erlaubt:

$$c \in \mathit{Const} =_{\mathrm{Def}} \mathbb{N}_0 + \{\, \mathit{true}, \mathit{false} \,\} + \{\, +, -, *, \mathit{mod}, \mathit{div}, \leq, <, =, \mathit{and}, \mathit{not}, \mathit{or} \,\}$$

Für Konstanten definieren wir nun ein semantisches Funktional $\mathcal{K}\,[\![\,\bullet\,]\!] \in \mathit{Const} \longrightarrow \mathit{Value}$. Die Konstanten *true* und *false* werden natürlich auf die entsprechenden Elemente in *Bool*, jedes $n \in \mathbb{N}_0$ auf das entsprechende Element $n \in \mathit{Nat}$ abgebildet.

Wir wollen nicht für jede einzelne Standardfunktion die Konstantensemantik tatsächlich angeben. Stattdessen diskutieren wir sie am Beispiel von $+ \in \mathit{Const}$. Der flache Bereich *Nat* besitzt eine zweistellige stetige Funktion $+ \in \mathit{Nat} \times \mathit{Nat} \longrightarrow \mathit{Nat}$. Wir haben stattdessen eine einstellige Funktion aus dem Bereich *FunVal* zu definieren, die ihr Argument, sagen wir ν_1, auf eine ebenfalls einstellige Funktion abbildet, die ihrerseits ihr Argument zu ν_1 addiert und an die Continuation weiterreicht. Ebenfalls ist zu überprüfen, ob die Argumente tatsächlich Zahlen sind. Wir fassen die Bemerkungen des vorigen Absatzes und die Definition von $+$ in der folgenden Definition zusammen:

Definition 4.23

$$\begin{array}{lll}
\mathcal{K}\,[\![\, n \,]\!] & =_{\mathrm{Def}} & (n \ \mathit{in}\ \mathit{Value}) \\
\mathcal{K}\,[\![\, \mathit{true} \,]\!] & =_{\mathrm{Def}} & (\mathit{true}\ \mathit{in}\ \mathit{Value}) \\
\mathcal{K}\,[\![\, \mathit{false} \,]\!] & =_{\mathrm{Def}} & (\mathit{false}\ \mathit{in}\ \mathit{Value}) \\
\mathcal{K}\,[\![\, + \,]\!] & =_{\mathrm{Def}} & (\lambda\nu\,.\,\lambda\kappa\,.\,\lambda\,\sigma\,.\,\nu \in \mathit{Nat} \rightarrow \\
 & & \quad \kappa\ ((\lambda\nu'\,.\,\lambda\kappa'\,.\,\lambda\,\sigma'\,.\,\nu' \in \mathit{Nat} \rightarrow \\
 & & \quad\quad \kappa'\ (\,(\nu + \nu')\ \mathit{in}\ \mathit{Value}\,)\ \sigma', \\
 & & \quad\quad \mathit{wrong}\ \sigma'\,)\ \mathit{in}\ \mathit{Value}\,)\ \sigma, \\
 & & \quad \mathit{wrong}\ \sigma\,)\ \mathit{in}\ \mathit{Value}
\end{array}$$

□

Schließlich ist die Semantik einer Konstanten als Ausdruck dann gegeben durch:

Definition 4.24

$$\mathcal{E}\,[\![\, c \,]\!]\,\rho\kappa \ =_{\mathrm{Def}}\ \kappa\,(\mathcal{K}\,[\![\, c \,]\!]\,).$$

□

Dies schließt die Definition der dynamischen Semantik zunächst ab. Fassen wir kurz zusammen, was wir bis jetzt definiert haben: Zunächst haben wir unser originales Programm, das in der Form $p = d\,;\ e \in \mathit{Program}$ in abstrakter Syntax vorlag, durch Anwenden der im Kapitel 2 definierten Funktion $\mathbf{IG}\,[\![\,\bullet\,]\!]$ auf den Deklarationsteil d in eine äquivalente andere syntaktische Form gebracht. Daraus ist also $\langle \gamma, e \,\rangle = \langle \mathbf{IG}\,[\![\, d \,]\!]\ \gamma_\perp, e \,\rangle \in \mathit{InhGraph} \times \mathit{Expr}$ entstanden. Dann haben wir in Kapitel 3 durch die Anwendung einer geeigneten Vererbungsstrategie den Deklarationsteil $\gamma \in \mathit{InhGraph}$ in eine Übersetzungszeitumgebung $g \in \mathit{CEnv}$ transformiert.

Diese Transformation ist nicht mehr eine äquivalente Umformung; verschiedene Vererbungsstrategien liefern durchaus verschiedene Ergebnisse. Insgesamt ist aus unserem Programm damit ein Paar $\langle g, e \rangle \in \mathit{CEnv} \times \mathit{Expr}$ geworden. $\langle g, e \rangle$ repräsentiert immer noch ein Programm unserer Sprache, nun allerdings eines ohne Vererbung. Den in diesem Programm vorkommenden Ausdrücken, insbesondere also auch dem Hauptprogrammausdruck e , haben wir nun eine denotationelle Semantik gegeben. Im Kapitel 5 zeigen wir, daß sich diese Semantik zunächst auf g und dann auch auf $\langle g, e \rangle$ ausdehnen läßt.

4.4 Die Zielsprache

In diesem Abschnitt definieren wir zwei zusätzliche Formen von Ausdrücken, die in Programmen auftauchen können, die durch die Anwendung des Übersetzers entstehen. Diesen geben wir dann schließlich im nächsten und letzten Abschnitt dieses Kapitels induktiv über den syntaktischen Aufbau von Ausdrücken an. Der Übersetzer läßt sich, wie die denotationelle Semantik, ebenfalls auf Deklarationsteile und letztlich auf Programme ausdehnen. Dies tun wir im Kapitel 5 simultan mit der Ausdehnung der denotationellen Semantik und dem Beweis der Korrektheit des Übersetzers. Es ist klar, daß sowohl die Semantik als auch die Übersetzung essentiell von der benutzten Vererbungsstrategie abhängen.

Wir werden Nachrichten durch einen endlichen generischen Auswahloperator ersetzen, der die richtige Methode auswählt und durch Anwendung auf ein Objekt die entsprechende Methodenfunktion liefert. Diesen Operator nennen wir, in Anlehnung an Lisp, *typecase*. Im vorigen Abschnitt haben wir bereits erwähnt, daß die nötigen Methoden in der Übersetzungszeitumgebung $g \in \mathit{CEnv}$ bereits an Identifikatoren der Form $\langle A, m \rangle$ gebunden sind. Um lästiges Umbenennen zu vermeiden, lassen wir $\langle A, m \rangle$ als Identifikatoren in Programmen zu:

Wir definieren also, daß zusätzlich zu den schon vorhandenen Ausdrücken auch Ausdrücke der Form

$$e ::= \langle A, m \rangle$$

auftreten dürfen.

Definition 4.25

Die Semantik von $\langle A, m \rangle$ definieren wir wie die Semantik von Variablen x durch

$$\mathcal{E}\,[\![\, \langle A, m \rangle \,]\!]\,\rho\kappa \quad =_{\mathrm{Def}} \quad \lambda\sigma\,.\,\kappa\,(\rho{\downarrow_1}\ \langle A, m \rangle)\,\sigma\,.$$

□

Zusätzlich erlauben wir Ausdrücke der Form

$$e \ ::= \ \mathit{typecase}\ e\ \mathit{of}\ A_1 : e_1\,,\ \ldots\ , A_n : e_n$$

Die Semantik von $\mathit{typecase}\ e\ \mathit{of}\ A_1 : e_1\,,\ \ldots\ , A_n : e_n$ definieren wir wie folgt: Bezeichnet e ein Objekt, so selektieren wir anhand des Framenamen dieses Objektes einen der Ausdrücke $e_1\,,\ \ldots\ ,\ e_n$, und wenden seine Bedeutung auf das ursprüngliche Objekt an.

Definition 4.26

Die Bedeutung des Ausdrucks *typecase e of* $A_1 : e_1, \ldots, A_n : e_n$ definieren wir durch die folgende semantische Gleichung:

$$\begin{aligned}
&\boldsymbol{\mathcal{E}}\,[\![\, \textit{typecase } e \textit{ of } A_1 : e_1, \ldots, A_n : e_n \,]\!]\,\rho\kappa =_{\mathrm{Def}} \\
&\quad \boldsymbol{\mathcal{E}}\,[\![\, e \,]\!]\,\rho\,\{\lambda\nu_0\,.\,\nu_0 \in \textit{Object} \to \nu_0{\downarrow_1} = A_1 \to \boldsymbol{\mathcal{E}}\,[\![\, e_1 \,]\!]\,\rho\,\{\lambda\nu_1\,.\,\nu_1 \mid_{\textit{FunVal}} \nu_0\,\kappa\,\}, \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vdots \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \nu_0{\downarrow_1} = A_n \to \boldsymbol{\mathcal{E}}\,[\![\, e_n \,]\!]\,\rho\,\{\lambda\nu_n\,.\,\nu_n \mid_{\textit{FunVal}} \nu_0\,\kappa\,\}, \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{wrong} \\
&\qquad \textit{wrong}\,\}
\end{aligned}$$

□

Der Ausdruck *typecase e of* $A_1 : e_1, \ldots, A_n : e_n$ ist nur dann wohldefiniert, wenn e ein Objekt bezeichnet, dessen zugeordneter Framename eines der A_i, $i \in \{1, \ldots, n\}$ ist, und wenn das entsprechende e_i eine Funktion aus dem semantischen Bereich *FunVal* ist. Das Ergebnis dieses Ausdrucks wird auf das Objekt e angewendet. Im übersetzten Programm werden die e_i gerade die Methodenfunktionsnamen $\langle A_i, m \rangle$ sein. Sie liefern, angewendet auf das Objekt e, die zugehörige Methodenfunktion. Dabei setzen wir natürlich voraus, daß die Bedeutung aller Ausdrücke in einer adäquaten Umgebung bestimmt wird. Daß dies der Fall ist, zeigen wir im nächsten Kapitel.

Die beiden folgenden Tabellen zeigen die semantischen Bereiche und die semantischen Gleichungen noch einmal im Zusammenhang.

Semantische Bereiche

$$
\begin{array}{rcll}
x, f & \in & \mathit{Var} & \\
y & \in & \mathit{InstVar} & \\
A & \in & \mathit{Framename} & \\
m & \in & \mathit{Message} & \\
I & \in & \mathit{Ident} & = \quad \mathit{Var} + (\mathit{Framename} \times \mathit{Message}) \\
 & & & \\
\rho & \in & \mathit{Env} & = \quad (\mathit{Ident} \longrightarrow \mathit{Value}) \times \ (\mathit{Framename} \longrightarrow \mathit{InstVar}^*) \\
\alpha & \in & \mathit{Location} & \\
\sigma & \in & \mathit{Store} & = \quad (\mathit{Location} \longrightarrow \mathit{Value}) \times (\mathit{Location} \longrightarrow \mathit{Bool}) \\
 & & & \\
\theta & \in & \mathit{Cont} & = \quad \mathit{Store} \longrightarrow \mathit{Answer} \\
 & & \mathit{Answer} & = \quad \mathit{Value} \times \mathit{Store} \\
\kappa & \in & \mathit{ECont} & = \quad \mathit{Value} \longrightarrow \mathit{Cont} \\
 & & & \\
\varphi & \in & \mathit{FVal} & = \quad \mathit{FunVal} + \mathit{CFunVal} \\
 & & \mathit{FunVal} & = \quad [\mathit{Value} \longrightarrow \mathit{ECont} \longrightarrow \mathit{Cont}]_\perp \\
 & & \mathit{CFunVal} & = \quad [\mathit{ECont} \longrightarrow \mathit{Cont}]_\perp \\
o & \in & \mathit{Objekt} & = \quad \mathit{Framename} \times (\mathit{InstVar} \longrightarrow \mathit{Location}) \\
 & & & \\
b & \in & \mathit{Bool} & = \quad \{\perp, \mathit{true}, \mathit{false}, \top\} \\
n & \in & \mathit{Nat} & = \quad \{\perp, 0, 1, \ldots, \top\} \\
 & & & \\
\nu & \in & \mathit{Value} & = \quad \mathit{Nat} + \mathit{Bool} + \mathit{FVal} + \mathit{Object}
\end{array}
$$

Semantische Gleichungen

$$
\begin{array}{lcl}
\mathcal{E}\,[\![\, c \,]\!]\,\rho\kappa & = & \kappa\,(\mathcal{K}\,[\![\, c \,]\!]\,) \\
\mathcal{E}\,[\![\, x \,]\!]\,\rho\kappa & = & \lambda\sigma\,.\,\kappa\,(\rho{\downarrow_1}\ x)\,\sigma \\
\mathcal{E}\,[\![\, \langle A, m \,\rangle \,]\!]\,\rho\kappa & = & \lambda\sigma\,.\,\kappa\,(\rho{\downarrow_1}\ \langle A, m \,\rangle)\,\sigma
\end{array}
$$

$$
\mathcal{E}\,[\![\, \mathit{if}\ e_0\ \mathit{then}\ e_1\ \mathit{else}\ e_2 \,]\!]\,\rho\kappa \ = \ \mathcal{E}\,[\![\, e_0 \,]\!]\,\rho\,\{\lambda\nu_0\,.\,\nu_0 = \mathit{false} \to \mathcal{E}\,[\![\, e_2 \,]\!]\,\rho\kappa\,,\ \mathcal{E}\,[\![\, e_1 \,]\!]\,\rho\kappa\}
$$

$$
\begin{array}{l}
\mathcal{E}\,[\![\, e_0\,(\,e_1\,) \,]\!]\,\rho\kappa \ = \\
\qquad \mathcal{E}\,[\![\, e_0 \,]\!]\,\rho\,\{\lambda\nu_0\,.\,\nu_0 \in \mathit{FunVal} \to \mathcal{E}\,[\![\, e_1 \,]\!]\,\rho\,\{\lambda\nu_1\,.\,\nu_0\,|_{\mathit{FunVal}}\ \nu_1\,\kappa\}\,, \mathit{wrong}\}
\end{array}
$$

$$
\begin{array}{lcl}
\mathcal{E}\,[\![\, e_0\,() \,]\!]\,\rho\kappa & = & \mathcal{E}\,[\![\, e_0 \,]\!]\,\rho\,\{\lambda\nu_0\,.\,\nu_0 \in \mathit{CFunVal} \to \nu_0\,|_{\mathit{CFunVal}}\ \kappa\,, \mathit{wrong}\} \\
\mathcal{E}\,[\![\, \lambda x\,.\,e \,]\!]\,\rho\kappa & = & \kappa\,(\lambda\nu'.\,\lambda\kappa'.\,\lambda\sigma'.\,\mathcal{E}\,[\![\, e \,]\!]\ \rho[x \leftarrow \nu']\ \kappa'\,\sigma'\ \mathit{in\ Value}) \\
\mathcal{E}\,[\![\, \lambda()\,.\,e \,]\!]\,\rho\kappa & = & \kappa\,(\lambda\kappa'.\,\lambda\sigma'.\,\mathcal{E}\,[\![\, e \,]\!]\ \rho\kappa'\,\sigma'\ \mathit{in\ Value}) \\
\mathcal{E}\,[\![\, e_1\,;\,e_2 \,]\!]\,\rho\kappa & = & \mathcal{E}\,[\![\, e_1 \,]\!]\,\rho\,\{\lambda\nu\,.\,\mathcal{E}\,[\![\, e_2 \,]\!]\,\rho\kappa\}
\end{array}
$$

$$
\begin{array}{l}
\mathcal{E}\,[\![\, \mathit{typecase}\ e\ \mathit{of}\ A_1 : e_1\,,\ \ldots\ , A_n : e_n \,]\!]\,\rho\kappa \ = \\
\qquad \mathcal{E}\,[\![\, e \,]\!]\,\rho\,\{\lambda\nu_0\,.\,\nu_0 \in \mathit{Object} \to \nu_0{\downarrow_1} = A_1 \to \mathcal{E}\,[\![\, e_1 \,]\!]\,\rho\,\{\lambda\nu_1\,.\,\nu_1\,|_{\mathit{FunVal}}\ \nu_0\,\kappa\}\,, \\
\qquad\qquad\qquad\qquad\qquad\qquad \vdots \\
\qquad\qquad\qquad\qquad\qquad\qquad \nu_0{\downarrow_1} = A_n \to \mathcal{E}\,[\![\, e_n \,]\!]\,\rho\,\{\lambda\nu_n\,.\,\nu_n\,|_{\mathit{FunVal}}\ \nu_0\,\kappa\}\,, \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{wrong} \\
\qquad\qquad \mathit{wrong}\,\}
\end{array}
$$

$$
\begin{array}{l}
\mathcal{E}\,[\![\, \mathit{self}\,.\,y \,]\!]\,\rho\kappa \ = \\
\qquad \mathcal{E}\,[\![\, \mathit{self} \,]\!]\,\rho\,\{\lambda\nu_0\,.\,\nu_0 \in \mathit{Object} \to \lambda\sigma\,.\,\kappa\,(\sigma\ (\nu_0{\downarrow_2}\ y))\,\sigma\,,\ \mathit{wrong}\}
\end{array}
$$

$$
\begin{array}{l}
\mathcal{E}\,[\![\, \mathit{self}\,.\,y\ :=\ e \,]\!]\,\rho\kappa \ = \\
\qquad \mathcal{E}\,[\![\, \mathit{self} \,]\!]\,\rho\,\{\lambda\nu_0\,.\,\nu_0 \in \mathit{Object} \to \mathcal{E}\,[\![\, e \,]\!]\,\rho\,\{\lambda\nu\,.\,\lambda\sigma\,.\,\kappa\,\nu\ \sigma\,[(\nu_0{\downarrow_2}\ y) \leftarrow \nu]\,\}\,,\ \mathit{wrong}\}
\end{array}
$$

$$
\mathcal{E}\,[\![\, \mathit{instance\text{-}of}\ A \,]\!]\,\rho\kappa \ = \ \mathit{make\text{-}instance}\ A\,(\lambda y\,.\,\bot_{\mathrm{Location}})\,(\rho{\downarrow_2}\ A)\,\kappa
$$

$$
\begin{array}{l}
\mathcal{E}\,[\![\, e.m \,]\!]\,\rho\kappa \ = \\
\qquad \mathcal{E}\,[\![\, e \,]\!]\,\rho\,\{\lambda\nu_0\,.\,\nu_0 \ \in\ \mathit{Object} \to (\rho{\downarrow_1}\ \langle \nu_0{\downarrow_1}\,, m \,\rangle)\,|_{\mathit{FunVal}}\ \nu_0\,\kappa\,,\ \mathit{wrong}\}
\end{array}
$$

4.5 Übersetzung von Ausdrücken

Wir wollen nun dieses Kapitel abschließen, indem wir, zunächst nur für Ausdrücke unserer Sprache, eine Übersetzungsfunktion spezifizieren, die die Nachrichten $e.m$ in die entsprechende endliche generische Methodenfunktionsauswahl übersetzt.

Da wir den Übersetzer induktiv über den Aufbau von Ausdrücken definieren wollen, muß er ebenfalls auf den in diesem Abschnitt zusätzlich definierten Zielsprachausdrücken definiert sein. Wir können leicht zeigen, daß ein übersetzter Ausdruck keine Nachrichten der Form $e.m$ mehr enthält. Der Übersetzer ist nicht inkrementell; er muß den Deklarationsteil bzw. die Übersetzungszeitumgebung $g \in \mathit{CEnv}$ komplett kennen. Deshalb parametrisieren wir die Übersetzungsfunktion

$$\mathcal{C}_e^g[\![\bullet]\!] : \mathit{Expr} \longrightarrow \mathit{Expr}$$

mit eben diesem $g \in \mathit{CEnv}$. Damit geht gerade an dieser Stelle die Vererbungsstrategie in den Übersetzer ein, denn sie erzeugt die Übersetzungszeitumgebung g.

Definition 4.27

Wir definieren $\mathcal{C}_e^g[\![\bullet]\!] : \mathit{Expr} \longrightarrow \mathit{Expr}$ induktiv über den Aufbau von e wie folgt:

$$\begin{array}{lll}
\mathcal{C}_e^g[\![c]\!] & =_{\mathrm{Def}} & c \\
\mathcal{C}_e^g[\![x]\!] & =_{\mathrm{Def}} & x \\
\mathcal{C}_e^g[\![\langle A, m\rangle]\!] & =_{\mathrm{Def}} & \langle A, m\rangle \\
\mathcal{C}_e^g[\![\mathit{self}.y]\!] & =_{\mathrm{Def}} & \mathit{self}.y \\
\mathcal{C}_e^g[\![\mathit{instance\text{-}of}\ A]\!] & =_{\mathrm{Def}} & \mathit{instance\text{-}of}\ A \\
 & & \\
\mathcal{C}_e^g[\![\mathit{if}\ e_0\ \mathit{then}\ e_1\ \mathit{else}\ e_2]\!] & =_{\mathrm{Def}} & \mathit{if}\ \mathcal{C}_e^g[\![e_0]\!]\ \mathit{then}\ \mathcal{C}_e^g[\![e_1]\!]\ \mathit{else}\ \mathcal{C}_e^g[\![e_2]\!] \\
\mathcal{C}_e^g[\![e_0\,(e_1)]\!] & =_{\mathrm{Def}} & \mathcal{C}_e^g[\![e_0]\!]\,(\mathcal{C}_e^g[\![e_1]\!]) \\
\mathcal{C}_e^g[\![e_0\,()]\!] & =_{\mathrm{Def}} & \mathcal{C}_e^g[\![e_0]\!]\,() \\
\mathcal{C}_e^g[\![\lambda x\,.\,e]\!] & =_{\mathrm{Def}} & \lambda x\,.\,\mathcal{C}_e^g[\![e]\!] \\
\mathcal{C}_e^g[\![\lambda()\,.\,e]\!] & =_{\mathrm{Def}} & \lambda()\,.\,\mathcal{C}_e^g[\![e]\!] \\
\mathcal{C}_e^g[\![e_1\,;\,e_2]\!] & =_{\mathrm{Def}} & \mathcal{C}_e^g[\![e_1]\!]\,;\,\mathcal{C}_e^g[\![e_2]\!]
\end{array}$$

$$\begin{array}{l}
\mathcal{C}_e^g[\![\mathit{typecase}\ e\ \mathit{of}\ A_1 : e_1\,,\ \ldots\ ,A_n : e_n]\!] =_{\mathrm{Def}} \\
\qquad \mathit{typecase}\ \mathcal{C}_e^g[\![e]\!]\ \mathit{of}\ A_1 : \mathcal{C}_e^g[\![e_1]\!]\,,\ \ldots\ ,A_n : \mathcal{C}_e^g[\![e_n]\!]
\end{array}$$

$$\mathcal{C}_e^g[\![\mathit{self}.y\ :=\ e]\!] \quad =_{\mathrm{Def}} \quad \mathit{self}.y\ :=\ \mathcal{C}_e^g[\![e]\!]$$

$$\begin{array}{l}
\mathcal{C}_e^g[\![e.m]\!] =_{\mathrm{Def}} \\
\qquad \mathit{typecase}\ \mathcal{C}_e^g[\![e]\!]\ \mathit{of}\ A_1 : \langle A_1, m\rangle\,,\ \ldots\ ,A_n : \langle A_n, m\rangle \qquad .
\end{array}$$

Dabei ist $\{A_1, \ldots, A_n\} = \{A \in \mathit{Framename} \mid g{\downarrow_1}\langle A, m\rangle \neq \bot\}$ in der letzten Gleichung gerade die Menge der Framenamen, zu denen in g eine Methode für m definiert ist. Die Übersetzung einer Nachricht $e.m$ hängt also von g ab. Wir benötigen die Kenntnis aller Frames, zu denen in g eine Methode für m vorhanden ist. In diesem Sinne ist g eine *Übersetzungszeitumgebung*. Die elementaren Ausdrücke $e = c$, $e = x$, $e = \langle A, m\rangle$, $e = \mathit{self}.y$ und $e = \mathit{instance\text{-}of}\ A$ werden in sich selbst übersetzt.

□

Wie bereits erwähnt, läßt sich leicht per Induktion über den syntaktischen Aufbau des Ausdrucks e zeigen, daß $\mathcal{C}_e^g[\![e]\!]$ keine Nachricht der Form $e.m$ mehr enthält. Alle Nachrichten sind durch die entsprechende endliche generische Methodenauswahl ersetzt.

Obwohl die nötigen Definitionen erst im folgenden Kapitel zu finden sind, wollen wir an dieser Stelle bereits die Erweiterung des Übersetzers zunächst auf Deklarationsteile $g \in \mathit{CEnv}$ und dann auf Programme $\langle g, e \rangle \in \mathit{CEnv} \times \mathit{Expr}$ kurz skizzieren: g bindet in der ersten Komponente Identifikatoren an Abstraktionen. Dabei sind sowohl Bindungen für die gewöhnlichen Funktionsnamen als auch solche für die verallgemeinerten Methodenidentifikatoren vorhanden. Die zweite Komponente bindet Framenamen an ihre Instanzvariablenlisten. Sie bleibt durch die Übersetzung unverändert. Die übersetzte erste Komponente ergibt sich durch Anwendung des eben definierten Übersetzers auf die Abstraktionen.

Ein übersetzter Deklarationsteil ist also ebenfalls eine Übersetzungszeitumgebung oder ein Deklarationsteil $\mathcal{C}_d[\![g]\!] \in \mathit{CEnv}$, der gerade durch die Anwendung des Übersetzers für Ausdrücke auf die in der ersten Komponente von g an Identifikatoren gebundenen Abstraktionen entsteht. Zusammen mit der Übersetzung des Hauptprogrammausdruckes e entsteht nun ein übersetztes Programm $\langle \mathcal{C}_d[\![g]\!], \mathcal{C}_e^g[\![e]\!] \rangle$, ebenfalls in dem syntaktischen Bereich $\mathit{CEnv} \times \mathit{Expr}$ der abstrakten Programme ohne Vererbung.

Der Schritt, den wir bei der Übersetzung von Programmen tun, ist also scheinbar sehr klein. Daß er tatsächlich größer ist, als der erste Anschein vermuten läßt, wollen wir im folgenden motivieren, um damit dieses Kapitel abzuschließen.

Abstrakte Programme mit Vererbung, also $\langle \gamma, e \rangle \in \mathit{InhGraph} \times \mathit{Expr}$, wie auch die durch die Anwendung der Vererbungsstrategie entstandenen abstrakten Programme $\langle g, e \rangle$ im Bereich $\mathit{CEnv} \times \mathit{Expr}$ ohne Vererbung, lassen auf einfache Weise die Rekonstruktion des konkreten objektorientierten Programms zu. Bindungen der Form

$$f \mapsto \lambda\, x\,.\, e \quad \text{bzw.} \quad \langle A, m \rangle \mapsto \lambda\, \mathit{self}\,.\, \lambda\, x\,.\, e$$

können wir leicht wieder in die entsprechenden Funktions- bzw. Methodendefinitionen

$$\mathit{defun}\ f\ x\ =\ e \quad \text{bzw.}$$

$$\mathit{defmethod}\ A\ m\ x\ =\ e$$

zurücktransformieren. Analog lassen sich aus der zweiten und dritten Komponente von γ bzw. aus der zweiten Komponente von g die jeweiligen Framedefinitionen zurückgewinnen.

In übersetzten Programmen $\langle \mathcal{C}_d[\![g]\!], \mathcal{C}_e^g[\![e]\!] \rangle$ fassen wir die Bindungen

$$\langle A, m \rangle \mapsto \lambda\, \mathit{self}\,.\, \lambda\, x\,.\, e$$

nun nicht mehr als Repräsentation von Methodendefinitionen auf, sondern stellen uns die entsprechenden *Funktionsdefinitionen*

$$\mathit{defun}\ \langle A, m \rangle\ \mathit{self}\ =\ \lambda\, x\,.\, e$$

unter ihnen vor. Übersetzte Programme enthalten weder Methodendefinitionen noch Nachrichten, sondern stattdessen Funktionsdefinitionen und -aufrufe.

Wir erhalten also durch die Übersetzung ein imperatives Programm, das aus einer Reihe von, i.a. wechselseitig rekursiven, Funktionsdefinitionen und einem Hauptprogrammausdruck besteht. Zusätzlich zu den Daten und Operationen des applikativen Kerns sind in diesem Programm dynamisch getypte *Records* sowie, als zusätzliche Operationen, deren dynamische Erzeugung und Zugriffe auf ihre Komponenten vorhanden.

Die Semantik des von unserem Übersetzer $\mathcal{C}_e^g[\![\bullet]\!]$ erzeugten generischen Methodenauswahloperators *typecase* e *of* $A_1 : e_1, \ldots, A_n : e_n$ weicht von der natürlichen Semantik eines *typecase*-Operators, wie er beispielsweise aus Lisp bekannt ist, ab. *typecase* wendet das Ergebnis des selektierten Ausdrucks e_i implizit auf das Resultat von e an. Speziell die durch den Übersetzer erzeugten *typecase*-Ausdrücke der Form

$$\mathit{typecase}\ e\ \mathit{of}\ A_1 : \langle A_1, m\rangle, \ldots, A_n : \langle A_n, m\rangle$$

liefern also nicht das an die $\langle A_i, m\rangle$ gebundene Funktional $\lambda\, \mathit{self}\,.\,\lambda x\,.\,e_i$ oder $\lambda\, \mathit{self}\,.\,\lambda()\,.\,e_i$, sondern die entsprechende Methodenfunktion $\lambda x\,.\,e_i$ bzw. $\lambda()\,.\,e_i$, mit einer Bindung für *self* in e_i an das Empfängerobjekt e.

Zum Abschluß dieses Kapitels wollen wir einen zweiten, *gewöhnlicheren* generischen Auswahloperator angeben und zeigen, auf welche Weise wir diesen im Zielcode hätten benutzen können. Dazu definieren wir die Semantik des Ausdrucks

$$\mathit{typecase}_{neu}\ e\ \mathit{of}\ A_1 : e_1, \ldots, A_n : e_n$$

durch

$$\begin{aligned}\mathcal{E}[\![\,\mathit{typecase}_{neu}\ e\ \mathit{of}\ A_1 : e_1, \ldots, A_n : e_n\,]\!]\,\rho\kappa\ =_{\mathrm{Def}}\\ \mathcal{E}[\![\,e\,]\!]\,\rho\{\lambda\nu_0\,.\,\nu_0 \in \mathit{Object} \to \nu_0{\downarrow_1} = A_1 \to \mathcal{E}[\![\,e_1\,]\!]\,\rho\kappa,\\ \vdots\\ \nu_0{\downarrow_1} = A_n \to \mathcal{E}[\![\,e_n\,]\!]\,\rho\kappa,\\ \mathit{wrong},\\ \mathit{wrong}\,\}\end{aligned}$$

Man sieht, daß dieser Ausdruck im Unterschied zu *typecase* einfach die Bedeutung des selektierten e_i liefert. Wir können nun den folgenden Satz zeigen, dessen Beweis (ein einfaches, aber langwieriges Nachrechnen) wir hier nicht angeben werden:

Satz 4.28

Für alle $\rho \in \mathit{Env}$ *und für alle* $\kappa \in \mathit{ECont}$ *gilt*

$$\begin{aligned}&\mathcal{E}[\![\,\mathit{typecase}\ e\ \mathit{of}\ A_1 : \langle A_1, m\rangle, \ldots, A_n : \langle A_n, m\rangle\,]\!]\,\rho\kappa\\ &\quad = \mathcal{E}[\![\,(\lambda x\,.\,\mathit{typecase}_{neu}\ x\ \mathit{of}\ A_1 : (\langle A_1, m\rangle\ x), \ldots, A_n : (\langle A_n, m\rangle\ x))\,(e)\,]\!]\,\rho\kappa\,.\end{aligned}$$

Die zusätzliche λ-Abstraktion ist nötig, da der Ausdruck e nicht fälschlicherweise mehrfach berechnet werden darf. Der zweite Ausdruck entspricht dem *Block*

$$\begin{aligned}&\mathit{let}\ (x\ =\ e)\ \mathit{in}\\ &\qquad \mathit{typecase}_{neu}\ x\ \mathit{of}\ A_1 : (\langle A_1, m\rangle\ x), \ldots, A_n : (\langle A_n, m\rangle\ x).\end{aligned}$$

Diesen Block hätten wir also ebenfalls als Übersetzung von Nachrichten benutzen können.

5

Korrektheit des Übersetzers

In den vorangehenden Kapiteln haben wir uns mit Vererbung, der Semantik und der Übersetzung von Ausdrücken beschäftigt. Ein Programm mit Vererbung haben wir durch eine syntaktische Transformation in ein Programm ohne Vererbung überführt, dessen Deklarationsteil wir als Übersetzungszeitumgebung $g \in \mathit{CEnv}$ bezeichnet haben und dessen Hauptteil unverändert geblieben ist. Dies geschah in zwei Schritten:

Zunächst haben wir das Programm $p = d\,;\, e$ in ein Paar $\langle \gamma, e \rangle$ überführt, bestehend aus einem Vererbungsgraphen oder abstrakten Deklarationsteil $\gamma \in \mathit{InhGraph}$ und dem Hauptteil e . Bereits in diesem Schritt wurden Methoden als gewöhnliche Funktionale mit gewissen statisch semantischen Einschränkungen an die Benutzung des Bezeichners *self* repräsentiert. Sie werden an Methodenfunktionsnamen, bestehend aus dem Framenamen und dem Methodennamen, gebunden. Dieser erste Schritt ist unabhängig von der Vererbungsstrategie.

Im zweiten Schritt haben wir den Vererbungsgraphen γ durch Anwendung der Vererbungsstrategie in die Übersetzungszeitumgebung g überführt, in der genau diese Bindung eine Bedeutung erhalten hat: $\langle A, m \rangle$ ist in g an die Methode für m in A gebunden. *self* bezeichnet in deren Rumpf eine Instanz des Frames A .

Die Übersetzungszeitumgebung g , zusammen mit einem Ausdruck e , fassen wir als (abstraktes) Programm $\langle g, e \rangle$ ohne Vererbung auf. Wir zeigen in diesem Kapitel, daß wir die den Ausdrücken zugeordnete denotationelle Semantik und den Übersetzer von Ausdrücken in korrekter Weise auf Programme ausdehnen können.

Theorem (5.11) zeigt, daß Ausdrücke korrekt übersetzt werden, wenn die Umgebung, in der ihre Semantik bestimmt wird, mit g korrespondiert. Solche Umgebungen nennen wir g-konform. Wir zeigen dann, daß sowohl die Semantik als auch der Übersetzer auf natürliche Weise zunächst auf Abstraktionen und dann auf den Deklarationsteil ausgedehnt werden können.

Die Korrektheit der Übersetzung von Ausdrücken zeigen wir per Induktion über den Aufbau von Ausdrücken und gehen dabei davon aus, daß die Umgebungen, in denen die Semantik bestimmt wird, vernünftig, genauer *g-konform* sind. Das Vorgehen ist ähnlich zu beispielsweise dem in [Cli84], der die Korrektheit eines Scheme-Übersetzers zeigt. Zusätzlich zeigen wir jedoch, daß die durch die Semantik des Deklarationsteils g eines Programmes gegebene

Umgebung tatsächlich vernünftig, also g-konform, ist (Theorem 5.16). Sie ist wie üblich als der kleinste Fixpunkt eines Funktionals zwischen Umgebungen definiert. Wir zeigen dann in Theorem (5.19), daß dem übersetzten Deklarationsteil $\mathcal{C}_d[\![g]\!]$ die gleiche g-konforme Umgebung zuordnet wird wie g . Damit läßt sich die Korrektheit der Übersetzung von Ausdrücken auf Programme übertragen, deren Semantik gerade die Semantik des Hauptprogrammausdrucks in der durch die Semantik des Deklarationsteils gebildeten Umgebung ist (Theorem 5.22).

5.1 G-konforme Umgebungen

Definition 5.1

Sei $g \in \mathit{CEnv}$ und sei

$$\mathit{Support}_m(g) =_{\mathrm{Def}} \{\, A \in \mathit{Framename} \mid (g{\downarrow_1}\ \langle A, m\rangle) \neq \bot \,\}.$$

die Menge aller Framenamen, für die in g eine Methode zu m definiert ist, dann heißt $\rho \in \mathit{Env} = (\mathit{Ident} \longrightarrow \mathit{Value}) \times (\mathit{Framename} \longrightarrow \mathit{InstVar}^*)$ *g-konform*, falls für alle Methodenfunktionsnamen $\langle A, m\rangle \in \mathit{Ident}$

$$A \notin \mathit{Support}_m(g) \Longrightarrow (\rho{\downarrow_1}\ \langle A, m\rangle) = \bot \qquad \text{gilt.} \tag{5.1}$$

□

Da $A \in \mathit{Support}_m(g)$ genau dann gilt, wenn $(g{\downarrow_1}\ \langle A, m\rangle) \neq \bot$, ist (5.1) gleichbedeutend damit, daß $(\rho{\downarrow_1}\ \langle A, m\rangle) \neq \bot \Longrightarrow (g{\downarrow_1}\ \langle A, m\rangle) \neq \bot$ gilt, d.h. ρ, genauer die erste Komponente $\rho{\downarrow_1}$, ist nur für Methodenfunktionsnamen $\langle A, m\rangle$ definiert, für die es in g auch eine Methode gibt. G-konforme Umgebungen sind also vernünftig (*reasonable*) in dem Sinne, daß sie keine Methoden erfinden, die in dem Originalprogramm nicht vorhanden waren. Da Übersetzungszeitumgebungen Paare endlicher Abbildungen sind, ist $\mathit{Support}_m(g)$ eine höchstens endliche Menge. Sie ist leer, falls für m keine Methode in g definiert ist.

Bemerkung 5.2

Es ist nach Definition (5.1) klar, daß die total undefinierte Umgebung $\bot_{\mathrm{Env}}$ g-konform ist.

Die wesentliche Eigenschaft, die die g-Konformität gewährleistet, ist, daß die endliche generische Methodenauswahl über die Framenamen in $\mathit{Support}_m(g)$ ausreicht, um die richtige in der Umgebung assoziierte Methodenfunktion zu bestimmen. Wir zeigen dies in dem folgenden Lemma, das wir im Beweis von Lemma (5.10) benutzen werden, um die Korrektheit der Übersetzung von Nachrichten einzusehen.

Lemma 5.3

Sei $g \in CEnv$ *, sei* $m \in Message$ *und sei* $Support_m(g) = \{ A_1, \ldots, A_n \}$ *. Sei* $\rho \in Env$ *g-konform und* $f \in Framename \longrightarrow FunVal$ *die Funktion*

$$\begin{aligned} \lambda A.\, & A = A_1 \rightarrow (\rho\downarrow_1\ \langle A_1, m \rangle)\,|_{FunVal}, \\ & \ldots \\ & A = A_n \rightarrow (\rho\downarrow_1\ \langle A_n, m \rangle)\,|_{FunVal}, \\ & \lambda\nu.\lambda\kappa.wrong \quad . \end{aligned}$$

Dann gilt für alle $\nu \in Value, \kappa \in ECont$ *und* $A \in Framename$

$$f\ A\ \nu\ \kappa\ =\ (\rho\downarrow_1\ \langle A, m \rangle)\,|_{FunVal}\ \nu\ \kappa.$$

Beweis:

Ist $A \in Support_m(g)$, also $A = A_i$ für ein $i \in \{ 1, \ldots, n \}$, so ist

$$f\ A\ \nu\ \kappa\ =\ f\ A_i\ \nu\ \kappa\ =\ (\rho\downarrow_1\ \langle A_i, m \rangle)\,|_{FunVal}\ \nu\ \kappa\ =\ (\rho\downarrow_1\ \langle A, m \rangle)\,|_{FunVal}\ \nu\ \kappa\ .$$

Ist andererseits $A \notin Support_m(g)$, folgt aus der g-Konformität von ρ , daß $(\rho\downarrow_1\ \langle A, m \rangle) = \bot_{\text{Value}}$ und damit $(\rho\downarrow_1\ \langle A, m \rangle)\,|_{FunVal} = \bot_{\text{FunVal}}$ gilt. Außerdem ist $(f\ A) = \lambda\nu.\lambda\kappa.wrong$, und da *wrong* die total undefinierte Continuation $wrong = \lambda\sigma\,.\,\bot_{\text{Answer}} : (Value \times Store)$ ist, gilt also für alle $\sigma \in Store$

$$f\ A\ \nu\ \kappa\ \sigma\ =\ wrong\ \sigma\ =\ \bot_{\text{Answer}}\ =\ \bot_{\text{FunVal}}\ \nu\ \kappa\ \sigma\ =\ (\rho\downarrow_1\ \langle A, m \rangle)\,|_{FunVal}\ \nu\ \kappa\ \sigma\ .$$

qed.

Man beachte, daß für beliebige $f, g \in FunVal$ aus $f\ \nu\ =\ g\ \nu$ nicht unbedingt $f\ =\ g$ folgt, da in diesem *gelifteten* Bereich $\bot \neq \lambda\nu.\bot$ ist, obwohl beide Elemente wertegleich sind. Immerhin folgt aber dennoch $f\ \nu\ =\ g\ \nu$ für alle $\nu \in Value$, falls $f = g$ ist. Es ist klar, daß zwei explizit konstruierte Funktionen $\lambda\nu.\lambda\kappa.\sigma_1$ und $\lambda\nu.\lambda\kappa.\sigma_2$ auch in *FunVal* gleich sind, wenn $\sigma_1\ =\ \sigma_2$ gilt. Denn sie sind in $[\, Value \longrightarrow [\, ECont \longrightarrow Cont\,]\,]$ die gleichen Funktionen aufgrund der Extensionalität des zugrundeliegenden getypten λ-Kalküls. Es sei daran erinnert, daß wir, nur aus Gründen der Lesbarkeit, statt $\langle 0, \lambda\nu.\lambda\kappa.\sigma \rangle$ *in FunVal* wiederum $\lambda\nu.\lambda\kappa.\sigma$ schreiben und daß damit $\lambda\nu.\lambda\kappa.\sigma \neq \bot_{\text{FunVal}}$ gilt.

5.2 Korrektheit der Übersetzung von Ausdrücken

Wir werden nun zeigen, daß in g-konformen Umgebungen die Semantik eines Ausdruckes gleich der des vom Ausdrucksübersetzer $\mathcal{C}_e^g[\![\,\bullet\,]\!]$ erzeugten Ausdrucks ist, daß der Übersetzer also auf Ausdrücken in g-konformen Umgebungen korrekt ist. Für beliebige Umgebungen ist das nicht zu erwarten, denn die generische Methodenauswahl, die im übersetzten Programm durch *typecase* explizit gemacht wird, selektiert nur über die Framenamen

$\{A_1, \ldots, A_n\} = \mathit{Support}_m(g)$ und liefert einen Fehler, falls das Empfängerobjekt einen Typ $A \notin \mathit{Support}_m(g)$ hat. Es muß also dafür gesorgt sein, daß auch die ursprüngliche Nachricht in einem solchen Fall sicher einen Fehler hervorruft. Dies stellen wir gerade durch die g-Konformität der Umgebung sicher, denn, zur Erinnerung, die Semantik eines Ausdruckes $e.m$ war gerade die Anwendung der in der Umgebung mit dem Typ des Objektes e und dem Methodennamen m assoziierten Methodenfunktion $(\rho{\downarrow_1}\ \langle A, m\rangle)$ auf das Empfängerobjekt selbst.

Der Beweis läuft durch strukturelle Induktion über den syntaktischen Aufbau des Ausdrucks e . Eine Bemerkung zur Notation: Sowohl Umgebungen als auch Übersetzungszeitumgebungen sind Paare von Abbildungen,

$$g \in \mathit{CEnv} = (\mathit{Ident} \xrightarrow{\mathit{fin}} \mathit{Abstraction}) \times (\mathit{Framename} \xrightarrow{\mathit{fin}} \mathit{InstVar}^*) \qquad \mathit{und}$$

$$\rho \in \mathit{Env} = (\mathit{Ident} \longrightarrow \mathit{Value}) \times (\mathit{Framename} \longrightarrow \mathit{InstVar}^*)\,.$$

Wir werden im folgenden häufig die Projektionen auf die jeweilige Komponente weglassen, wenn diese aus dem Zusammenhang klar zu erkennen ist, d.h. $(\rho\ \langle A, m\rangle)$ meint $(\rho{\downarrow_1}\ \langle A, m\rangle)$, $(g\ A)$ meint $(g{\downarrow_2}\ A)$. Ebenfalls schreiben wir für $x \in \mathit{Variable}$ und $\nu \in \mathit{Value}$ einfach $\rho\,[x \leftarrow \nu]$ statt $\langle\, \rho{\downarrow_1}\ [x \leftarrow \nu]\,, \rho{\downarrow_2}\ \rangle$.

Lemma 5.4

Sei $g \in \mathit{CEnv}$ und sei $\rho \in \mathit{Env}$ g-konform. Dann ist für $x \in \mathit{Variable}$ und $\nu \in \mathit{Value}$ auch $\rho\,[x \leftarrow \nu]$ g-konform.

Der Beweis ist trivial, da sich $\rho\,[x \leftarrow \nu]$ für Methodenfunktionsnamen $\langle A, m\rangle$ von ρ nicht unterscheidet. Die folgenden Lemmata zeigen den Induktionsschluß des Beweises von Theorem (5.11).

Lemma 5.5

Sei $g \in \mathit{CEnv}$, sei $\rho \in \mathit{Env}$ g-konform. Sei ferner für alle $\kappa \in \mathit{ECont}$ und $\sigma \in \mathit{Store}$ $\boldsymbol{\mathcal{E}}\,[\![\, e_0 \,]\!]\,\rho\kappa\sigma = \boldsymbol{\mathcal{E}}\,[\![\, \boldsymbol{\mathcal{C}}_e^g\,[\![\, e_0 \,]\!]\,]\!]\,\rho\kappa\sigma$ und $\boldsymbol{\mathcal{E}}\,[\![\, e_1 \,]\!]\,\rho\kappa\sigma = \boldsymbol{\mathcal{E}}\,[\![\, \boldsymbol{\mathcal{C}}_e^g\,[\![\, e_1 \,]\!]\,]\!]\,\rho\kappa\sigma$. Dann gilt für alle $\kappa \in \mathit{ECont}$ und $\sigma \in \mathit{Store}$

$$\boldsymbol{\mathcal{E}}\,[\![\, e_0\,(e_1) \,]\!]\ \rho\,\kappa\,\sigma \;=\; \boldsymbol{\mathcal{E}}\,[\![\, \boldsymbol{\mathcal{C}}_e^g\,[\![\, e_0\,(e_1) \,]\!]\,]\!]\ \rho\,\kappa\,\sigma \qquad \mathit{und}$$

$$\boldsymbol{\mathcal{E}}\,[\![\, e_0\,() \,]\!]\ \rho\,\kappa\,\sigma \;=\; \boldsymbol{\mathcal{E}}\,[\![\, \boldsymbol{\mathcal{C}}_e^g\,[\![\, e_0\,() \,]\!]\,]\!]\ \rho\,\kappa\,\sigma.$$

Beweis:

Wir zeigen nur die erste Aussage. Die zweite Behauptung für die parameterlosen Applikationen läßt sich völlig analog nachrechnen. Für beliebige $\kappa \in \mathit{ECont}$ und $\sigma \in \mathit{Store}$ gilt:

$$\begin{aligned}
&\mathcal{E}[\![e_0\,(e_1)]\!]\,\rho\kappa\sigma \\
&\quad = \mathcal{E}[\![e_0]\!]\,\rho\{\ \lambda\nu_0\,.\,\nu_0\ \in\ \mathit{FunVal}\ \rightarrow\ \mathcal{E}[\![e_1]\!]\,\rho\{\lambda\nu_1\,.\,\nu_0\,|_{\mathit{FunVal}}\ \nu_1\ \kappa\},\ \mathit{wrong}\ \}\,\sigma
\end{aligned}$$

, nach Definition der Ausdruckssemantik $\mathcal{E}[\![\bullet]\!]$.

$$\begin{aligned}
&\quad = \mathcal{E}[\![e_0]\!]\,\rho\{\ \lambda\nu_0\,.\,\nu_0\ \in\ \mathit{FunVal}\ \rightarrow\ \mathcal{E}[\![\mathcal{C}_e^g[\![e_1]\!]]\!]\,\rho\{\lambda\nu_1\,.\,\nu_0\,|_{\mathit{FunVal}}\ \nu_1\ \kappa\},\ \mathit{wrong}\ \}\,\sigma
\end{aligned}$$

, nach Voraussetzung mit $\kappa_1 = \{\lambda\nu_1\,.\,\nu_0\,|_{\mathit{FunVal}}\ \nu_1\ \kappa\}$.

$$\begin{aligned}
&\quad = \mathcal{E}[\![\mathcal{C}_e^g[\![e_0]\!]]\!]\,\rho\{\ \lambda\nu_0\,.\,\nu_0\ \in\ \mathit{FunVal}\ \rightarrow\ \mathcal{E}[\![\mathcal{C}_e^g[\![e_1]\!]]\!]\,\rho\{\lambda\nu_1\,.\,\nu_0\,|_{\mathit{FunVal}}\ \nu_1\ \kappa\},\ \mathit{wrong}\ \}\,\sigma
\end{aligned}$$

, nach Voraussetzung mit $\kappa_0 = \{\lambda\nu_0\,.\,\nu_0\ \in\ \mathit{FunVal}\ \rightarrow\ \cdots\}$.

$$\mathcal{E}[\![\mathcal{C}_e^g[\![e_0]\!]\ (\mathcal{C}_e^g[\![e_1]\!])]\!]\,\rho\kappa\sigma$$

, nach Definition der Ausdruckssemantik.

$$\mathcal{E}[\![\mathcal{C}_e^g[\![e_0\,(e_1)]\!]]\!]\,\rho\kappa\sigma$$

, nach Definition des Übersetzers für Applikationen.

qed.

Lemma 5.6

Sei $g \in \mathit{CEnv}$, *sei* $\rho \in \mathit{Env}$ *g-konform. Sei ferner für alle* $\kappa \in \mathit{ECont}$ *und* $\sigma \in \mathit{Store}$ $\mathcal{E}[\![e_i]\!]\rho\kappa\sigma = \mathcal{E}[\![\mathcal{C}_e^g[\![e_i]\!]]\!]\rho\kappa\sigma$, $i = 0, \ldots, 2$. *Dann gilt für alle* $\kappa \in \mathit{ECont}$ *und* $\sigma \in \mathit{Store}$

$$\mathcal{E}[\![e_1\,;\,e_2]\!]\,\rho\kappa\sigma\ =\ \mathcal{E}[\![\mathcal{C}_e^g[\![e_1\,;\,e_2]\!]]\!]\,\rho\kappa\sigma\quad \mathit{und}$$

$$\mathcal{E}[\![\mathit{if}\ e_0\ \mathit{then}\ e_1\ \mathit{else}\ e_2]\!]\,\rho\kappa\sigma\ =\ \mathcal{E}[\![\mathcal{C}_e^g[\![\mathit{if}\ e_0\ \mathit{then}\ e_1\ \mathit{else}\ e_2]\!]]\!]\,\rho\kappa\sigma.$$

Da der Übersetzer auch hier nur rekursiv über die Teilausdrücke verteilt wird, läuft der Beweis völlig analog zu dem des vorigen Lemmas. Der nächste interessante Fall ist die Abstraktion:

Lemma 5.7

Sei $g \in \mathit{CEnv}$, *sei* $\rho \in \mathit{Env}$ *g-konform. Sei ferner für alle* $\kappa \in \mathit{ECont}$ *und* $\sigma \in \mathit{Store}$ $\mathcal{E}[\![e]\!]\rho\kappa\sigma = \mathcal{E}[\![\mathcal{C}_e^g[\![e]\!]]\!]\rho\kappa\sigma$. *Dann gilt für alle* $\kappa \in \mathit{ECont}$ *und* $\sigma \in \mathit{Store}$

$$\mathcal{E}[\![\lambda x\,.\,e]\!]\,\rho\kappa\sigma\ =\ \mathcal{E}[\![\mathcal{C}_e^g[\![\lambda x\,.\,e]\!]]\!]\,\rho\kappa\sigma\quad \mathit{und}$$

$$\mathcal{E}[\![\lambda()\,.\,e]\!]\,\rho\kappa\sigma\ =\ \mathcal{E}[\![\mathcal{C}_e^g[\![\lambda()\,.\,e]\!]]\!]\,\rho\kappa\sigma.$$

Beweis:

$$
\begin{aligned}
&\mathcal{E}\,[\![\,\lambda\,x\,.\,e\,]\!]\;\rho\,\kappa\,\sigma \\
&\quad = \quad \kappa\;(\lambda\nu'\,.\,\lambda\kappa'\,.\,\lambda\sigma'\,.\,\mathcal{E}\,[\![\,e\,]\!]\;\;\rho\,[x \leftarrow \nu']\;\kappa'\,\sigma'\;\;\mathit{in}\;\mathit{Value})\;\sigma
\end{aligned}
$$

, nach Definition der Ausdruckssemantik $\mathcal{E}\,[\![\,\bullet\,]\!]$.

$$
\begin{aligned}
&\quad = \quad \kappa\;(\lambda\nu'\,.\,\lambda\kappa'\,.\,\lambda\sigma'\,.\,\mathcal{E}\,[\![\,\mathcal{C}_e^g\,[\![\,e\,]\!]\,]\!]\;\;\rho\,[x \leftarrow \nu']\;\kappa'\,\sigma'\;\;\mathit{in}\;\mathit{Value})\;\sigma
\end{aligned}
$$

, da nach Lemma (5.4) mit ρ auch $\rho\,[x \leftarrow \nu']$ eine g-konforme Umgebung ist und damit nach Voraussetzung $\mathcal{E}\,[\![\,e\,]\!]\;\rho\,[x \leftarrow \nu']\;\kappa'\,\sigma' \;=\; \mathcal{E}\,[\![\,\mathcal{C}_e^g\,[\![\,e\,]\!]\,]\!]\;\rho\,[x \leftarrow \nu']\;\kappa'\,\sigma'$ gilt.

$$
\begin{aligned}
&\quad = \quad \mathcal{E}\,[\![\,\lambda\,x\,.\,\mathcal{C}_e^g\,[\![\,e\,]\!]\,]\!]\;\rho\,\kappa\,\sigma
\end{aligned}
$$

, wiederum nach Definition von $\mathcal{E}\,[\![\,\bullet\,]\!]$.

$$
\begin{aligned}
&\quad = \quad \mathcal{E}\,[\![\,\mathcal{C}_e^g\,[\![\,\lambda\,x\,.\,e\,]\!]\,]\!]\;\rho\,\kappa\,\sigma
\end{aligned}
$$

, nach Definition des Übersetzers.

Die zweite Behauptung folgt mit ρ anstelle von $\rho\,[x \leftarrow \nu']$ analog:

$$
\begin{aligned}
&\mathcal{E}\,[\![\,\lambda\,(\,)\,.\,e\,]\!]\;\rho\,\kappa\,\sigma \\
&\quad = \quad \kappa\;(\lambda\kappa'\,.\,\lambda\sigma'\,.\,\mathcal{E}\,[\![\,e\,]\!]\;\;\rho\,\kappa'\,\sigma'\;\;\mathit{in}\;\mathit{Value})\,\sigma \\
&\quad = \quad \kappa\;(\lambda\nu'\,.\,\lambda\kappa'\,.\,\lambda\sigma'\,.\,\mathcal{E}\,[\![\,\mathcal{C}_e^g\,[\![\,e\,]\!]\,]\!]\;\;\rho\,\kappa'\,\sigma'\;\;\mathit{in}\;\mathit{Value})\,\sigma \\
&\quad = \quad \mathcal{E}\,[\![\,\mathcal{C}_e^g\,[\![\,\lambda\,(\,)\,.\,e\,]\!]\,]\!]\;\rho\,\kappa\,\sigma
\end{aligned}
$$

qed.

Lemma 5.8

Sei $g \in \mathit{CEnv}$ *, sei* $\rho \in \mathit{Env}$ *g-konform. Sei ferner für alle* $\kappa \in \mathit{ECont}$ *und* $\sigma \in \mathit{Store}$ $\mathcal{E}\,[\![\,e\,]\!]\,\rho\kappa\sigma \;=\; \mathcal{E}\,[\![\,\mathcal{C}_e^g\,[\![\,e\,]\!]\,]\!]\,\rho\kappa\sigma$*. Dann gilt für alle* $\kappa \in \mathit{ECont}$ *und* $\sigma \in \mathit{Store}$

$$\mathcal{E}\,[\![\,\mathit{self}\,.\,y\,]\!]\;\rho\,\kappa\,\sigma \;=\; \mathcal{E}\,[\![\,\mathcal{C}_e^g\,[\![\,\mathit{self}\,.\,y\,]\!]\,]\!]\;\rho\,\kappa\,\sigma \quad \mathit{und}$$

$$\mathcal{E}\,[\![\,\mathit{self}\,.\,y\;:=\;e\,]\!]\;\rho\,\kappa\,\sigma \;=\; \mathcal{E}\,[\![\,\mathcal{C}_e^g\,[\![\,\mathit{self}\,.\,y\;:=\;e\,]\!]\,]\!]\;\rho\,\kappa\,\sigma\;.$$

Beweis:

Die erste Behauptung ist trivial, da $\mathcal{C}_e^g[\![\, self.y \,]\!] = self.y$ ist. Für den Beweis der zweiten Aussage rechnen wir nach:

$$
\begin{aligned}
&\mathcal{E}[\![\, self.y := e \,]\!]\,\rho\kappa\sigma \\
&\quad= \mathcal{E}[\![\, self \,]\!]\,\rho\{\ \lambda\nu_0.\nu_0 \in Object \to \mathcal{E}[\![\, e \,]\!]\,\rho\{\lambda\nu.\lambda\sigma.\kappa\,\nu\,\sigma[(\nu_0{\downarrow_2}\ y) \leftarrow \nu]\ \}, \\
&\qquad\qquad wrong\ \}\,\sigma \\
&\quad= \mathcal{E}[\![\, self \,]\!]\,\rho\{\ \lambda\nu_0.\nu_0 \in Object \to \mathcal{E}[\![\, \mathcal{C}_e^g[\![\, e \,]\!] \,]\!]\,\rho\{\lambda\nu.\lambda\sigma.\kappa\,\nu\,\sigma[(\nu_0{\downarrow_2}\ y) \leftarrow \nu]\ \}, \\
&\qquad\qquad wrong\ \}\,\sigma
\end{aligned}
$$

, nach Voraussetzung mit $\kappa_0 = \{\lambda\nu.\lambda\sigma.\kappa\,\nu\,\sigma[(\nu_0{\downarrow_2}\ y) \leftarrow \nu]\ \}$.

$$
\begin{aligned}
&= \mathcal{E}[\![\, self.y := \mathcal{C}_e^g[\![\, e \,]\!] \,]\!]\,\rho\kappa\sigma \\
&= \mathcal{E}[\![\, \mathcal{C}_e^g[\![\, self.y := e \,]\!] \,]\!]\,\rho\kappa\sigma
\end{aligned}
$$

qed.

Lemma 5.9

Sei $g \in CEnv$, sei $\rho \in Env$ g-konform. Sei ferner für alle $\kappa \in ECont$ und $\sigma \in Store$ $\mathcal{E}[\![\, e_i \,]\!]\,\rho\kappa\sigma = \mathcal{E}[\![\, \mathcal{C}_e^g[\![\, e_i \,]\!] \,]\!]\,\rho\kappa\sigma$, $i = 0, \ldots, n$. Dann gilt für alle $\kappa \in ECont$ und $\sigma \in Store$

$$
\begin{aligned}
&\mathcal{E}[\![\, typecase\ e\ of\ A_1 : e_1, \ \ldots\ , A_n : e_n \,]\!]\,\rho\kappa\sigma\ = \\
&\qquad \mathcal{E}[\![\, \mathcal{C}_e^g[\![\, typecase\ e\ of\ A_1 : e_1, \ \ldots\ , A_n : e_n \,]\!] \,]\!]\,\rho\kappa\sigma.
\end{aligned}
$$

Beweis:

$$
\begin{aligned}
&\mathcal{E}[\![\, typecase\ e\ of\ A_1 : e_1, \ \ldots\ , A_n : e_n \,]\!]\,\rho\kappa\sigma \\
&\quad= \mathcal{E}[\![\, e \,]\!]\,\rho\{\ \lambda\nu_0.\nu_0 \in Object \to \\
&\qquad\qquad \nu_0{\downarrow_1} = A_1 \to \mathcal{E}[\![\, e_1 \,]\!]\,\rho\{\lambda\nu_1.\nu_1\,|_{FunVal}\,\nu_0\,\kappa\}, \\
&\qquad\qquad\qquad\qquad \ldots \\
&\qquad\qquad \nu_0{\downarrow_1} = A_n \to \mathcal{E}[\![\, e_n \,]\!]\,\rho\{\lambda\nu_n.\nu_n\,|_{FunVal}\,\nu_0\,\kappa\}, \\
&\qquad\qquad\qquad\qquad wrong, \\
&\qquad\qquad wrong\ \}\,\sigma
\end{aligned}
$$

, nach Definition der Ausdruckssemantik. Wenden wir nun die Voraussetzung an, so erhalten wir $\mathcal{E}[\![\, e_i \,]\!]\,\rho\{\lambda\nu_i.\nu_i\,|_{FunVal}\,\nu_0\,\kappa\} = \mathcal{E}[\![\, \mathcal{C}_e^g[\![\, e_i \,]\!] \,]\!]\,\rho\{\lambda\nu_i.\nu_i\,|_{FunVal}\,\nu_0\,\kappa\}$, $i = 1, \ldots, n$, und damit

$$
\begin{aligned}
&= \mathcal{E}[\![e]\!]\,\rho\{\ \lambda\nu_0.\nu_0 \in \mathit{Object} \to \\
&\qquad\qquad \nu_0{\downarrow_1} = A_1 \to \mathcal{E}[\![\mathcal{C}_e^g[\![e_1]\!]]\!]\,\rho\{\lambda\nu_1.\nu_1\,|_{\mathit{FunVal}}\,\nu_0\,\kappa\}, \\
&\qquad\qquad\qquad\qquad \dots \\
&\qquad\qquad \nu_0{\downarrow_1} = A_n \to \mathcal{E}[\![\mathcal{C}_e^g[\![e_n]\!]]\!]\,\rho\{\lambda\nu_n.\nu_n\,|_{\mathit{FunVal}}\,\nu_0\,\kappa\}, \\
&\qquad\qquad\qquad\qquad \mathit{wrong}, \\
&\qquad\qquad \mathit{wrong}\ \}\,\sigma \\
&= \mathcal{E}[\![\mathcal{C}_e^g[\![e]\!]]\!]\,\rho\{\lambda\nu_0.\nu_0 \in \mathit{Object} \to \dots\}
\end{aligned}
$$

, nach Voraussetzung mit $\kappa_0 = \{\lambda\nu_0.\nu_0 \in \mathit{Object} \to \dots\}$.

$$
\begin{aligned}
&= \mathcal{E}[\![\mathit{typecase}\ \mathcal{C}_e^g[\![e]\!]\ \mathit{of}\ A_1 : \mathcal{C}_e^g[\![e_1]\!], \dots, A_n : \mathcal{C}_e^g[\![e_n]\!]]\!]\,\rho\kappa\sigma \\
&= \mathcal{E}[\![\mathcal{C}_e^g[\![\mathit{typecase}\ e\ \mathit{of}\ A_1 : e_1, \dots, A_n : e_n]\!]]\!]\,\rho\kappa\sigma
\end{aligned}
$$

qed.

Das letzte Lemma zeigt den wirklich interessanten Fall, die Übersetzung einer Nachricht in die endliche Methodenselektion über die Framenamen in $\mathit{Support}_m(g)$.

Lemma 5.10

Sei $g \in \mathit{CEnv}$, sei $\rho \in \mathit{Env}$ g-konform. Sei ferner für alle $\kappa \in \mathit{ECont}$ und $\sigma \in \mathit{Store}$ $\mathcal{E}[\![e]\!]\,\rho\kappa\sigma = \mathcal{E}[\![\mathcal{C}_e^g[\![e]\!]]\!]\,\rho\kappa\sigma$. Dann gilt für alle $\kappa \in \mathit{ECont}$ und $\sigma \in \mathit{Store}$

$$\mathcal{E}[\![e.m]\!]\,\rho\kappa\sigma = \mathcal{E}[\![\mathcal{C}_e^g[\![e.m]\!]]\!]\,\rho\kappa\sigma.$$

Beweis:

Wenden wir zunächst die Definitionen auf die linke und rechte Seite an, so erhalten wir

$$
\begin{aligned}
&\mathcal{E}[\![e.m]\!]\,\rho\kappa\sigma \\
&\quad = \mathcal{E}[\![e]\!]\,\rho\{\ \lambda\nu_0.\nu_0 \in \mathit{Object} \to (\rho{\downarrow_1}\ \langle\nu_0{\downarrow_1}, m\rangle)\,|_{\mathit{FunVal}}\,\nu_0\,\kappa, \\
&\qquad\qquad\qquad\qquad \mathit{wrong}\ \}\,\sigma
\end{aligned}
$$

und auf der rechten Seite

$$
\begin{aligned}
&\mathcal{E}[\![\mathcal{C}_e^g[\![e.m]\!]]\!]\,\rho\kappa\sigma \\
&\quad = \mathcal{E}[\![\mathit{typecase}\ \mathcal{C}_e^g[\![e]\!]\ \mathit{of}\ A_1 : \langle A_1, m\rangle, \dots, A_n : \langle A_n, m\rangle]\!]\,\rho\kappa\sigma
\end{aligned}
$$

, wobei $\{A_1, \dots, A_n\} = \mathit{Support}_m(g)$ die (endliche) Menge der Framenamen ist, für die in g eine Methode zu m definiert ist.

$$
\begin{array}{lll}
= & \mathcal{E}\,[\![\,\mathcal{C}_e^g\,[\![\,e\,]\!]\,]\!]\;\rho\{\;\lambda\nu_0\,.\,\nu_0 \;\in\; \mathit{Object}\to \\
& \qquad\qquad \nu_0{\downarrow_1} \;=\; A_1 \;\to\; \mathcal{E}\,[\![\,\langle A_1, m\,\rangle\,]\!]\;\rho\{\lambda\,\nu_1\,.\,\nu_1\;|_{\mathit{FunVal}}\;\nu_0\,\kappa\}\,, \\
& \qquad\qquad\qquad\qquad \cdots \\
& \qquad\qquad \nu_0{\downarrow_1} \;=\; A_n \;\to\; \mathcal{E}\,[\![\,\langle A_n, m\,\rangle\,]\!]\;\rho\{\lambda\,\nu_n\,.\,\nu_n\;|_{\mathit{FunVal}}\;\nu_0\,\kappa\}\,, \\
& \qquad\qquad\qquad\qquad \mathit{wrong}\,, \\
& \qquad\qquad \mathit{wrong}\;\}\;\sigma
\end{array}
$$

, nach Definition der Semantik von *typecase*. Die Voraussetzung für e liefert dann mit $\kappa_0 = \{\lambda\nu_0\,.\,\nu_0 \in \mathit{Object}\to\ \ldots\}$

$$
= \quad \mathcal{E}\,[\![\,e\,]\!]\;\rho\{\lambda\nu_0\,.\,\nu_0 \in \mathit{Object}\to\ \ldots\}\,\sigma
$$

und daraus ergibt sich, da für $i = 1,\ldots,n$ die Bedeutung von $\langle A_i, m\,\rangle$ gerade die Anwendung der Continuation $\{\lambda\,\nu_i\,.\,\nu_i\;\;|_{\mathit{FunVal}}\;\;\nu_0\,\kappa\}$ auf das ρ-Bild von $\langle A_i, m\,\rangle$ ist,

$$
\begin{array}{lll}
= & \mathcal{E}\,[\![\,e\,]\!]\;\rho\{\;\lambda\nu_0\,.\,\nu_0 \;\in\; \mathit{Object}\to \\
& \qquad\qquad \nu_0{\downarrow_1} \;=\; A_1 \;\to\; (\rho{\downarrow_1}\;\langle A_1, m\,\rangle)\;|_{\mathit{FunVal}}\;\nu_0\,\kappa\,, \\
& \qquad\qquad\qquad\qquad \cdots \\
& \qquad\qquad \nu_0{\downarrow_1} \;=\; A_n \;\to\; (\rho{\downarrow_1}\;\langle A_n, m\,\rangle)\;|_{\mathit{FunVal}}\;\nu_0\,\kappa\,, \\
& \qquad\qquad\qquad\qquad \mathit{wrong}\,, \\
& \qquad\quad \mathit{wrong}\;\}\;\sigma \quad .
\end{array}
$$

Vergleichen wir die beiden entstandenen Formeln, so bleibt zu zeigen:

$$
\begin{array}{lll}
(\rho{\downarrow_1}\;\langle\nu_0{\downarrow_1}\,, m\,\rangle)\;|_{\mathit{FunVal}}\;\nu_0\,\kappa \;=\; (\;\; \nu_0{\downarrow_1} \;=\; A_1 \;\to\; (\rho{\downarrow_1}\;\langle A_1, m\,\rangle)\;|_{\mathit{FunVal}}\;\nu_0\,\kappa\,, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \cdots \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \nu_0{\downarrow_1} \;=\; A_n \;\to\; (\rho{\downarrow_1}\;\langle A_n, m\,\rangle)\;|_{\mathit{FunVal}}\;\nu_0\,\kappa\,, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{wrong}\;)
\end{array}
$$

Das ist aber gerade die Aussage von Lemma (5.3), denn g ist als g-konform vorausgesetzt und es gilt $\{\,A_1,\ldots,A_n\,\} = \mathit{Support}_m(g)$.

qed.

Da der Übersetzer die elementaren Ausdrücke, d.h. Konstanten c, Variablen x, Methodenfunktionsnamen $\langle A, m\,\rangle$ und Instanziierungen *instance-of* A identisch übersetzt, haben wir mit den vorangehenden Lemmata die Korrektheit der Übersetzung von Ausdrücken gezeigt:

Theorem 5.11

Sei $g \in \mathit{CEnv}$, und sei $\rho \in \mathit{Env}$ g-konform. Dann gilt für alle Ausdrücke $e \in \mathit{Expr}$ und für alle $\kappa \in \mathit{ECont}$ und $\sigma \in \mathit{Store}$

$$
\mathcal{E}\,[\![\,e\,]\!]\;\;\rho\;\kappa\;\sigma \;=\; \mathcal{E}\,[\![\,\mathcal{C}_e^g\,[\![\,e\,]\!]\,]\!]\;\;\rho\;\kappa\;\sigma.
$$

Im folgenden zeigen wir, daß sich dieses Resultat auf die Deklarationssemantik von abstrakten Programmen verallgemeinern lassen. Abstrakte Deklarationsteile sind Paare endlicher

Abbildungen aus dem Bereich

$$CEnv = (Ident \xrightarrow{fin} Abstraction) \times (Framename \xrightarrow{fin} InstVar^*).$$

Bei unseren bisherigen Betrachtungen haben wir das Verhalten des semantischen Funktionals $\mathcal{E}[\![\bullet]\!]$ und auch des Übersetzers $\mathcal{C}_e^g[\![\bullet]\!]$ auf undefinierten syntaktischen Elementen außer acht gelassen. In $g \in CEnv$ kann die einem Identifikator $I \in Ident$ zugeordnete Abstraktion $(g\downarrow_1 I)$ durchaus undefiniert sein. Deshalb setzen wir im folgenden die Striktheit dieser Abbildungen auf dem syntaktischen Bereich *Expr* voraus. Gleiches gilt für die im nächsten Abschnitt (5.3) definierte Abstraktionssemantik $\mathcal{A}[\![\bullet]\!]$ auf dem Bereich *Abstraction*.

5.3 Abstraktionen

Speziell für die Ausdrücke $\lambda x\,.\,e$ und $\lambda()\,.\,e$ unserer Sprache sind wir nicht nur an der Ausdruckssemantik $\mathcal{E}[\![\lambda x\,.\,e]\!], \mathcal{E}[\![\lambda()\,.\,e]\!] : Env \longrightarrow ECont \longrightarrow Cont$ interessiert, sondern auch an ihrem Wert als Element des Bereichs

$$FVal = [Value \longrightarrow ECont \longrightarrow Cont]_\perp + [ECont \longrightarrow Cont]_\perp\ ,$$

der in den Bereich $Value = \ldots + FVal + \ldots$ eingebettet ist. Wir definieren deshalb eine *Semantik von Abstraktionen* wie folgt:

Definition 5.12

Auf dem syntaktischen Bereich *Abstraction* der Abstraktionen definieren wir das semantische Funktional

$$\mathcal{A}[\![\bullet]\!] : Abstraction \longrightarrow Env \longrightarrow Value$$

durch

$$\begin{array}{lll} \mathcal{A}[\![\lambda x\,.\,e]\!]\,\rho & =_{\mathrm{Def}} & \lambda\nu\,.\,\lambda\kappa\,.\,\lambda\sigma\,.\,\mathcal{E}[\![e]\!]\,\rho[x \leftarrow \nu]\,\kappa\,\sigma \ \ in\ Value \\ \mathcal{A}[\![\lambda()\,.\,e]\!]\,\rho & =_{\mathrm{Def}} & \lambda\kappa\,.\,\lambda\sigma\,.\,\mathcal{E}[\![e]\!]\,\rho\,\kappa\,\sigma \ \ in\ Value\ . \end{array}$$

□

Die Abstraktionssemantik eines λ-Ausdrucks ist also gerade die Funktion, die von der Ausdruckssemantik $\mathcal{E}[\![\bullet]\!]$ an die Continuation übergeben wird. Wir hätten die Semantik des Ausdrucks $\lambda x\,.\,e$ ebenfalls durch

$$\mathcal{E}[\![\lambda x\,.\,e]\!]\,\rho\kappa\ =_{\mathrm{Def}}\ \kappa\ (\mathcal{A}[\![\lambda x\,.\,e]\!]\,\rho)$$

definieren können. $\mathcal{A}[\![\bullet]\!]$ sei als strikt auf dem syntaktischen Bereich *Abstraction* angenommen, d.h. wir definieren zusätzlich $\mathcal{A}[\![\perp]\!] =_{\mathrm{Def}} \perp_{\mathrm{Value}}$. Das stellt sicher, daß $\mathcal{A}[\![\bullet]\!]$, angewendet auf einen undefinierten Eintrag in g , auch einen undefinierten Eintrag in der Umgebung $\rho \in Env$ erzeugt. (s.u.)

Wir zeigen nun, daß der Übersetzer die Ausdrücke $\lambda x\,.\,e$ und $\lambda()\,.\,e$ auch bezüglich der Abstraktionssemantik korrekt übersetzt:

Satz 5.13

Sei $g \in CEnv$ und sei $\rho \in Env$ g-konform. Dann gilt

$$\mathcal{A}[\![\lambda x\,.\,e]\!]\,\rho \;=\; \mathcal{A}[\![\mathcal{C}_e^g[\![\lambda x\,.\,e]\!]]\!]\,\rho \quad \textit{und}$$

$$\mathcal{A}[\![\lambda ()\,.\,e]\!]\,\rho \;=\; \mathcal{A}[\![\mathcal{C}_e^g[\![\lambda ()\,.\,e]\!]]\!]\,\rho.$$

Beweis:

Sei $\rho \in Env$ g-konform. Dann ist mit $x \in Var$ auch $\rho[x \leftarrow \nu]$ g-konform nach Lemma (5.4) und damit gilt

$$\begin{aligned}
\mathcal{A}[\![\lambda x\,.\,e]\!]\,\rho &= \lambda\nu'\,.\,\lambda\kappa'\,.\,\lambda\sigma'\,.\,\mathcal{E}[\![e]\!]\,\rho[x \leftarrow \nu']\,\kappa'\,\sigma' \;\textit{in Value}\\
&= \lambda\nu'\,.\,\lambda\kappa'\,.\,\lambda\sigma'\,.\,\mathcal{E}[\![\mathcal{C}_e^g[\![e]\!]]\!]\,\rho[x \leftarrow \nu']\,\kappa'\,\sigma' \;\textit{in Value}
\end{aligned}$$

,nach Theorem (5.11).

$$\begin{aligned}
&= \mathcal{A}[\![\lambda x.\mathcal{C}_e^g[\![e]\!]]\!]\,\rho\\
&= \mathcal{A}[\![\mathcal{C}_e^g[\![\lambda x\,.\,e]\!]]\!]\,\rho \quad .
\end{aligned}$$

Völlig analog gilt auch

$$\begin{aligned}
\mathcal{A}[\![\lambda ()\,.\,e]\!]\,\rho &= \lambda\kappa'\,.\,\lambda\sigma'\,.\,\mathcal{E}[\![e]\!]\,\rho\,\kappa'\,\sigma' \;\textit{in Value}\\
&= \lambda\kappa'\,.\,\lambda\sigma'\,.\,\mathcal{E}[\![\mathcal{C}_e^g[\![e]\!]]\!]\,\rho\,\kappa'\,\sigma' \;\textit{in Value}\\
&= \mathcal{A}[\![\lambda().\mathcal{C}_e^g[\![e]\!]]\!]\,\rho\\
&= \mathcal{A}[\![\mathcal{C}_e^g[\![\lambda ()\,.\,e]\!]]\!]\,\rho
\end{aligned}$$

qed.

5.4 Deklarationen

In dem nun folgenden Abschnitt wenden wir uns dem Deklarationsteil von Programmen zu. Zur Erinnerung: Als Deklarationsteil eines abstrakten Programmes bezeichnen wir die Übersetzungszeitumgebung $g \in CEnv$, die nach der Durchführung der Vererbung aus dem Vererbungsgraphen des ursprünglichen Programmes entstanden ist. Diese bindet zum einen Framenamen an die Liste ihrer Instanzvariablennamen. Zum anderen stellt die Bindung der Funktions- und Methodenfunktionsnamen an die zugehörigen Abstraktionen ein i.a. wechselseitig rekursives Gleichungssystem dar.

Semantisch ordnen wir einer Übersetzungszeitumgebung eine Umgebung zu. Sie wird, aufgrund der wechselseitigen Rekursion, als der kleinste Fixpunkt eines Funktionals zwischen Umgebungen definiert (Definition 5.15) und ordnet den Funktionsnamen die entsprechenden Funktionen zu. Letztere erhalten wir durch die komponentenweise Anwendung von $\mathcal{A}[\![\bullet]\!]$ (vgl. den vorherigen Abschnitt) auf die in g gebundenen Abstraktionen. Durch die

Anwendung des Übersetzers für Ausdrücke auf diese Abstraktionen erhalten wir einen übersetzten Deklarationsteil $\mathcal{C}_d[\![g]\!]$, dessen Semantik gleich der bereits g zugeordneten Umgebung ist. Diese Korrektheitsaussage zeigen wir in Theorem (5.19). Zudem zeigen wir in Theorem (5.16), daß die entstehenden Umgebungen g-konform sind. Diese zweite zentrale Aussage erlaubt, die Korrektheit des Ausdrucksübersetzers auf komplette Programme zu übertragen. Dies tun wir im nächsten Abschnitt.

Wir definieren nun zunächst den Deklarationsübersetzer $\mathcal{C}_d[\![\bullet]\!]$ für abstrakte Deklarationsteile. Die Übersetzung geschieht dadurch, daß der Ausdrucksübersetzer auf die an die Identifikatoren gebundenen Abstraktionen angewendet wird. Die zweite Komponente, die Bindung von Framenamen an Instanzvariablenlisten, lassen wir unverändert. Aufgrund der angenommenen Striktheit des Ausdrucksübersetzers bleibt der übersetzte Deklarationsteil genau an den Stellen definiert, an denen auch der ursprüngliche definiert gewesen ist.

Definition 5.14

Der Deklarationsübersetzer

$$\mathcal{C}_d[\![\bullet]\!] : CEnv \longrightarrow CEnv$$

ist für $g \in CEnv$ definiert durch

$$\mathcal{C}_d[\![g]\!] =_{\text{Def}} < \lambda I.\mathcal{C}_e^g[\![(g{\downarrow_1}\ \ I)]\!], g{\downarrow_2} >$$

□

$\mathcal{C}_e^g[\![\bullet]\!]$ überführt Abstraktionen in Abstraktionen und damit gilt $\mathcal{C}_d[\![g]\!] \in CEnv$. Es ist klar, daß mit einem wohlgeformten abstrakten Programm $\langle g, e \rangle$ auch $\langle \mathcal{C}_d[\![g]\!], \mathcal{C}_e^g[\![e]\!] \rangle$ wohlgeformt ist.

Wir haben bereits bemerkt, daß die erste Komponente eines $g \in CEnv$ einem wechselseitig rekursiven Gleichungssystem von Funktionsdefinitionen entspricht. g soll semantisch eine Umgebung

$$\rho \in Env = (Ident \longrightarrow Value) \times (Framename \longrightarrow InstVar^*)$$

zugeordnet werden. Es ist natürlich, ρ durch den kleinsten Fixpunkt eines Funktionals zu definieren, das Umgebungen auf Umgebungen abbildet. Wir werden im folgenden sehen, daß dieser Fixpunkt einem Deklarationsteil g bereits eine g-konforme Umgebung zuordnet.

Definition 5.15

Die Semantik $\mathcal{D}[\![\bullet]\!] : CEnv \longrightarrow Env$ ist der kleinste Fixpunkt der Abbildung

$$\hat{\mathcal{D}}[\![\bullet]\!] : CEnv \longrightarrow Env \longrightarrow Env,$$

die wie folgt definiert ist:

$$\hat{\mathcal{D}}[\![g]\!]\,\rho =_{\text{Def}} < \lambda I.\mathcal{A}[\![(g{\downarrow_1}\ \ I)]\!]\ \rho\ ,\ g{\downarrow_2} >$$

□

Theorem 5.16

Sei $g \in CEnv$ *. Dann ist* $\mathcal{D}[\![g]\!] \in Env$ *g-konform.*

Beweis:

Zu zeigen ist, daß $(\mathcal{D}[\![g]\!]\downarrow_1 \langle A, m \rangle) = \bot$ folgt, falls für $A \in Framename$ und $m \in Message$ $A \notin Support_m(g)$ gilt. Wir zeigen, daß für beliebige $\rho \in Env$ bereits $\hat{\mathcal{D}}[\![g]\!]\rho$ g-konform ist. Dies liegt in der Striktheit der Abstraktionssemantik $\mathcal{A}[\![\bullet]\!]$ begründet, denn für $\langle A, m \rangle \in Ident$ ist mit $A \notin Support_m(g)$, also $(g\downarrow_1 \langle A, m \rangle) = \bot$,

$$((\hat{\mathcal{D}}[\![g]\!]\rho)\downarrow_1 \langle A, m \rangle) = \mathcal{A}[\![(g\downarrow_1 \langle A, m \rangle)]\!]\rho = \mathcal{A}[\![\bot]\!]\rho = \bot.$$

Da nun $\mathcal{D}[\![g]\!]$ Fixpunkt von $\hat{\mathcal{D}}[\![g]\!]$ ist, gilt speziell für $\rho = \mathcal{D}[\![g]\!]$

$$(\mathcal{D}[\![g]\!]\downarrow_1 \langle A, m \rangle) = ((\hat{\mathcal{D}}[\![g]\!](\mathcal{D}[\![g]\!]))\downarrow_1 \langle A, m \rangle)$$

und damit ist $(\mathcal{D}[\![g]\!]\downarrow_1 \langle A, m \rangle) = \bot$, was zu zeigen war. *qed.*

Es ist damit klar, daß $\mathcal{D}[\![\mathcal{C}_d[\![g]\!]]\!]$ bezüglich $g' = \mathcal{C}_d[\![g]\!]$ g'-konform ist. Daß $\mathcal{D}[\![\mathcal{C}_d[\![g]\!]]\!]$ auch g-konform ist, zeigt das folgende Theorem, das rein technischer Natur ist. Wir benötigen seine Aussage, um im Beweis von Theorem (5.19) den Standardschluß der Fixpunkttheorie anwenden zu können, daß zwei kleinste Fixpunkte bereits dann gleich sind, wenn der eine ein Fixpunkt des anderen Funktionals ist und umgekehrt [Sto77].

Theorem 5.17

Sei $g \in CEnv$ *. Dann ist* $\mathcal{D}[\![\mathcal{C}_d[\![g]\!]]\!] \in Env$ *g-konform.*

Beweis:

Zu zeigen ist jetzt, daß $(\mathcal{D}[\![\mathcal{C}_d[\![g]\!]]\!]\downarrow_1 \langle A, m \rangle) = \bot$ ist, falls $A \notin Support_m(g)$ für alle $m \in Message$ gilt. Wir schließen genauso wie im vorigen Beweis: Für $A \notin Support_m(g)$, also $(g\downarrow_1 \langle A, m \rangle) = \bot$, gilt

$$\begin{aligned}((\hat{\mathcal{D}}[\![\mathcal{C}_d[\![g]\!]]\!]\rho)\downarrow_1 \langle A, m \rangle) &= \mathcal{A}[\![\mathcal{C}_d[\![g]\!]\downarrow_1 \langle A, m \rangle]\!]\ \rho \\ &= \mathcal{A}[\![\mathcal{C}_e^g[\![g\downarrow_1 \langle A, m \rangle]\!]]\!]\ \rho \\ &= \mathcal{A}[\![\mathcal{C}_e^g[\![\bot]\!]]\!]\ \rho = \mathcal{A}[\![\bot]\!]\ \rho = \bot,\end{aligned}$$

für beliebige Umgebungen $\rho \in Env$, was aus der Striktheit von $\mathcal{A}[\![\bullet]\!]$ und $\mathcal{C}_e^g[\![\bullet]\!]$ sofort folgt. Außerdem gilt

$$(\mathcal{D}[\![\mathcal{C}_d[\![g]\!]]\!]\downarrow_1 \langle A, m \rangle) = ((\hat{\mathcal{D}}[\![\mathcal{C}_d[\![g]\!]]\!]\ (\mathcal{D}[\![\mathcal{C}_d[\![g]\!]]\!]))\downarrow_1 \langle A, m \rangle),$$

da $\mathcal{D}[\![\mathcal{C}_d[\![g]\!]]\!]$ Fixpunkt von $\hat{\mathcal{D}}[\![\mathcal{C}_d[\![g]\!]]\!]$ ist. Damit ist $(\mathcal{D}[\![\mathcal{C}_d[\![g]\!]]\!]\downarrow_1 \langle A, m \rangle) = \bot$, was zu zeigen war.

qed.

Wir zeigen nun, daß in jedem einzelnen Schritt der Fixpunktiteration durch das Funktional $\hat{\mathcal{D}}[\![\bullet]\!]$ dem Deklarationsteil g und dem übersetzten Deklarationsteil $\mathcal{C}_d[\![g]\!]$ in g-konformen Umgebungen die gleiche neue Umgebung zugeordnet wird. Die Beweise der beiden vorigen Theoreme zeigen, daß diese dann auch wieder g-konform ist.

Theorem 5.18

Sei $g \in CEnv$ und $\rho \in Env$ sei g-konform. Dann gilt

$$\hat{\mathcal{D}}[\![g]\!]\ \rho = \hat{\mathcal{D}}[\![\mathcal{C}_d[\![g]\!]]\!]\ \rho.$$

Beweis:

Nach Definition gilt

$$\hat{\mathcal{D}}[\![g]\!]\rho =< \lambda I.\mathcal{A}[\![(g{\downarrow_1}\ I)]\!]\rho, g{\downarrow_2} > \quad \text{und}$$

$$\hat{\mathcal{D}}[\![\mathcal{C}_d[\![g]\!]]\!]\rho =< \lambda I.\mathcal{A}[\![(\mathcal{C}_d[\![g]\!]{\downarrow_1}\ I)]\!]\rho, \mathcal{C}_d[\![g]\!]{\downarrow_2} > .$$

Ebenfalls wissen wir nach Definition des Deklarations-Übersetzers, daß die zweite Komponente von g gleich bleibt, d.h. $\mathcal{C}_d[\![g]\!]{\downarrow_2} = g{\downarrow_2}$. Wir haben also zu zeigen, daß für jeden Identifikator $I \in Ident$

$$\mathcal{A}[\![(\mathcal{C}_d[\![g]\!]{\downarrow_1}\ I)]\!]\rho = \mathcal{A}[\![(g{\downarrow_1}\ I)]\!]\rho \quad \text{gilt.}$$

Nach Definition von $\mathcal{C}_d[\![\bullet]\!]$ ist

$$\mathcal{A}[\![(\mathcal{C}_d[\![g]\!]{\downarrow_1}\ I)]\!]\rho = \mathcal{A}[\![\mathcal{C}_e^g[\![(g{\downarrow_1}\ I)]\!]]\!]\rho.$$

Im Fall $(g{\downarrow_1}\ I) = \perp$ gilt

$$\mathcal{A}[\![\mathcal{C}_e^g[\![(g{\downarrow_1}\ I)]\!]]\!]\rho\ =\ \perp\ =\ \mathcal{A}[\![(g{\downarrow_1}\ I)]\!]\rho$$

wegen der Striktheit von $\mathcal{C}_e^g[\![\bullet]\!]$. Ist andererseits $(g{\downarrow_1}\ I)$ eine definierte Abstraktion $\lambda x\,.\,e$ oder $\lambda()\,.\,e$, wissen wir aus Satz (5.13)

$$\mathcal{A}[\![\mathcal{C}_e^g[\![\lambda x\,.\,e]\!]]\!]\rho = \mathcal{A}[\![\lambda x\,.\,e]\!]\rho \quad \text{bzw.}$$

$$\mathcal{A}[\![\mathcal{C}_e^g[\![\lambda()\,.\,e]\!]]\!]\rho = \mathcal{A}[\![\lambda()\,.\,e]\!]\rho,$$

denn ρ haben wir als g-konform vorausgesetzt. In beiden Fällen gilt also die in der Behauptung geforderte Gleichheit. *qed.*

Mit diesem Satz und der G-Konformität der beiden Umgebungen $\mathcal{D}[\![g]\!]$ und $\mathcal{D}[\![\mathcal{C}_d[\![g]\!]]\!]$ können wir nun endlich die Gleichheit dieser beiden Umgebungen, d.h. die Korrektheit des Deklarationsübersetzers zeigen. Der Beweis benutzt den folgenden Standardschluß der Fixpunkttheorie: Zwei kleinste Fixpunkte *fix* H_1 und *fix* H_2 zweier Funktionale H_1 und H_2 sind gleich, falls der eine ebenfalls ein Fixpunkt des jeweils anderen Funktionals ist und umgekehrt.

Theorem 5.19

Sei $g \in CEnv$ *. Dann ist*

$$\mathcal{D}[\![g]\!] = \mathcal{D}[\![\mathcal{C}_d[\![g]\!]]\!].$$

Beweis:

Da $\mathcal{D}[\![g]\!] = fix\ \hat{\mathcal{D}}[\![g]\!]$ und $\mathcal{D}[\![\mathcal{C}_d[\![g]\!]]\!] = fix\ \hat{\mathcal{D}}[\![\mathcal{C}_d[\![g]\!]]\!]$ ist, genügt es, zu zeigen, daß $\mathcal{D}[\![g]\!]$ Fixpunkt von $\hat{\mathcal{D}}[\![\mathcal{C}_d[\![g]\!]]\!]$ ist und umgekehrt. Denn dann gilt

$$\begin{array}{lll} \mathcal{D}[\![g]\!] & \sqsupseteq & fix\ \hat{\mathcal{D}}[\![\mathcal{C}_d[\![g]\!]]\!] = \mathcal{D}[\![\mathcal{C}_d[\![g]\!]]\!] \quad \text{und} \\ \mathcal{D}[\![\mathcal{C}_d[\![g]\!]]\!] & \sqsupseteq & fix\ \hat{\mathcal{D}}[\![g]\!] = \mathcal{D}[\![g]\!], \end{array}$$

da *fix* den kleinsten Fixpunkt definiert. Im folgenden benutzen wir die G-Konformität von $\mathcal{D}[\![g]\!]$ und $\mathcal{D}[\![\mathcal{C}_d[\![g]\!]]\!]$, die wir in Theorem (5.16) und Theorem (5.17) gezeigt haben. Nach Satz (5.18) haben wir

$$\hat{\mathcal{D}}[\![\mathcal{C}_d[\![g]\!]]\!]\ (\mathcal{D}[\![g]\!]) = \hat{\mathcal{D}}[\![g]\!]\ (\mathcal{D}[\![g]\!]) = \mathcal{D}[\![g]\!],$$

da $\mathcal{D}[\![g]\!]$ g-konform ist, und weil $\mathcal{D}[\![g]\!] = fix\ \hat{\mathcal{D}}[\![g]\!]$ ist, und

$$\hat{\mathcal{D}}[\![g]\!]\ (\mathcal{D}[\![\mathcal{C}_d[\![g]\!]]\!]) = \hat{\mathcal{D}}[\![\mathcal{C}_d[\![g]\!]]\!]\ (\mathcal{D}[\![\mathcal{C}_d[\![g]\!]]\!]) = \mathcal{D}[\![\mathcal{C}_d[\![g]\!]]\!],$$

da $\mathcal{D}[\![\mathcal{C}_d[\![g]\!]]\!]$ g-konform ist, und weil $\mathcal{D}[\![\mathcal{C}_d[\![g]\!]]\!] = fix\ \hat{\mathcal{D}}[\![\mathcal{C}_d[\![g]\!]]\!]$ ist.

qed.

5.5 Programme

Kommen wir nun zur Semantik und Übersetzung von Programmen. Programme, genauer abstrakte Programme ohne Vererbung, haben wir als Paare

$$\langle g, e \rangle \in AbsProg =_{\text{Def}} CEnv \times Expr$$

definiert. Programme werden übersetzt, indem der Deklarationsübersetzer auf $g \in CEnv$ und der Ausdrucksübersetzer auf $e \in Expr$ angewendet werden:

Definition 5.20

Der Übersetzer $\mathcal{C}[\![\bullet]\!] : AbsProg \longrightarrow AbsProg$ von Programmen ist für $p = \langle g, e \rangle$ definiert durch

$$\mathcal{C}[\![\langle g, e \rangle]\!] =_{\text{Def}} \langle \mathcal{C}_d[\![g]\!], \mathcal{C}_e^g[\![e]\!] \rangle.$$

□

Die Semantik eines Programmes ist auf natürliche Weise durch die Anwendung der Semantik des Hauptprogrammausdrucks auf die Umgebung definiert, die wir durch die Deklarationssemantik aus dem Deklarationsteil des Programmes gewinnen. Diesen *Code* wenden wir auf eine Start-Continuation und einen initialen Speicher an, um die Programmantwort zu erhalten.

Definition 5.21

Die Semantik von Programmen definieren wir durch das semantische Funktional

$$\boldsymbol{\mathcal{P}}[\![\bullet]\!] : AbsProg \longrightarrow Answer \;=\; AbsProg \longrightarrow (Value \times Store).$$

Sei $p = \langle g, e \rangle \in AbsProg$. Dann ist $\boldsymbol{\mathcal{P}}[\![\bullet]\!]$ durch

$$\boldsymbol{\mathcal{P}}[\![p]\!] \;= \boldsymbol{\mathcal{P}}[\![\langle g, e \rangle]\!] =_{\text{Def}} \boldsymbol{\mathcal{E}}[\![e]\!](\boldsymbol{\mathcal{D}}[\![g]\!])\kappa_{Start}\sigma_{Start}$$

definiert, wobei $\kappa_{Start} =_{\text{Def}} \lambda\nu.\lambda\sigma.\langle \nu, \sigma \rangle \in ECont$ die Start-Continuation und $\sigma_{Start} \in Store$ der initiale Speicher sind. □

Eine einfache Anwendung der Resultate aus den vorherigen Abschnitten liefert nun das Hauptresultat dieses Kapitels, die Korrektheit des Übersetzers. Wir zeigen, daß die Programmsemantik des Programmes $\langle g, e \rangle$ gleich der Programmsemantik des übersetzten Programmes $\boldsymbol{\mathcal{C}}[\![\langle g, e \rangle]\!] = \langle \boldsymbol{\mathcal{C}}_d[\![g]\!], \boldsymbol{\mathcal{C}}_e^g[\![e]\!] \rangle$ ist.

Theorem 5.22

Sei $p = \langle g, e \rangle \in AbsProg$. Dann gilt

$$\boldsymbol{\mathcal{P}}[\![p]\!] \;= \boldsymbol{\mathcal{P}}[\![\boldsymbol{\mathcal{C}}[\![p]\!]]\!].$$

Beweis:

$$\begin{aligned}
\boldsymbol{\mathcal{P}}[\![p]\!] \;= \boldsymbol{\mathcal{P}}[\![\langle g, e \rangle]\!] \;&=\; \boldsymbol{\mathcal{E}}[\![e]\!](\boldsymbol{\mathcal{D}}[\![g]\!])\kappa_{Start}\sigma_{Start} \\
&=\; \boldsymbol{\mathcal{E}}[\![\boldsymbol{\mathcal{C}}_e^g[\![e]\!]]\!](\boldsymbol{\mathcal{D}}[\![\boldsymbol{\mathcal{C}}_d[\![g]\!]]\!])\kappa_{Start}\sigma_{Start} \\
&=\; \boldsymbol{\mathcal{P}}[\![\langle \boldsymbol{\mathcal{C}}_d[\![g]\!], \boldsymbol{\mathcal{C}}_e^g[\![e]\!] \rangle]\!] \\
&=\; \boldsymbol{\mathcal{P}}[\![\boldsymbol{\mathcal{C}}[\![p]\!]]\!]
\end{aligned}$$

Nach Theorem (5.19) ist $\boldsymbol{\mathcal{D}}[\![g]\!] = \boldsymbol{\mathcal{D}}[\![\boldsymbol{\mathcal{C}}_d[\![g]\!]]\!]$, nach Theorem (5.16) ist $\boldsymbol{\mathcal{D}}[\![g]\!]$ und damit natürlich auch $\boldsymbol{\mathcal{D}}[\![\boldsymbol{\mathcal{C}}_d[\![g]\!]]\!]$ g-konform und nach Theorem (5.11) gilt für beliebige g-konforme Umgebungen $\rho \in Env$ und beliebige $\kappa \in ECont, \sigma \in Store$

$$\boldsymbol{\mathcal{E}}[\![e]\!]\,\rho\kappa\sigma \;=\; \boldsymbol{\mathcal{E}}[\![\boldsymbol{\mathcal{C}}_e^g[\![e]\!]]\!]\,\rho\kappa\sigma.$$

qed.

Wir wollen dieses Kapitel abschließen, indem wir nach einer Bemerkung mehr technischer Natur noch einmal das Vorgehen rekapitulieren, das uns erlaubt hat, zu der objektorientierten Sprache mit multipler Vererbung, deren Syntax wir in Kapitel 2 definiert haben, eine

mathematisch exakte Semantik und einen Übersetzer in eine Scheme-ähnliche Teilsprache zu definieren und schließlich die Korrektheit des Übersetzers zu zeigen.

Zunächst jedoch zu einer eher technischen Bemerkung: Die G-Konformität von Umgebungen ist eine Einschränkung an erlaubte Umgebungen, allerdings nur bezüglich der Bindungen an Methodenfunktionsnamen $\langle A, m \rangle$. Wir können die Umgebung, in der die Semantik des Hauptprogrammausdruckes bestimmt wird, um schließlich des Semantik des Programmes zu erhalten, auf die folgende Weise um Bindungen für die freien gewöhnlichen Variablen des Programmes ergänzen:

Bemerkung 5.23

Sei $\langle g, e \rangle \in \mathit{CEnv} \times \mathit{Expr}$ und sei $\{ x_1, \ldots, x_n \} \subseteq \mathit{Var}$ die Menge der frei in dem Programm vorkommenden gewöhnlichen Variablen. Sei $\bar{\rho} =_{\text{Def}} \rho_\perp [x_1 \leftarrow \nu_1] \cdots [x_n \leftarrow \nu_n]$ die Umgebung, die gerade den x_i entsprechende $\nu_i \in \mathit{Value}$ zuordnet. Dann ist mit $\rho \in \mathit{Env}$ auch die Umgebung

$$\bar{\rho}; \rho \in \mathit{Env} : I \mapsto \begin{cases} (\bar{\rho}\ I) & \text{, falls } (\bar{\rho}\ I) \neq \perp_{\text{Value}} \\ (\rho\ I) & \text{, sonst} \end{cases}$$

g-konform, denn $\bar{\rho}; \rho$ unterscheidet sich von ρ für Methodenfunktionsnamen nicht. Damit läßt sich die Startumgebung des Programmes durch

$$\hat{\boldsymbol{\mathcal{D}}}' [\![g]\!]\ \rho =_{\text{Def}} < \lambda I. \boldsymbol{\mathcal{A}} [\![(g{\downarrow_1}\ I)]\!]\ \bar{\rho}; \rho\ ,\ g{\downarrow_2} > \qquad \text{und}$$

$$\boldsymbol{\mathcal{D}}' [\![g]\!] =_{\text{Def}} \mathit{fix}\ (\hat{\boldsymbol{\mathcal{D}}}' [\![g]\!])$$

um die Bindungen für die freien gewöhnlichen Variablen $\{ x_1, \ldots, x_n \}$ des Programmes ergänzen, ohne daß die Korrektheit der Übersetzung verloren geht.

Diese Bemerkung rechtfertigt, daß wir im Kapitel 3 freie gewöhnliche Variablen in wohlgeformten Programmen zugelassen haben.

Fassen wir nun unser Vorgehen nocheinmal zusammen: Zunächst haben wir unser originales Programm, das in der Form $p = d; e \in \mathit{Program}$ in abstrakter Syntax vorlag, durch Anwenden der im Kapitel 2 definierten Funktion $\mathbf{IG}\ [\![\bullet]\!]$ auf den Deklarationsteil d in eine äquivalente andere syntaktische Form gebracht. Daraus ist also $\langle \gamma, e \rangle = \langle \mathbf{IG}\ [\![d]\!]\ \gamma_\perp, e \rangle \in \mathit{InhGraph} \times \mathit{Expr}$ entstanden. Dann haben wir in Kapitel 3 durch die Anwendung einer geeigneten Vererbungsstrategie den Deklarationsteil $\gamma \in \mathit{InhGraph}$ in eine Übersetzungszeitumgebung $g \in \mathit{CEnv}$ transformiert. Diese Transformation ist nicht mehr eine äquivalente Umformung; verschiedene Vererbungsstrategien liefern durchaus verschiedene Ergebnisse. Insgesamt ist aus unserem Programm damit ein Paar $\langle g, e \rangle \in \mathit{CEnv} \times \mathit{Expr}$ geworden. $\langle g, e \rangle$ repräsentiert immer noch ein Programm unserer Sprache, nun allerdings eines ohne Vererbung.

Den in diesem Programm vorkommenden Ausdrücken, insbesondere also auch dem Hauptprogrammausdruck e, haben wir im Kapitel 4 eine denotationelle Continuation-Semantik $\boldsymbol{\mathcal{E}}\ [\![\bullet]\!]$ zugeordnet. Ebenfalls in Kapitel 4 haben wir eine Übersetzungsfunktion $\boldsymbol{\mathcal{C}}_e^g\ [\![\bullet]\!]$ für Ausdrücke definiert, die jede Nachricht in eine entsprechende *endliche* generische Methodenfunktionsauswahl übersetzt. Dabei haben wir die Kenntnis aller Methoden aller Frames nach Durchführung der Vererbung ausgenutzt.

In diesem Kapitel nun haben wir gezeigt, daß sowohl die denotationelle Semantik als auch die Übersetzungsfunktion auf natürliche Weise zunächst auf den Deklarationsteil g und dann auf das gesamte Programm $\langle g, e \rangle$ ausgedehnt werden können. Es ist nun also ein übersetztes Programm $\langle \boldsymbol{\mathcal{C}}_d [\![g]\!], \boldsymbol{\mathcal{C}}_e^g [\![e]\!] \rangle$ entstanden, dem wir, wie auch dem Programm $\langle g, e \rangle$, mit dem semantischen Funktional $\boldsymbol{\mathcal{P}} [\![\bullet]\!]$ eine Semantik zugeordnet haben. Wir haben dann die Gleichheit der Semantik $\boldsymbol{\mathcal{P}} [\![\langle g, e \rangle]\!]$ mit der Semantik $\boldsymbol{\mathcal{P}} [\![\langle \boldsymbol{\mathcal{C}}_d [\![g]\!], \boldsymbol{\mathcal{C}}_e^g [\![e]\!] \rangle]\!]$ des übersetzten Programmes und damit die Korrektheit der Übersetzung von Programmen gezeigt.

Auf diese Weise ist aus dem ursprünglichen Programm $p = d\,;\, e$ schließlich ein semantisch äquivalentes abstraktes Programm entstanden, das wir konkret nunmehr als Programm einer Scheme-ähnlichen Teilsprache auffassen können, indem wir auch die Bindungen der Methodenfunktionsnamen an die Methodenfunktionen wie Funktionsdefinitionen lesen. Die konkret benutzte Vererbungsstrategie bestimmt sowohl die Semantik des ursprünglichen Programmes als auch das Ergebnis der Übersetzung.

6

Schlußbemerkungen und Ausblick

Wir haben einen Ansatz zur Beschreibung der Semantik objektorientierter Programmiersprachen vorgeschlagen und ausgearbeitet, der es erlaubt, die Bedeutung der Vererbung in objektorientierten Programmen getrennt von der dynamischen Semantik des objektorientierten Kerns zu definieren. Dieses Vorgehen hat gegenüber den rein denotationellen Semantiken, die für Sprachen mit einfacher Vererbung existieren, den Vorteil, daß unterschiedliche mathematische Methoden zur Definition der Vererbung und der dynamischen Semantik benutzt werden können. Unter Zuhilfenahme einer Vererbungsstrategie, die auf dem Vererbungsgraphen des Programmes beruht, haben wir so die Semantik objektorientierter Programme auch mit multipler Vererbung mathematisch exakt definieren können. Der Ansatz ist offen gegenüber der Modifikation der Vererbungsstrategie und läßt so Experimente mit unterschiedlichen Strategien insbesondere in objektorientierten Wissensrepräsentationssprachen zu.

Zur Definition des objektorientierten Kerns der Sprache haben wir eine denotationelle Continuation-Semantik benutzt. Dieses Vorgehen ist bei der Definition der Semantik imperativer Sprachen mit höheren Funktionen und dem Konzept von Variablen und Zuweisungen, wie beispielsweise in Scheme, heute üblich und basiert auf den Arbeiten von D. Scott, Ch. Strachey und J. Stoy zur Theorie der denotationellen Semantik von Programmiersprachen. Insofern fügt sich unser Ansatz in die heute benutzte Technik zur Beschreibung der Semantik Lisp-artiger Programmiersprachen ein.

Zur Beschreibung der Semantik der Vererbung haben wir einen sehr allgemein gefaßten Begriff von Vererbungsstrategien benutzt, um die heute noch nötigen Experimente mit unterschiedlichen Vererbungsmechanismen insbesondere im Zusammenhang mit der objektorientierten Wissensrepräsentation zu ermöglichen, ohne auf die mathematisch exakte Definierbarkeit der Semantik verzichten zu müssen. Tatsächlich lassen sich die in den heute vorliegenden Sprachimplementationen benutzten Vererbungsstrategien im Rahmen der hier vorgestellten Theorie beschreiben. Wir benutzen die Begriffe Vererbungsrelation, Vererbungsgraph und auch Vererbungsstrategie in einem Sinne, der ebenfalls im Umfeld Lisp-artiger objektorientierter Sprachen mit multipler Vererbung durchaus üblich ist. Wir setzen allerdings voraus, daß das gesamte Programm und damit alle zur Bedeutung der Vererbung beitragenden Programmbestandteile statisch bekannt sind.

Wir haben zudem einen Übersetzer angegeben, der ein Programm nach Durchführung

der Vererbung in ein Programm einer Scheme-ähnlichen Teilsprache übersetzt und haben die Korrektheit des Übersetzers und der damit vorhandenen Implementation nachgewiesen, wobei wir allerdings eine korrekte Implementation der Zielsprache unterstellen.

Die Arbeit ist im Rahmen eines Projektes zur „effizienten Implementierung wissensbasierter Expertensysteme“ entstanden, in dem die *Kompilation von Wissensbasen* als eine neuartige Technik zur effizienten Implementierung und Portierung wissensbasierter Expertensysteme entwickelt worden ist. Konkret ist im Rahmen dieses Projektes ein Übersetzer für die hybride Wissensrepräsentationssprache BABYLON entstanden, der es erlaubt, in BABYLON entwickelte Expertensystemanwendungen zunächst nach Common Lisp und dann weiter nach C zu kompilieren und sie so mit erheblichem Effizienzgewinn sogar auf kleine Anwendungsrechner zu portieren.

Die in der vorliegenden Arbeit beschriebene Technik zur Übersetzung objektorientierter Sprachen ist im Rahmen des genannten Projektes speziell für BABYLON implementiert worden. BABYLON enthält einen mächtigen objektorientierten Wissensrepräsentationsformalismus.

Die Motivation, insbesondere den Vererbungsmechanismus möglichst flexibel zu halten, hat im Zusammenhang mit der Entwicklung der zur Zeit aktuellen objektorientierten Erweiterung CLOS von Lisp zur Definition eines sogenannten *Metaobjekt-Protokolls* geführt [KdB91]. Auch die Definition von EuLisp [Pad92] enthält die Spezifikation eines Metaobjekt-Protokolls [BK92]. Mit einem solchen Protokoll wird dem Systemprogrammierer u.a. die Möglichkeit der Modifikation des in der Sprache vorhandenen Vererbungsmechanismus' in die Hand gegeben, damit er anwendungsspezifische objektorientierte Programmteile direkt und adäquat auf die zugrundeliegende objektorientierte Implementierungssprache abbilden kann.

Im Rahmen des Forschungsprojektes APPLY[1] [GS91] [BCF+92], das die „effiziente und bedarfsgerechte Implementation von Lisp“ zur Zielsetzung hat, stellt die *Komplettkompilation von Programmen*, die Bestandteile des Metaobjekt-Protokolls enthalten, einen zur Zeit aktuellen Forschungsschwerpunkt dar. Wir hoffen, daß auf der Basis der dieser Arbeit zugrundeliegenden Ideen auch die die *Komplettkompilation* von CLOS– und EuLisp-Programmen gelingt.

[1]Das Verbundprojekt APPLY wird vom Bundesminister für Forschung und Technologie unter dem Kennzeichen ITW 9102 gefördert. Die beteiligten Institutionen sind das Fraunhofer Institut für Software und Systemtechnik (ISST), die Gesellschaft für Mathematik und Datenverarbeitung (GMD), die Firma VW-GEDAS sowie das Institut für Informatik und Praktische Mathematik der Christian-Albrechts-Universität (CAU).

Literaturverzeichnis

[Abe91] Abelson, H. et al. Revised^4 Report on the Algorithmic Language Scheme. J. Rees, W. Clinger (Hrsg.), 1991.

[AG87a] D. Ackermann und W. Goerigk. Das Projekt: Übersetzung von Wissensbasen. In *Workshop der Fachgruppe 2.1.4. „Alternative Konzepte für Sprachen und Rechner" der Gesellschaft für Informatik*, Bad Honnef, 1987.

[AG87b] D. Ackermann und W. Goerigk. Übersetzung von PROLOG nach LISP. In *Workshop der Fachgruppe 2.1.4. „Alternative Konzepte für Sprachen und Rechner" der Gesellschaft für Informatik*, Bad Honnef, 1987.

[AGS88] D. Ackermann, W. Goerigk und F. Simon. Kompilation von Wissensrepräsentationssprachen am Beispiel von BABYLON. Institutsbericht Nr. 8810, Institut für Informatik und Prakt. Math. der Christian-Albrechts Universität, Kiel, 1988.

[AGS91] D. Ackermann, W. Goerigk und F. Simon. Wissensbasiskompilation: KI-Technik in industrieller Anwendung. *KI*, 91(2):93–96, Juni 1991.

[BCF+92] H. Bretthauer, Th. Christaller, H. Friedrich, W. Goerigk, W. Heicking, D. Hovekamp, H. Knutzen, J. Kopp, E.U. Kriegel, I. Mohr, R. Rosenmüller und F. Simon. Das Verbundprojekt APPLY: Ein modernes und bedarfsgerechtes Lisp. *KI*, 92(2):50–54, 1992.

[BGK91] O. Burkart, W. Goerigk und H. Knutzen. CLICC: A New Approach to the Compilation from Common Lisp to C. In *Workshop der Fachgruppe 2.1.4. „Alternative Konzepte für Sprachen und Rechner" der Gesellschaft für Informatik*, S. 176–195, Bad Honnef, 1991. Erschienen als Bericht 8/91-I des Instituts für angewandte Mathematik und Informatik der Universität Münster.

[BK92] H. Bretthauer und J. Kopp. Balancing the EuLisp Metaobject Protocol. APPLY Arbeitspapier APPLY/GMD/V.1, Gesellschaft für Mathematik und Datenverarbeitung (GMD), St. Augustin, 1992. Eingereicht und akzeptiert für den *IMSA'92 International Workshop on Reflection and Meta-Level Architecture*, Tokyo.

[Bre87] G. Brewka. A Logic of Inheritance in Frame Systems. In *Proc. of the IJCAI'87 International Joint Conference on Artificial Intelligence*, Mailand, 1987.

[Bur91] O. Burkart. Das Frontend eines Compilers von Common Lisp nach C. Diplomarbeit, Institut für Informatik und Prakt. Math. der Christian-Albrechts Universität, Kiel, 1991.

[Car84] L. Cardelli. A Semantics of Multiple Inheritance. In G. Kahn, D.B. MacQueen und G. Plotkin, Hrsg., *Intern. Symposium on Semantics of Data Types, Lecture Notes in Computer Science No. 173*, S. 51–68, Berlin, Heidelberg, New York, 1984. Springer Verlag.

[CdV89] Th. Christaller, F. diPrimio und A. Voß, Hrsg. *Die KI–Werkbank BABYLON.* Addison–Wesley, Bonn, 1989.

[Cli84] W. Clinger. The Scheme 311 Compiler. An Exercise in Denotational Semantics. In *Proceedings of the 1984 ACM Conference on LISP and Functional Programming*, 1984.

[CM84] W.F. Clocksin und C.S. Mellish. *Programming in Prolog.* Springer Verlag, Berlin, Heidelberg, New York, 1984.

[CP89] W. Cook und J. Palsberg. A Denotational Semantics of Inheritance and it's Correctness. In *Proc. of the OOPSLA'89 International Conference on Object Oriented Programming Systems, Languages and Applications*, S. 433–443, 1989.

[CW86] L. Cardelli und P. Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *Computing Surveys 17*, 4, 1986.

[dBC85] F. diPrimio, D. Bungers und Th. Christaller. BABYLON als Werkzeug zum Aufbau von Expertensystemen. In W. Brauer und B. Radig, Hrsg., *Informatik–Fachberichte Nr. 112 "Wissensbasierte Systeme"*, S. 234–245, Berlin, 1985. Springer Verlag.

[DN66] O. Dahl und K. Nygaard. Simula, an Algol-based Simulation Language. *Communications of the ACM*, 9:671–678, 1966.

[Goe89a] W. Goerigk. Semantik und Übersetzung objektorientierter Wissensrepräsentationssprachen. In *Workshop der Fachgruppe 2.1.4. „Alternative Konzepte für Sprachen und Rechner" der Gesellschaft für Informatik*, Bad Honnef, 1989.

[Goe89b] W. Goerigk. Zur Korrektheit eines Übersetzers für objektorientierte Sprachen: Ein Beweisansatz. In W. Dosch, Hrsg., *Arbeitstreffen Funktionale und logische Programmierung – Sprachen, Methoden, Implementationen*, S. 8–20, Institut für Mathematik der Universität Augsburg, 1989.

[GR83] A. Goldberg und D. Robson. *Smalltalk–80: The Language and its Implementation.* Addison–Wesley, Reading, MA, 1983.

[GS91] W. Goerigk und F. Simon. Zielsetzungen im Verbundvorhaben APPLY. In *Workshop der Fachgruppe 2.1.4. „Alternative Konzepte für Sprachen und Rechner" der Gesellschaft für Informatik*, Bad Honnef, 1991.

[Gü88] H.W. Güsgen. *CONSAT – Foundations of a System for Constraint Satisfaction.* Dissertation, Universität Kaiserslautern, 1988.

[JW78] K. Jensen und N. Wirth. *Pascal User Manual and Report.* Springer Verlag, Berlin, Heidelberg, New York, 1978.

[Kam88] S. Kamin. Inheritance in Smalltalk–80: A Denotational Approach. In *Proc. of the POPL'88 International Conference on Principles of Programming Languages*, S. 80–87. ACM, 1988.

[KdB91] G. Kiczales, J. des Revières und D.G. Bobrow. *The Art of the Metaobject Protocol.* MIT Press, Cambridge, MA, USA, 1991.

[Kee89] S.E. Keene. *Object Oriented Programming in Common Lisp. A Programmer's Guide to CLOS.* Addison–Wesley, Reading, MA, 1989.

[Knu91] H. Knutzen. Codegenerierung und Laufzeitsystem eines Compilers von Common Lisp nach C. Diplomarbeit, Institut für Informatik und Prakt. Math. der Christian-Albrechts Universität, Kiel, 1991.

[KS87] M. Krause und F. Simon. Semantik von Flavors mit statischer Bindung. Institutsbericht Nr. 8702, Institut für Informatik und Prakt. Math. der Christian-Albrechts Universität, Kiel, 1987.

[LS84] J. Loeckx und K. Sieber. *The Foundations of Program Verification.* Wiley-Teubner, Stuttgart, 1984.

[Mey85] B. Meyer. Eiffel: A Language for Software Engineering. Technischer Bericht TRCS85-19, University of California, Santa Barbara, CA, USA, 1985.

[Min80] M. Minsky. A Framework for Representing Knowledge. In *D. Metzing, Hrsg. „Frame Conception and Text Understanding"*, S. 1–25, Berlin, 1980. de Gruyter.

[Pad92] Padget, J. and Nuyens, G. (Hrsg.). The EuLisp Definition, Version 0.8, 1992. Unveröffentlicht.

[Red88] U.S. Reddy. Objects as Closures: Abstract Semantics of Object Oriented Languages. In *Proceedings of the 1988 ACM Conference on LISP and Functional Programming*, S. 289–297, 1988.

[Sch87] P. Schnupp. *Expertensystempraktikum.* Springer Verlag, Berlin, 1987.

[Sco76] D. Scott. Data Types as Lattices. *SIAM Journal of Computing*, 5, 1976.

[Sco81] D. Scott. Lectures on Mathematical Theory of Computation. Technische Monographie PRG-19, Programming Research Group, University of Oxford, Oxford, 1981.

[Ste84] G.L. Steele. *Common Lisp: The Language.* Digital Press, Bedford, MA, 1984.

[Ste90] G.L. Steele. *Common Lisp: The Language. Second Edition.* Digital Press, Bedford, MA, 1990.

[Sto77] J.E. Stoy. *Denotational Semantics: The Scott–Strachey Approach to Programming Language Theory.* MIT Press, Cambridge, MA., London, 1977.

[Str86] B. Stroustrup. *The C++ Programming Language.* Addison Wesley, 1986.

[SW74] C. Strachey und C.P. Wadsworth. Continuations – a Mathematical Semantics for Handling Full Jumps. Technische Monographie PRG-11, Programming Research Group, University of Oxford, Oxford, 1974.

[Sym85] Symbolics Inc. Reference Guide to Symbolics LISP. Benutzerdokumentation, Symbolics Inc., Cambridge, MA, 1985.

[Sym86] Symbolics Inc. Symbolics Common Lisp: Language Concepts. Benutzerdokumentation, Symbolics Inc., Cambridge, MA, 1986.

[Tar55] A. Tarski. A Lattice-Theoretic Fixpoint Theorem and it's Applications. *Pacific Journal of Mathematics*, 5, 1955.

[Wol87] M. Wolczko. Semantics of Smalltalk-80. In *Proc. of the ECOOP'87 European Conference on Object Oriented Programming*, Paris, 1987.

Index

Symbole