Institut für Informatik
Der Technischen Universität München
Univ. Prof. Dr. J. Eickel

Diplomarbeit

Spezifikation und Implementierung einer Annotationssprache für eine objektorientierte Programmiersprache

von
Peter Müller

Aufgabensteller:	Prof. Dr. J. Eickel
Betreuer:	Dr. A. Poetzsch-Heffter
Abgabedatum:	15. November 1995

Erklärung

Ich versichere, daß ich diese Diplomarbeit selbständig verfaßt und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 8. November 1995

Peter Müller

Vorwort

In den vergangenen Monaten hatte ich Gelegenheit, mich mit verschiedenen Überlegungen und Ansätzen zur Annotierung von Programmen zu beschäftigen.
Ich habe mich dazu entschlossen, die in dieser Arbeit entwickelte Annotationssprache praxisnah zu gestalten, um untersuchen zu können, inwiefern der objektorientierte Softwareentwurf durch den Einsatz von Annotationssprachen verbessert werden kann. Kontrollen des Laufzeitverhaltens und Aspekte der formalen Dokumentation erlangen dadurch Vorrang vor der Verifikation.
Die dargestellten theoretischen Grundlagen gelten dennoch auch für andere Formen der Annotierung. An verschiedenen Stellen gebe ich Hinweise, wie die Annotationssprache zu erweitern ist, um die automatische Verifikation der annotierten Programme besser zu unterstützen. Durch die formale Beschreibung der Semantik wird hierfür bereits wesentliche Vorarbeit geleistet.
Ich hoffe, einen interessanten Aspekt der Themenstellung ausgewählt und sowohl Theorie als auch Praxis übersichtlich und verständlich dargestellt zu haben.

Zur Sprache Soweit möglich habe ich in dieser Arbeit versucht, deutsche Fachausdrücke zu benutzen. Begriffe, die im Glossar erklärt werden, sind bei ihrem ersten Auftreten im Text *hervorgehoben* gedruckt. Programmtexte sowie Zitate aus diesen werden im Text durch `Schreibmaschinenschrift` abgesetzt.

Danksagung

Besonderer Dank gilt meinem Betreuer, Herrn Dr. Arnd Poetzsch-Heffter, ohne den die Arbeit nicht in dieser Form zustande gekommen wäre. Durch große Freiheit bei der Schwerpunktsetzung und regelmäßige Diskussion der Ergebnisse trug er entscheidend dazu bei, Motivation und Spaß bis zur Fertigstellung der Arbeit zu erhalten.

München, den 8. November 1995

Peter Müller

Zusammenfassung

In der vorliegenden Arbeit wird eine Annotationssprache für die objektorientierte Programmiersprache Sather entwickelt.
Aus Bertrand Meyers Theorie des *Design by Contract* werden die wesentlichen Konstrukte zur Annotierung von Programmen abgeleitet. Dieser allgemeine Entwurf wird dann auf die Sprache Sather übertragen. Es entsteht die Annotationssprache SALSA — Sather Annotation Language for Specification —, die neben allgemeinen Verfahren zur Annotierung von Programmen auch Mechanismen zur Behandlung von Vererbung und Polymorphismus bereitstellt.
Sowohl Sather als auch SALSA werden mit dem MAX System als Interpreter implementiert. Die dazu benötigte formale Beschreibung der operationellen Semantik stellt den Kern dieser Arbeit dar.
Verschiedene Beispiele demonstrieren die praktischen Einsatzmöglichkeiten der Annotationssprache und machen die Stärken und Schwächen des hier verfolgten Konzepts deutlich.
Abschließend wird gezeigt, daß der Einsatz einer Annotationssprache wesentliche Fortschritte im Bereich der formalen Spezifikation, der automatischen formalen Dokumentation und der Fehlersuche bringt.

Inhaltsverzeichnis

1 Einführung **1**
1.1 Motivation . . . 1
1.2 Ziele der Arbeit . . . 2
1.3 Überblick über die Arbeit . . . 2
1.4 Abgrenzung von verwandten Projekten . . . 3
1.4.1 Sather . . . 3
1.4.2 Eiffel . . . 3
1.4.3 Larch . . . 4

2 Design by Contract — Eine Theorie für den Softwareentwurf **5**
2.1 Die Idee . . . 5
2.2 Die Verträge . . . 6
2.2.1 Ein Beispielvertrag . . . 6
2.2.2 Verträge in der Softwareentwicklung . . . 7
2.3 Die Rolle der Annotationen . . . 8

3 Allgemeiner Entwurf einer Annotationssprache **10**
3.1 Grundlegende Konzepte einer Annotationssprache . . . 10
3.1.1 Der Aufbau von Annotationen . . . 10
3.1.2 Die Formulierung von Verträgen . . . 12
3.2 Verfahren zur Behandlung der Vererbung . . . 15
3.2.1 Das Vererbungskonzept . . . 15
3.2.2 Vererbung von Annotationen auf Prozedurebene . . . 16
3.2.3 Vererbung von Annotationen auf Objektebene . . . 17
3.3 Anforderungen an die Basissprache . . . 18
3.3.1 Funktionale Programmierung . . . 18
3.3.2 Standardfunktionen . . . 19

4 Sather — Die Basissprache **20**
4.1 Beschreibung der Sprache Sather . . . 21
4.1.1 Das Typ- und Klassensystem . . . 21
4.1.2 Die Klassenkomponenten . . . 24
4.1.3 Anweisungen . . . 25

4.1.4 Ausdrücke 26
4.1.5 Annotierung und Ausnahmebehandlung 27
4.2 Definition des implementierten Ausschnitts 27
4.2.1 Bestandteile des Ausschnitts 28
4.2.2 Abweichungen von der Sather-Spezifikation 29
4.2.3 Spezielle Implementierungsdetails 30
4.3 Ein Programmbeispiel 32
4.4 Beschreibung der Semantik 34
4.4.1 Das Spezifikationsverfahren von MAX 34
4.4.2 Die Semantik von Sather 38

5 SALSA — Die Annotationssprache 56
5.1 Der Aufbau einer Annotation 56
5.1.1 Einzelne Bedingungen 56
5.1.2 Zusätzliche Ausdrücke 57
5.1.3 Listen von Bedingungen 59
5.2 Objektinvarianten 62
5.2.1 Vererbung 62
5.2.2 Semantik 62
5.2.3 Beispiel 63
5.3 Vor- und Nachbedingungen 63
5.3.1 Vererbung 64
5.3.2 Semantik 64
5.3.3 Beispiel 70
5.4 Zusicherungen 72
5.4.1 Syntax und Semantik 72
5.4.2 Beispiel 72

6 Der praktische Einsatz von SALSA 74
6.1 Das Programmbeispiel 74
6.2 Die Spezifikation 75
6.2.1 Die abstrakten Typen 75
6.2.2 Die konkreten Typen 80
6.2.3 Grenzen der Ausdrucksmöglichkeit 83
6.3 Die automatische formale Dokumentation 84

7 Beurteilung der Ergebnisse 87
7.1 Das Erreichen der Ziele 87
7.1.1 Die formale Spezifikation von Programmeigenschaften . . . 87
7.1.2 Die Automatisierung einer formalen Dokumentation 88
7.1.3 Die Unterstützung bei der Fehlersuche 88
7.2 Ausblick 89
7.2.1 Verbesserungen des Spezifikationsverfahrens 89

7.2.2 Verbesserungen der Laufzeitumgebung 90
7.2.3 Weiterführung des SALSA-Konzeptes 91

A Programmbeispiele **92**
A.1 Die Klasse **$IS_EQ** 92
A.2 Die Klasse **FRACTION** 92
A.3 Die Klasse **$STACK** 94
A.4 Die Klasse **STACK** 94
A.5 Die Klasse **CALC** 95
A.6 Die Klasse **MAIN** 96

B Die Implementierung des Interpreters **98**
B.1 Die vordefinierten Klassen 98
B.2 Der Quellcode 100
B.2.1 Der Scanner 100
B.2.2 Der Parser 105
B.2.3 Die MAX-Spezifikation 136
B.2.4 Die C-Funktionen 165
B.2.5 Das Makefile 196

C Die Implementierung von AutoDoc **198**
C.1 Die MAX-Spezifikation 198
C.2 Die C-Funktionen 206
C.3 Das Makefile 219

Glossar **221**

Literaturverzeichnis **227**

Abbildungsverzeichnis **229**

Tabellenverzeichnis **231**

Kapitel 1

Einführung

Unter *zuverlässiger* Software versteht man Programme, die zugleich ein hohes Maß an *Korrektheit* und *Robustheit* erreichen. Das bedeutet, sie erfüllen zum einen ihre *Spezifikation* und sind zum anderen in der Lage, beim Auftreten eines Programm- oder Systemfehlers kontrolliert zu reagieren, zum Beispiel eine geordnete Programmbeendigung herbeizuführen, und die Sicherheit der Daten zu gewährleisten.

1.1 Motivation

Obwohl die Erstellung zuverlässiger Programme vorrangiges Ziel beim Softwareentwurf ist, haben Techniken der formalen *Spezifikation* und *Verifikation* bisher kaum Einzug in die praktische Softwareentwicklung gehalten (vgl. [Mey92a, S. 1]). Dies ist umso erstaunlicher, als sich die *objektorientierte Programmierung* weiter auf dem Vormarsch befindet. Einer der Grundgedanken der Objektorientiertheit ist die Wiederverwendung des Programmcodes. Die Tendenz geht also weg vom traditionellen Entwicklungsschema, bei dem jedes Problem durch ein individuelles Programmstück gelöst wurde, hin zur Nutzung von Programmbibliotheken. Der objektorientierte Softwareentwurf besteht daher im wesentlichen aus der neuen Kombination bereits vorhandener Klassen. Der neu zu entwickelnde Code regelt dabei hauptsächlich das Zusammenspiel dieser Bibliotheksklassen. Außerdem dient er der Lösung spezieller Probleme, für die noch keine Implementierung verfügbar ist.

Der Entwickler im objektorientierten Softwareentwurf nutzt also in großem Maße Klassen, deren Implementierung er wegen dem Prinzip der *Kapselung* nicht kennt. Daraus ergeben sich zwei wesentliche Folgerungen:

- Die Korrektheit und Robustheit des genutzten Codes muß gesichert sein.
- Um die Bibliotheksklassen korrekt einbinden zu können, ist eine exakte, das heißt möglichst formale Beschreibung ihres Verhaltens erforderlich.

Man erkennt also, daß die mehrfache Verwendung von Programmstücken den Druck auf die Softwareentwicklung verstärkt, Techniken der formalen Spezifikation und Verifikation anzuwenden, um die oben angesprochenen Probleme zu bewältigen.

1.2 Ziele der Arbeit

In der vorliegenden Arbeit wird die *Annotationssprache SALSA* für die objektorientierte Sprache *Sather* entworfen. SALSA steht für Sather Annotation Language for Specification. Sie ermöglicht dem Programmierer, den in der sog. *Basissprache* formulierten Programmtext um *Annotationen* anzureichern, womit drei wesentliche Ziele verfolgt werden. Die Annotationen sollen

- die formale Spezifikation von Programmeigenschaften ermöglichen,
- die Automatisierung einer formalen Dokumentation des Programmcodes erleichtern und
- den Entwickler bei der Fehlersuche unterstützen.

Sie erleichtern damit die Programmerstellung gemäß einer Spezifikation und die korrekte Anwendung von Bibliotheksklassen. Somit tragen sie erheblich dazu bei, die *Zuverlässigkeit* der Software zu erhöhen.

Da die Annotierung die formale Spezifikation von Programmeigenschaften ermöglicht, wird auch die Verifikation erleichtert. Als Korrektheitsbeweis genügt es meist, zu zeigen, daß ein Programm seine Annotationen, und damit seine Spezifikation, erfüllt. Der Aspekt der Verifikation ist nicht direkt Ziel dieser Arbeit, sondern stellt lediglich eine Perspektive dar, für die gewisse Vorarbeiten geleistet werden, um sie in weiterführenden Arbeiten in Angriff nehmen zu können.

1.3 Überblick über die Arbeit

Aufbauend auf der Theorie des *Design by Contract*, die in Kapitel 2 erläutert wird, erfolgt zunächst eine allgemeine Darstellung der Anforderungen, die eine Annotationssprache für eine objektorientierte Programmiersprache erfüllen muß. Auch an die Basissprache werden im Abschnitt 3 einige Forderungen formuliert. In Kapitel 4 wird die Sprache Sather beschrieben. Es wird zunächst erklärt, wie die *Semantik* einer Sprache mit dem MAX-System formal beschrieben werden kann. Die für SALSA wesentlichen Aspekte der Semantik von Sather werden dargestellt. Den Schwerpunkt der Diplomarbeit bildet die formale Beschreibung der Semantik von SALSA. Sie erfolgt zusammen mit einer Beschreibung des gesamten Sprachschatzes von SALSA in Kapitel 5. Im darauffolgenden Abschnitt wird gezeigt, welche Möglichkeiten sich durch die Verwendung von SALSA eröffnen. Es

werden Beispiele typischer Anwendungen einer Annotationssprache besprochen. Abschließend wird in Kaptiel 7 diskutiert, inwieweit die eingangs (Abschnitt 1.2) formulierten Ziele mit der Annotationssprache SALSA erreicht wurden.

1.4 Abgrenzung von verwandten Projekten

Um die vorliegende Arbeit besser einordnen zu können, werden im folgenden kurz Gemeinsamkeiten und Unterschiede zu verwandten Projekten aufgezeigt. Dargestellt werden hier nur die wichtigsten Ansätze, die die Entwicklung von SALSA beeinflußt haben. Nicht näher besprochen werden die Arbeiten, die im Bereich der fehlertoleranten Programmierung angesiedelt sind, da deren Schwerpunkt meist weniger auf der Annotierung als auf Verfahren zur *Ausnahmebehandlung* liegt. Näheres hierzu findet sich zum Beispiel in [BJ89].

1.4.1 Sather

Die Sprache Sather, auf der die hier beschriebene Annotationssprache aufbaut, enthält bereits einige Möglichkeiten, *Zusicherungen*, sog. *assertions*, in den Programmtext einzustreuen. Diese unterscheiden sich aber von SALSA-Annotationen vor allem dadurch, daß ihre Auswertung den Zustand des Systems verändern kann. Das bedeutet, daß sich ein Programm mit Zusicherungen möglicherweise völlig anders verhält als das gleiche Programm ohne diese. Außerdem sind die Möglichkeiten der *assertions* sehr begrenzt. Eine Behandlung spezieller Eigenschaften der objektorientierten Programmierung, wie z. B. der *Vererbung*, ist nicht vorgesehen. Eine formale Beschreibung der Semantik fehlt völlig, was die Nutzung zur formalen Spezifikation und Verifikation unmöglich macht.

1.4.2 Eiffel

Der in der Sprache *Eiffel* (vgl. [Mey92b]) verfolgte Ansatz geht bereits weit über die Möglichkeiten von Sather hinaus. Geblieben ist jedoch die Möglichkeit, den Systemzustand mittels Annotationen zu manipulieren, da jeder Ausdruck des Typs *Boolean* als Annotation verwendet werden kann, also auch *Methoden*, die unter Umständen den Ablauf des Programms völlig verändern. Auch das Manko einer fehlenden formalen Semantik ist bei Eiffel zu finden.

Da der Entwickler von Eiffel, Bertrand Meyer, zugleich Autor der Theorie des *Design by Contract* ist, die auch dem Entwurf von SALSA zugrunde liegt, weisen die Annotationsmöglichkeiten von Eiffel und die Annotationssprache SALSA große Gemeinsamkeiten auf. Beide sehen Möglichkeiten vor, Programme auf drei Ebenen, nämlich der Objekt-, Prozedur- und Anweisungsebene, zu annotieren. Auch der Aufbau einer Annotation, gegliedert in einen Identifikator und

einen Ausdruck, sowie die Möglichkeit, *dynamische* Abprüfungen unterschiedlichen Ausmaßes durchzuführen, ist identisch. In manchen Bereichen, wie z. B. den *Schleifeninvarianten*, geht Eiffel sogar über SALSA hinaus. Grundgedanke der Entwicklung von SALSA war jedoch nicht die Schaffung einer möglichst umfassenden Annotationssprache, sondern der Entwurf eines kleinen und trotzdem mächtigen Sprachschatzes, dessen semantische Bedeutung exakt definiert ist.

1.4.3 Larch

Der Schwerpunkt des *Larch*-Projekts, das unter anderem am MIT und im DEC System Research Center durchgeführt wird, liegt ausschließlich auf der Spezifikation und Verifikation von Programmen. Grundlegender Bestandteil von Larch ist die Spezifikationssprache *Larch Shared Language LSL*, die in [GHW85] und [GHM90] beschrieben wird. Sie ermöglicht die Erstellung *algebraischer Spezifikationen*. Für jede unterstützte Programmiersprache existiert darüber hinaus eine Komponente, über welche die LSL-Spezifikationen in die Basissprache eingebunden werden. Beispiele hierfür sind LCL für die Sprache C und LM3 für Modula-3 (siehe hierzu [GH93, GH91, Jon91]). LM3 berücksichtigt bereits Besonderheiten der objektorientierten Programmierung. So bezieht sich z. B. Vererbung nicht nur auf Schnittstellen oder Implementierungen, sondern auch auf zugehörige Spezifikationen.

Die Verwendung der algebraischen Spezifikationen durch LSL ermöglicht eine weitgehend vollständige formale Spezifikation der Programmeigenschaften. Gleichzeitig wird aber z. B. durch den Gebrauch von Quantoren eine Unterstützung des Programmierers durch dynamische Abprüfungen unmöglich gemacht. Auch das Ziel der Verifikation wurde bisher kaum erreicht, so daß sich der Nutzen von Larch bis jetzt im wesentlichen auf eine formale Dokumentation beschränkt. Wie Sather und Eiffel entbehren auch die Schnittstellensprachen LCL und LM3 einer formalen Beschreibung der Semantik.

Kapitel 2

Design by Contract — Eine Theorie für den Softwareentwurf

Bertrand Meyer stellt in [Mey92a] eine neue Sichtweise für den Softwareentwurf dar. Aus seiner Theorie lassen sich unmittelbar Aufgabe und Bedeutung von Annotationen ableiten.

2.1 Die Idee

Meyer geht davon aus, daß ein Softwareentwickler zur Implementierung einer Problemlösung ein Programmstück entwirft. Dieses läßt sich in der Regel in mehrere kleinere Einheiten aufteilen. Das Beispiel in Abbildung 2.1[1] gibt eine derartig gegliederte Lösung wieder.

```
my_task is
    do
        subtask_1;
        subtask_2;
        ...
        subtask_n;
    end
```

Abbildung 2.1: Gliederung eines Tasks in Teilaufgaben

Jeder der `subtasks` stellt dabei eine Teilaufgabe dar, die vom Programmierer zu lösen ist. Dies kann grundsätzlich auf zwei Arten geschehen:

[1] siehe [Mey92a, S. 2]

1. Durch eine lokale Implementierung mit einer oder mehreren elementaren Anweisungen;

2. Durch den Aufruf einer *Prozedur*, die andernorts, z. B. in einer Bibliothek, implementiert ist;

Die zweite Variante ist der Hauptansatzpunkt von Meyers Theorie. Die Anwendung eines ausgegliederten Programmstücks, d. h. den Aufruf der zugehörigen Prozedur, interpretiert er als Vertragsschluß zwischen dem Aufrufer und der aufgerufenen Prozedur. Dieser *Vertrag* regelt das Zusammenspiel der beiden Partner. Da die Anwendung von Prozeduren — und damit das Schließen von Verträgen — einen zentralen Punkt in der Softwareentwicklung darstellt, nennt Meyer seine Theorie *Design by Contract*, also „Entwurf durch Vertragsschluß".

2.2 Die Verträge

Aus der oben geschilderten Funktion eines Vertrages lassen sich Aussagen über seine Bestandteile ableiten. Das folgende Beispiel[2] aus dem zwischenmenschlichen Bereich soll nicht nur den Aufbau, sondern auch einige weitere Eigenschaften von Verträgen verdeutlichen.

2.2.1 Ein Beispielvertrag

Wir betrachten einen Vertrag zwischen einem Bauherrn und einer Baufirma:

> Die beiden Partner kommen überein, daß die Baufirma auf einem vom Bauherrn bereitgestellten Grundstück ein mindestens drei Stockwerke hohes Gebäude errichtet, das jedoch nicht teurer als eine Million Francs sein darf. Das Grundstück darf die Größe von einem Hektar nicht unterschreiten.

Die Analyse dieses Vertrages zeigt, daß er für beide Partner sowohl Verpflichtungen als auch Vorzüge beinhaltet. Diese sind in Tabelle 2.1 zusammengefaßt. Bereits an diesem einfachen Beispiel lassen sich einige wichtige Eigenschaften von Verträgen ablesen, die sich später auf den Softwareentwurf übertragen lassen:

- Jeder Vertrag besteht für beide Partner aus Verpflichtungen und Vorzügen. In der Regel stellt die Verpflichtung eines Partners einen Vorzug für den anderen dar.

- Alle Verpflichtungen und Vorzüge sind im Vertrag explizit erwähnt. Ihre detaillierte Beschreibung gehört zur Aufgabe eines Vertrages und dient der Vermeidung von Mißverständnissen.

[2] siehe [Mey92a, S. 3]

- Verträge enthalten möglicherweise einige implizite Bedingungen, die für jeden Vertrag gelten. Hierzu zählen z. B. die Gesetze eines Landes oder bestimmte Geschäftsbedingungen.
- Jede Verpflichtung stellt gleichzeitig auch einen Vorzug für die jeweilige Partei dar. Sie gibt nämlich nicht nur wieder, welche Aufgaben diese erfüllen muß, sondern besagt zudem, daß darüber hinaus keine weiteren Leistungen erbracht werden müssen.

Zusammenfassend läßt sich also feststellen, daß ein Vertrag im wesentlichen eine detaillierte Beschreibung von Vorzügen und Verpflichtungen für jeden der beiden Vertragspartner ist. Durch die exakte Festlegung der Verantwortlichkeiten wird das Zusammenspiel beider Parteien geregelt. In diesem Sinn läßt sich der Begriff auf die Softwareentwicklung übertragen.

	Verpflichtung	Vorzug
Bauherr	Bereitstellung von mindestens 1 Hektar Grund	Erhalt eines dreistöckigen Gebäudes für höchstens 1 000 000 Francs
Baufirma	Errichtung eines dreistöckigen Gebäudes für höchstens 1 000 000 Francs	Keinerlei Verpflichtungen, falls das Grundstück nicht vorhanden oder zu klein ist

Tabelle 2.1: Vorzüge und Verpflichtungen des Beispielvertrages

2.2.2 Verträge in der Softwareentwicklung

Im Bereich des Softwareentwurfs besteht oft die Notwendigkeit, die Interaktion verschiedener Komponenten, z. B. von Prozeduren oder Objekten, zu regeln. Als Veranschaulichung soll die Implementierung einer Funktion dienen, die für ihren Parameter `n` den Wert der Fakultät berechnet, vorausgesetzt $n \geq 0$ (vgl. Abbildung 2.2).

```
proc fac(n: integer): integer
    begin
        if n = 0 then return 1;
        else return n * fac(n-1);
    end;
```

Abbildung 2.2: Implementierung der Fakultätsfunktion

Man erkennt, daß die Anwendung der Funktion nicht völlig unproblematisch ist, da z. B. von der korrekten Belegung des Parameters ausgegangen wird. Für einen

negativen Wert von `n` resultiert der Aufruf in einer nicht terminierenden Rekursion. Die Beschreibung des Verhaltens von `fac` durch einen Vertrag wie in Tabelle 2.2 ist also angebracht.

	Verpflichtung	Vorzug
Aufrufer	Parameter `n` ≥ 0	Erhalt des Wertes `n`!
`fac`	Korrekte Berechnung des Werts `n`!	Undefiniertes Verhalten für `n` < 0 möglich

Tabelle 2.2: Vertrag für den Aufruf der Funktion `fac`

Die explizite Formulierung eines Vertrages bringt durch die detaillierte Regelung der Zuständigkeiten der beiden involvierten Programmstücke wesentliche Erleichterungen für die Programmentwicklung:

- Sie ermöglicht die korrekte Anwendung eines unbekannten Programmcodes, da dessen Verhalten im Vertrag genau fixiert ist. Somit stellt sie zugleich eine formale Spezifikation und Dokumentation des Programmtextes dar, was die Nutzung von Bibliotheken vereinfacht. Die Dokumentation kann durch spezielle Hilfsprogramme, wie dem in Abschnitt 7 beschriebenen *AutoDoc*, maschinell aus dem Quelltext extrahiert werden.

- Jede Komponente beschränkt sich auf die Implementierung ihrer eigentlichen Aufgabe. Sicherheitsabfragen, wie z. B. die Überprüfung der Parameter auf sinnvolle Werte, werden überflüssig. Benutzer der Komponente sind durch die Verpflichtungen des Vertrages ohnehin zur korrekten Anwendung gezwungen. Die Programme werden dadurch effizienter.

- Ein Programmstück kann dann als logisch korrekt bezeichnet werden, wenn es für jede vertragsgerechte Anwendung alle Verpflichtungen seines Vertrages erfüllt. Diese Sichtweise vereinfacht die Durchführung von Korrektheitsbeweisen, da Voraussetzungen und Behauptung des Beweises bereits durch den Vertrag vorgegeben sind. Außerdem wird die Kontrolle des Verhaltens zur Laufzeit ermöglicht. Durch dynamische Abprüfungen können Vertragsverletzungen aufgespürt und somit eine entsprechende Reaktion eingeleitet werden.

Die Einbindung von Verträgen in den Softwareentwurf kann also dazu beitragen, die Erstellung korrekter, robuster und effizienter Programme zu erleichtern.

2.3 Die Rolle der Annotationen

Die Aufgabe einer Annotationssprache in diesem Zusammenhang ist es, die exakte Formulierung der Verträge zu ermöglichen. Annotationen stellen somit formalisierte, in den Programmcode eingebundene Verträge dar. Wie in Abschnitt 2.2.1

beschrieben, bestehen Verträge aus expliziten Vorzügen und Verpflichtungen sowie möglicherweise aus impliziten Bedingungen. Diese werden durch Annotationen folgendermaßen umgesetzt:

- Bedingungen, die vor Abarbeitung eines Codestücks erfüllt sein müssen, stellen die Verpflichtungen für den Nutzer dar. Demzufolge sind sie zugleich Vorzüge für dieses Programmstück. Hierzu zählen z. B. die *Vorbedingungen* einer Prozedur.

- Annotationen, die nach Abarbeitung der Komponente erfüllt sein müssen, stellen die Vorzüge für den Nutzer und somit Verpflichtungen für das Programmstück dar. Hier sind u. a. die *Nachbedingungen* einer Prozedur einzuordnen.

- Die impliziten Bedingungen werden hauptsächlich durch die Semantik der Annotationen vorgegeben. Als Bestandteil jedes Vertrages sind sie für dessen Interpretation unabdingbar, weshalb der Semantik der Annotationssprache in dieser Arbeit besondere Bedeutung beigemessen wird.

Da jede Annotation Teil eines Vertrages ist, muß jede Verletzung einer solchen als Nichteinhaltung des Vertrages interpretiert werden. Vertragsverletzungen stellen grundsätzlich immer einen *Programmfehler* dar: Erfüllt der Anwender einer Komponente seine Verpflichtungen nicht, resultiert dies in einem fehlerhaften Aufruf des externen Codes. Verletzt hingegen das aufgerufene Programmstück seine Verpflichtungen, so kann es nach der Definition auf Seite 8 nicht korrekt sein.

Durch dynamische Abprüfungen lassen sich derartige Vertragsverletzungen aufspüren, was die Einleitung einer entsprechenden Ausnahmebehandlung ermöglicht. Meyer leitet in [Mey92a] auch die Grundzüge eines solchen Mechanismus her, der im Zusammenspiel mit der Annotierung die Erstellung robuster Software ermöglicht. Die Darstellung dieses Gebiets würde jedoch den Rahmen dieser Arbeit sprengen, weshalb ich hier auf [Mar94] verweisen möchte, wo umfassend auf die Rolle der Ausnahmebehandlung eingegangen wird.

Kapitel 3

Allgemeiner Entwurf einer Annotationssprache

Im folgenden Kapitel wird beschrieben, wie eine Annotationssprache aufgebaut sein muß, um zum einen den Ansprüchen des *Design by Contract* zu genügen und zum anderen das Erreichen der in Abschnitt 1.2 genannten Ziele zu ermöglichen. Dabei werden Konzepte vorgestellt, die auf eine Vielzahl von Programmiersprachen anwendbar sind. Gemeint sind damit diejenigen Sprachen, bei denen die Programmausführung durch eine sequentielle Abarbeitung von Anweisungen charakterisiert ist. Jede dieser Anweisungen überführt das System von einem Zustand in einen Folgezustand. Dabei ist es möglich, innerhalb eines Zustandes den Wert von Ausdrücken zu ermitteln. Es werden also sowohl *imperative* als auch *objektorientierte* Sprachen abgedeckt.

3.1 Grundlegende Konzepte einer Annotationssprache

Wie in Kapitel 2 gezeigt, liegt die Hauptaufgabe von Annotationen in der Formalisierung von Verträgen. Eine Annotationssprache muß daher einerseits Konstrukte zur Formulierung der Vorzüge und Verpflichtungen bereitstellen, andererseits die Gestaltung von Verträgen zwischen verschiedenen Partnern erlauben.

3.1.1 Der Aufbau von Annotationen

Aus der Zielsetzung von Kapitel 1.2, den Ausführungen von Abschnitt 2 sowie einigen grundsätzlichen Überlegungen zur Anreicherung von Programmiersprachen um Möglichkeiten der Annotierung lassen sich eine Reihe von Forderungen ableiten:

1. Annotationen sollen den Charakter einer Anmerkung haben. Das bedeutet, daß sich korrekte annotierte Programme nicht anders verhalten sollen als Programme ohne Annotierung.

2. Die Ausdrucksmöglichkeiten der Annotationssprache sollen ausreichen, um alle Vorzüge und Verpflichtungen eines Vertrages zwischen Komponenten der Basissprache formulieren zu können.

3. Um den Softwareentwickler bei der Fehlersuche unterstützen zu können und eine sinnvolle Ausnahmebehandlung zu ermöglichen, muß zur Laufzeit überprüft werden können, ob eine Annotation verletzt ist. Außerdem muß die verletzte Annotation benennbar sein.

Die zweite Forderung legt nahe, die Annotationen direkt auf Konstrukten der Basissprache aufzubauen. Eine Verwendung der Ausdrücke der Basissprache bringt eine Reihe von Vorteilen. Zum einen ermöglicht sie eine ausreichend große Ausdrucksmöglichkeit, zum anderen ist gewährleistet, daß die Verletzung einer Annotation leicht festgestellt werden kann, da hierfür nur diese Ausdrücke ausgewertet werden müssen. Die einfache Implementierbarkeit ist ein weiterer Vorzug dieser Lösung. Um verletzte Annotationen lokalisieren zu können, werden diese zudem durch Identifikatoren benannt. Eine bloße Angabe des Ortes im Quelltext reicht hier nicht aus, da bei Nutzung von Bibliotheken der Quelltext meist nicht vorliegt. Die Dokumentation der Bibliothek sollte jedoch die Annotierung der genutzten Komponenten enthalten.

Eine einzelne Bedingung einer Annotation ist im einfachsten Fall somit ein Tupel, bestehend aus einem Identifikator und einem booleschen Ausdruck. Eine Erweiterung um den Bezeichner der zugehörigen Ausnahmeaktion oder einen Fehlermeldungstext ist sinnvoll, um besser auf die Verletzung einer Annotation reagieren zu können. Da ein Vertrag jeweils mehrere Vorzüge und Verpflichtungen enthalten kann, wird eine Annotation in der Regel nicht nur aus einer, sondern aus einer Liste von Einzelbedingungen bestehen.

Zur Annotierung können diejenigen Ausdrücke der Basissprache herangezogen werden, die keine Änderung des Systems herbeiführen, also der ersten Forderung genügen. Diese werden noch durch spezielle Konstrukte der Annotationssprache ergänzt, die ebenso wie der Begriff *Systemänderung* später (Abschnitt 5.1.2 bzw. 4.4.2) erläutert werden. Durch diese Kombination eröffnet sich eine enorme Ausdrucksmöglichkeit für die Annotationssprache.

Die Einbindung von Funktionen eröffnet die volle Mächtigkeit der Rekursion, birgt jedoch die Gefahr, daß die Auswertung einer Annotation nicht terminiert und dadurch die erste Forderung verletzt wird. Im Verlauf dieser Arbeit wird jedoch immer davon ausgegangen, daß die Annotationen korrekt formuliert sind. Die Problematik nicht terminierender Berechnungen wird nicht näher behandelt, auf eine strikte Semantik verzichtet.

3.1.2 Die Formulierung von Verträgen

Die im vorangegangenen Abschnitt beschriebene Struktur einer Annotation ermöglicht die Formulierung einzelner Bedingungen eines Vertrages. Um zu erkennen, welche Konstrukte zur Spezifikation vollständiger Verträge benötigt werden, muß zuerst ermittelt werden, zwischen welchen Partnern Verträge in objektorientierten Programmen geschlossen werden können.

Bei der Formulierung eines Vertrages genügt es, sich auf einen der beiden Partner zu beschränken, da die Verpflichtungen des einen zugleich die Vorzüge des anderen Partners darstellen. Die Bedingungen eines Vertrages sind bereits dann exakt fixiert, wenn der Auftragsempfänger die erwarteten Voraussetzungen, für ihn die Vorzüge des Vertrages, und die garantierte Leistung, also die Verpflichtung des Vertrages, festlegt. Im folgenden werden daher Verträge immer aus der Sicht der aufgerufenen Komponente beschrieben.

Abbildung 3.1 zeigt die sequentielle Abarbeitung eines Programms. Die Numerierung der Anweisungen gibt dabei nicht ihre Position im Quelltext, sondern die Reihenfolge der Ausführung an. Man erkennt, daß die Anweisungen in *Anweisungsblöcken*, diese in Prozeduren und die wiederum in Objekten organisiert sind. Diese Elemente bilden eine Hierarchie. Bei jedem Eintritt in eine neue Struktur wird ein Vertrag geschlossen. Diese Beziehungen sind durch Pfeile gekennzeichnet.

Aufgrund der oben beschriebenen Gliederung der Anweisungen, Blöcke, Prozeduren und Objekte können beim Betreten einer Komponente gleichzeitig mehrere Verträge zustande kommen. Der Aufruf einer *Methode*[1] in einem anderen Objekt bewirkt z. B. einen Vertragsschluß mit dem Objekt, der Methode und deren erstem Anweisungsblock. Die Hierarchie gibt außerdem die möglichen Empfänger eines Auftrages vor. Es sind dies Objekte, Prozeduren oder Anweisungsblöcke. Einzelne Anweisungen werden hier als Blöcke mit nur einem Element verstanden. Man erkennt also, daß die Annotationssprache Möglichkeiten schaffen muß, Verträge auf diesen drei Ebenen zu spezifizieren.

Die Anweisungsebene

Die Annotierung auf der Anweisungsebene spielt eine eher untergeordnete Rolle, da Anweisungsblöcke, denen eine größere Bedeutung zukommt, meist als eigene Prozedur implementiert werden. Die Annotierung kann dann auf der höheren Hierarchiestufe erfolgen. Es reicht daher aus, einen rudimentären Mechanismus zur Verfügung zu stellen.

Die Annotationssprache sollte die Möglichkeit schaffen, Bedingungen in jedem Zustand des Systems zu prüfen. Da in der Regel Anweisungen die Zustandsänderung bewirken, wird ein Konstrukt benötigt, das eine Annotation zwischen zwei beliebigen Anweisungen einfügt. Dieses Konstrukt wird hier als *Zusicherung* bezeichnet.

[1] Der Begriff *Methode* wird hier synonym zu *Prozedur* und *Routine* gebraucht

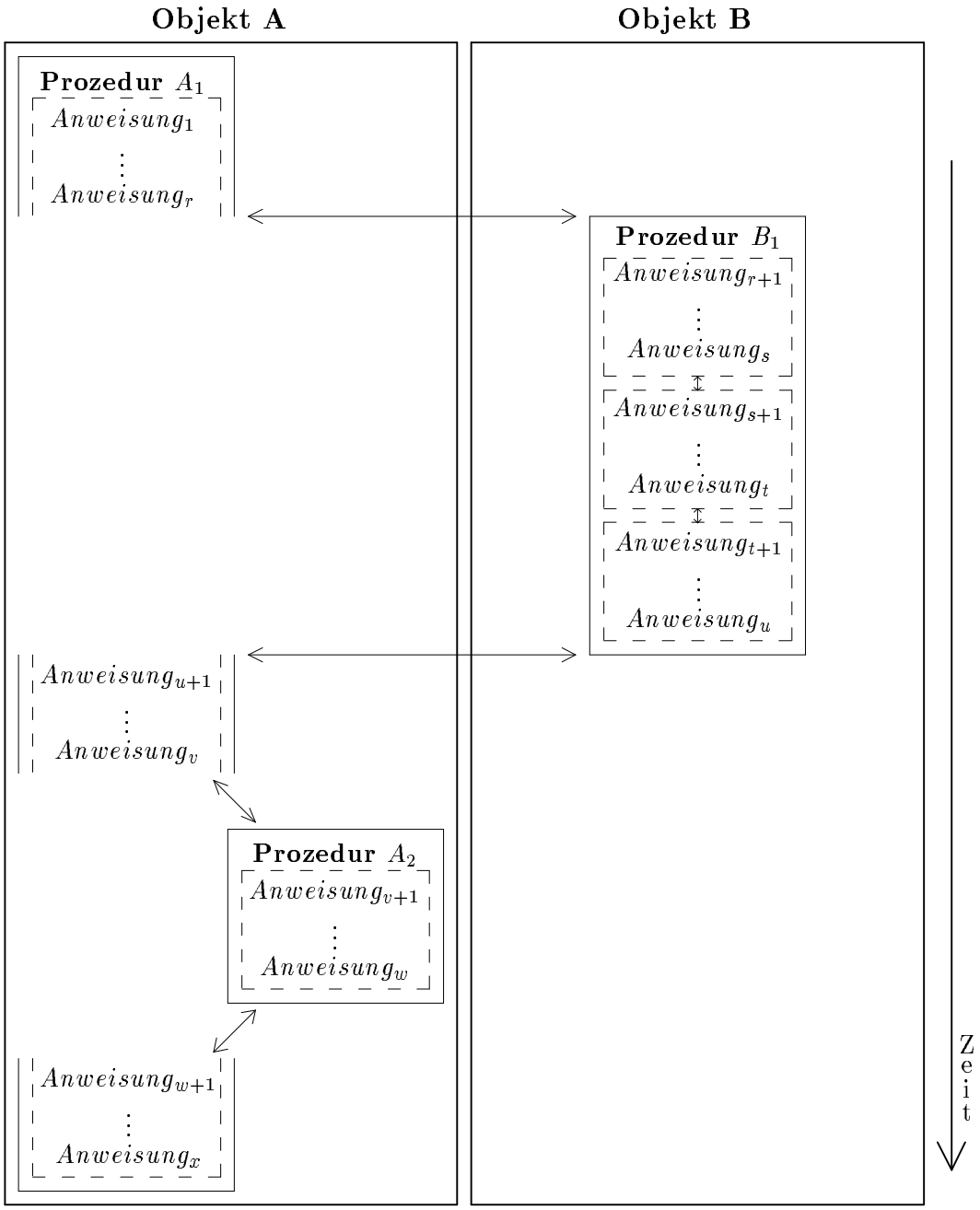

Abbildung 3.1: Gliederung der Anweisungen in Blöcke, Prozeduren und Objekte

Die Gliederung der Anweisungssequenz zu Blöcken erfolgt derart, daß neben Prozeduranfang und -ende nur Zusicherungen die Blockgrenzen markieren. Die Unterteilung weicht also von derjenigen der Basissprache insofern ab, als sie nicht durch die Klammerung mittels `begin`–`end` oder ähnlichem, sondern lediglich durch die Annotierung bestimmt wird.

Zusicherungen erlauben die Formulierung eines Vertragsverhältnisses zwischen zwei aufeinanderfolgenden Anweisungsblöcken B_1 und B_2. Die spezifizierten Bedingungen stellen die Verpflichtungen für B_1 und damit die Vorzüge für B_2 dar. Die Annotierung auf Anweisungsebene wird komfortabler, wenn dem Programmierer neben den Zusicherungen weitere Konstrukte zur Verfügung gestellt werden. Ein Beispiel hierfür sind *Schleifeninvarianten* und Schleifenvarianten, die in [Mey92b, S. 129ff] besprochen werden.

Die Prozedurebene

Ebenso wie im gesamten Softwareentwurf spielen die Prozeduren auch bei der Annotierung eine zentrale Rolle. Der größte Teil der Spezifikation von Programmeigenschaften wird auf Prozedurebene geleistet, weshalb die Annotationssprache hier besondere Möglichkeiten schaffen muß, auch im Hinblick auf Vererbung, die im Abschnitt 3.2 besprochen wird.

Wird ein Vertrag mit einer Prozedur geschlossen, so bestehen die Vorzüge für diese in gewissen Bedingungen, die beim Aufruf gelten müssen. Diese werden im weiteren als *Vorbedingungen* bezeichnet. Analog dazu stellen die *Nachbedingungen* die Verpflichtungen dar, deren Erfüllung beim Verlassen der Prozedur garantiert sein muß. Durch Vor- und Nachbedingungen kann also der gesamte Vertrag formuliert werden.

Bei der Abarbeitung einer Prozedur gibt es somit zwei ausgezeichnete Zustände. Unmittelbar vor Betreten des Prozedurrumpfes — die Parameter sind zu diesem Zeitpunkt bereits mit Werten versorgt — können die Vorbedindungen geprüft werden. Dieser Zustand, in der Literatur als *prestate* bezeichnet, wird im folgenden *Initialzustand* genannt. Im Zustand nach dem Austritt aus dem Rumpf, dem *poststate* oder *Terminalzustand*, erfolgt die Auswertung der Nachbedingungen. Hier ist der Rückgabewert der Prozedur, soweit vorhanden, bekannt.

Um eine sinnvolle Spezifikation zu ermöglichen, muß der Rückgabewert in Abhängigkeit von den Prozedurparametern beschrieben werden können. Daher ist es erforderlich, auf die Parameterwerte des Initialzustandes vom Terminalzustand aus zugreifen zu können. Elegantere Formulierungen ergeben sich jedoch, wenn die Werte ganzer Ausdrücke im Initialzustand berechnet und im Terminalzustand genutzt werden können. Diese Ausdrücke werden hier als *Initialausdrücke* bezeichnet. Zu ihrer Kennzeichnung wird ein neues Schlüsselwort, wie z. B. `old` in Eiffel oder `initial` in Sather, eingeführt. Dieser Mechanismus kann auch den Zusicherungen auf Anweisungsebene zugänglich gemacht werden.

Neben den Initialausdrücken müssen die Ausdrücke der Basissprache auch noch um eine Zugriffsmöglichkeit auf den Rückgabewert der Prozedur erweitert werden. Von dieser kann natürlich nur in den Nachbedingungen Gebrauch gemacht werden.

Die Objektebene

Die Annotierung auf der Objektebene beschränkt sich auf die Formulierung gewisser Bedingungen, die in jedem nach außen sichtbaren Zustand des Objekts erfüllt sein müssen. Das heißt, sie müssen beim Betreten und Verlassen des Objekts garantiert werden, wohingegen eine Verletzung während des Aufenthalts im Objekt möglich ist. Diese Bedingungen werden *Objekt-* oder *Klasseninvarianten* genannt. Sie fungieren zugleich als Vorzug und Verpflichtung eines Vertrages, da sie einerseits beim Betreten des Objekts vom Aufrufer, andererseits aber auch beim Verlassen vom Objekt selbst garantiert werden müssen.

Aus diesem Grund kann man Objektinvarianten auch als Vor- bzw. Nachbedingung sehen: Vorbedingung derjenigen Prozedur, deren Aufruf den Eintritt in das Objekt bewirkt, und Nachbedingung der letzten innerhalb des Objekts ausgeführten Methode. Lediglich in Bezug auf Vererbung ergeben sich hier einige Unterschiede zwischen beiden Sichtweisen, wie in Abschnitt 3.2.3 deutlich gemacht wird.

Exisitieren in der Basissprache ausgezeichnete Konstruktoren für Objekte, so brauchen die Invarianten beim Aufruf des Konstruktors nicht geprüft zu werden, wohl aber für das neu erzeugte Objekt.

Objektinvarianten eignen sich zur Formulierung von Bedingungen, die während des ganzen Lebenszyklus des Objekts erfüllt sein müssen. Hierunter zählt nicht nur die Konsistenz der Attribute, auch der Zusammenhang zu anderen Objekten kann so beschrieben werden. Beispiele hierfür, wie auch zur Anwendung der anderen dargestellten Mechanismen, finden sich in Abschnitt 6.

3.2 Verfahren zur Behandlung der Vererbung

Die im vorangegangenen Abschnitt dargestellten Verfahren sind geeignet, die verschiedenen Verträge objektorientierter Programmierung zu spezifizieren. Im folgenden Abschnitt wird nun beschrieben, welche Auswirkungen die typischen Eigenschaften von objektorientierten Sprachen wie Vererbung (und damit *Polymorphismus* und *dynamisches Binden*) auf die Annotierung haben.

3.2.1 Das Vererbungskonzept

Es gibt eine Reihe unterschiedlicher Vorgehensweisen, um Vererbung bei einer objektorientierten Programmiersprache zu realisieren. Bevor auf eine geeignete

Behandlung der Vererbung durch die Annotationssprache eingegangen werden kann, muß deshalb erklärt werden, welches Vererbungskonzept den hier vorgestellten Überlegungen zugrunde liegt.

Bei der Vererbung werden alle öffentlichen Methoden und Attribute der Superklasse an die abgeleitete Klasse weitergegeben. Ist die übergeordnete Klasse ein abstrakter Typ, d. h. sie besteht aus einer Schnittstelle ohne die zugehörige Implementierung, so werden lediglich die Signaturen ihrer Methoden vererbt. Falls die erbende Klasse nicht ihrerseits wieder abstrakt ist, ist sie verpflichtet, diese Implementierung zur Verfügung zu stellen. Beide Arten des Erbens — sowohl von abstrakten wie auch von konkreten Superklassen — gehen mit der Definition von Methoden einher, sei es durch die Bereitstellung einer Implementierung für eine abstrakte Signatur oder durch die mögliche Neudefinition geerbter Methoden.

Um Polymorphismus und dynamisches Binden zu ermöglichen, muß dabei jederzeit gewährleistet sein, daß ein an eine Superklasse gerichteter Methodenaufruf von jeder abgeleiteten Klasse verarbeitet werden kann. Daraus folgt, daß jeder formale Parameter der neu implementierten Methode vom gleichen Typ oder einem in der Vererbungshierarchie übergeordneten Typ sein muß wie der korrespondierende Parameter der ersetzten Methode bzw. der abstrakten Signatur. Umgekehrt muß der Rückgabetyp der neuen Methode gleich dem der geerbten sein, oder aber unterhalb dessen in der Hierarchie stehen.

Dadurch wird erreicht, daß die neu implementierte Methode mindestens die gleichen Parametertypen wie die geerbte verarbeiten kann. Gleichzeitig ist sichergestellt, daß der Code, der den Rückgabewert der Prozedur verwendet, deren Ergebnis handhaben kann, da die Menge der möglichen Rückgabetypen eine Teilmenge der Rückgabetypen der Methode der Superklasse ist. Methoden abgeleiteter Klassen können also allgemeiner als die der Superklassen gehalten werden, ohne größere Ansprüche an die aufrufende Befehlsfolge zu stellen.

3.2.2 Vererbung von Annotationen auf Prozedurebene

Dieses Konzept läßt sich auch auf die Annotierung übertragen. Da Methoden allgemeiner, nicht jedoch restriktiver als die korrespondierenden Signaturen übergeordneter Klassen sein dürfen, können seitens der Vorbedingungen lediglich *Abschwächungen*, aber keine *Verschärfungen* zugelassen werden. Das heißt, jede Einschränkung der Universalität der Methode durch eine Vorbedingung muß bereits bei den Superklassen vorhanden sein. Für die Nachbedingungen gilt analog die Umkehrung. Hier sind Verschärfungen erlaubt, Abschwächungen jedoch keinesfalls möglich. Dabei gilt: Eine Bedingung A ist dann eine Abschwächung von B, wenn A aus B folgt. Umgekehrt ist A eine Verschärfung von B, wenn A B impliziert.

Dieser Sachverhalt läßt sich sehr gut am Beispiel aus Abschnitt 2.2.1 verdeutlichen. Der Bauherr wäre sicherlich mit einer Abschwächung seiner Verpflichtung und einer Vergrößerung seiner Vorzüge einverstanden: er würde das errichtete Gebäude auch dann akzeptieren, wenn die Kosten nur 800 000 Francs und der Platzbedarf weniger als ein Hektar betragen würden.

Mit der beschriebenen Vorgehensweise lassen sich die Prinzipien des Polymorphismus und des dynamischen Bindens auf annotierte Programme übertragen. Die Problematik dieses Ansatzes liegt jedoch darin, daß es einem Compiler oder Interpreter kaum möglich ist, Verschärfungen und Abschwächungen zu erkennen. Ein Ausweg eröffnet sich, wenn hierfür nur die logische Und- und Oder-Verknüpfung zugelassen werden.

Die Analogie zwischen Annotierung und Typisierung einer Methode bei der Vererbung läßt den Schluß zu, daß Typangaben prinzipiell als Annotationen aufgefaßt werden können. Typisierte Sprachen ließen sich somit wie annotierte untypisierte Sprachen behandeln. Ein näheres Eingehen auf diesen Sachverhalt geht jedoch über die Zielsetzung der Arbeit hinaus.

3.2.3 Vererbung von Annotationen auf Objektebene

Wie im Abschnitt 3.1.2 gezeigt, erfolgt die Annotierung auf Objektebene mit Hilfe von Invarianten. Sie dienen der Formulierung bestimmter Bedingungen, die für jedes Objekt dieser Klasse eingehalten werden müssen. Da jedes dieser Objekte gleichzeitig auch Instanz der übergeordneten Klassen ist, muß jede Klasse alle Invarianten ihrer Superklassen enthalten.

Im wesentlichen ist eine Annotation eine Liste einzelner Bedingungen. Die Invarianten der Superklassen können, sofern sie nicht bereits enthalten sind, einfach an die bestehende Liste angefügt werden. Somit besteht die Annotation einer Klasse aus der Vereinigung der Invarianten aller ihrer Superklassen, kann jedoch noch um eigene Bedingungen erweitert werden, um implementierungsspezifische Forderungen formulieren zu können.

Hier wird deutlich, warum Invarianten nicht immer als Vor- und Nachbedingungen von Methoden gesehen werden dürfen. Die Invarianten der Superklassen dürfen nämlich um neue Bedingungen erweitert werden, was eine Verschärfung darstellt. Hier wird also vom oben dargestellten Prinzip abgewichen. Dies ist insbesondere dadurch erforderlich, daß es bei der Verwendung abstrakter Klassen sonst nicht möglich ist, implementierungsabhängige Forderungen einzubringen. Diese können nur in tieferen Klassen der Vererbungshierarchie formuliert werden, da die Implementierung erst dort existiert.

3.3 Anforderungen an die Basissprache

Die dargestellten Merkmale einer Annotationssprache stellen die Grundelemente dar, die zur Realisierung des *Design by Contract* erforderlich sind. Da sich die Annotationen stark auf die Basissprache abstützen, muß auch diese einige Mindestanforderungen erfüllen, um eine sinnvolle Annotierung überhaupt erst möglich zu machen.

3.3.1 Funktionale Programmierung

Die einzelnen Bedingungen einer Annotation bestehen im wesentlichen aus Ausdrücken der Basissprache. Wie bereits erwähnt, muß sichergestellt sein, daß deren Auswertung den Gesamtzustand des Systems nicht verändert. Das bedeutet, daß Objekte auf der Halde weder kreiert noch manipuliert oder entfernt werden dürfen.

Problematisch ist die Umsetzung dieser Forderung durch die Zulassung von Prozeduraufrufen in den Annotationen. Es muß garantiert sein, daß auch die genutzten Methoden den Systemzustand nicht verändern. Sie können daher lediglich einen Wert liefern, bei dessen Berechnung keine Seiteneffekte auftreten. Diese Methoden können deshalb als *funktional* bezeichnet werden.

Um dem Compiler oder Interpreter zu ermöglichen, die Einhaltung der *funktionalen Programmierung statisch* zu überprüfen, sollten funktionale Methoden auch als solche gekennzeichnet sein. Dynamisches Binden erfordert zudem, daß eine Prozedur einer abgeleiteten Klasse funktional ist, falls deren korrespondierende Methode der Superklasse diese Eigenschaft aufweist.

Das Verbot, Objekte auf der Halde explizit zu erzeugen oder zu verändern, setzt voraus, daß neben den sog. *Referenzobjekten* noch eine weitere Gruppe von Objekten, die *Value-Objekte*, zur Verfügung stehen. Referenzobjekte sind Objekte, die im Speicher erzeugt und stets über Referenzen angesprochen werden. Sie stehen möglicherweise in Geflechten mit anderen Objekten in Verbindung, weshalb eine Manipulation enorme Seiteneffekte haben kann.

Im Gegensatz dazu stehen die Value-Objekte, deren Bearbeitug nicht über Referenzen erfolgt. Es werden hier direkt die Werte, nicht ihre Speicherplätze verwendet. Die Veränderung eines Value-Objekts bewirkt stets die Erzeugung eines neuen Objekts, weshalb hierbei keine Auswirkungen auf andere Objekte auftreten können.

Die Basissprache kann dem Programmierer entweder direkt die Möglichkeit bieten, ein Klasse als *Referenz-* oder *Value-Klasse* zu deklarieren, oder bestimmte Grundtypen wie `integer`, `char` oder `boolean` mit Value-Semantik zur Verfügung stellen.

3.3.2 Standardfunktionen

Bei der Annotierung objektorientierter Programme ist es wünschenswert, Aussagen über Objekte treffen zu können, die z. B. ihren Typ oder ihre Lokalisation im Speicher betreffen. Besonderer Bedeutung kommt hierbei der Objektgleichheit zu. Sie spielt bei der Formulierung von Beziehungen zwischen Objekten, z. B. in Geflechten, eine große Rolle. Außerdem sollte die Möglichkeit bestehen, den Typ eines Objekts zur Laufzeit zu ermitteln, um beispielsweise das Ergebnis einer Prozedur in Abhängigkeit vom Parametertyp spezifizieren zu können.

In der Annotationssprache werden dafür keine Mechanismen zur Verfügung gestellt, da die meisten objektorientierten Sprachen diese Prozeduren ohnehin anbieten. Sie können daher als gegeben vorausgesetzt werden.

Kapitel 4

Sather — Die Basissprache

Thema der Arbeit ist der Entwurf und die Implementierung einer Annotationssprache für eine objektorientierte Programmiersprache. Nachdem in den vorangegangenen Kapiteln die theoretischen Überlegungen, die in die Entwicklung einer Annotationssprache einfließen, dargestellt worden sind, wird nun mit dem konkreten Entwurf und der Implementierung begonnen.

Da Annotationen in engem Zusammenhang mit der Basissprache stehen, kann ihre Semantik nur verstanden werden, wenn man mit der Basissprache vertraut ist und deren Semantik kennt. Im folgenden Kapitel wird deshalb auf die Sprache Sather, die im praktischen Teil dieser Arbeit als Basissprache dient, eingegangen, wobei neben den grundlegenden Sprachkonzepten auch Semantik und Implementierung in Auszügen besprochen werden.

Die Implementierung erfolgt als Interpreter mit Hilfe des MAX Systems. MAX steht für *Munich Attribution system for UNIX environments*. Es bietet neben einer umfangreichen Attributauswertung vor allem die Möglichkeit, die *dynamische Semantik* der Sprache zu formalisieren. Die MAX Spezifikation stellt also zugleich eine formale Beschreibung der *operationellen Semantik* sowie die Implementierung der Sprache dar.

Neben der MAX Spezifikation besteht der Interpreter aus einem von lex erzeugten Scanner, einem mit yacc generierten Parser sowie einigen C-Funktionen zur Unterstützung der anderen Komponenten des Interpreters. Hauptaufgabe des Parsers ist neben der syntaktischen Analyse die Erstellung eines abstrakten Syntaxbaums, auf dem das MAX System zuerst mit der statischen, dann mit der dynamischen Auswertung aufsetzt.

In dieser Arbeit wird nur auf die zum Verständnis der Semantik von Basis- und Annotationssprache notwendigen Teile der Implementierung eingegangen. Der vollständige Quelltext befindet sich in Anhang B.2, eine nähere Beschreibung des MAX Systems findet man in [PHE93].

4.1 Beschreibung der Sprache Sather

Sather ist eine objektorientierte Progammiersprache, die am *International Computer Science Institute* in Berkeley entwickelt wurde. Als Basissprache für die Entwicklung der Annotationssprache eignet sie sich vor allem deshalb, weil sie alle typischen Konzepte objektorientierter Progammierung enthält. Die Überlegungen dieser Arbeit können deshalb leicht auf andere Sprachen übertragen werden.

Außer durch Mehrfachvererbung, parametrisierte Klassen und Polymorphismus zeichnet sich Sather durch Iterationsabstraktion, *Speicherbereinigung* und Möglichkeiten der Annotierung und Ausnahmebehandlung aus. Die für das Verständnis der Annotationssprache und der Beispiele wichtigen Aspekte Sathers werden im folgenden kurz beschrieben. Eine ausführliche Darstellung findet sich in [Omo94] und [MOS93].

4.1.1 Das Typ- und Klassensystem

Der Quelltext eines Sather-Programms besteht aus einer Liste von Klassendefinitionen. Dabei können fünf Arten von Klassen auftreten: drei konkrete Typen, die Referenzklassen, Value-Klassen und gebundenen Klassen, sowie abstrakte und externe Typen. Objekte können nur für konkrete Typen erzeugt werden. Abstrakte Typen stellen Mengen von konkreten Typen dar, externe Typen dienen der Einbindung anderer Sprachen, wie z. B. C, in Sather. Für die weiteren Überlegungen spielen nur Referenz-, Value- und abstrakte Typen eine Rolle, weshalb auf die beiden anderen hier nicht näher eingegangen wird.

Vererbung

Auf den Klassen ist die Vererbungsrelation definiert. Man kann daher einen gerichteten Vererbungsgraphen konstruieren, dessen Knoten die Klassen und dessen Kanten diese Relation bilden. Verläuft in der transitiven Hülle des Graphen eine Kante von T_1 zu T_2, so ist T_1 eine Superklasse von T_2, T_2 erbt von T_1. Der Graph ist azyklisch und kann deshalb als Hierarchie aufgefaßt werden, in der Superklassen über abgeleiteten Klassen stehen. Außerhalb der untersten Hierarchiestufe, können in Sather nur abstrakte Typen auftreten. Alle Klassen sind von der vordefinierten Klasse `$OB` abgeleitet. Diese bildet somit die Spitze der Hierarchie und die Vereinigung aller im Programm auftretenden Typen. Die Definition der Klasse `$OB` ist in Anhang B.1 zu finden. Einen Vererbungsgraphen gibt Abbildung 6.1 wieder.

Jede Variable hat einen deklarierten Typ. Ihr können nur Objekte zugewiesen werden, deren Typ vom deklarierten Typ abgeleitet ist. Das bedeutet, daß die Typkorrektheit in Sather statisch überprüft werden kann.

Die Operationen auf Objekten werden mit Routinen und Iteratoren durchgeführt. Letztere stellen eine Abstraktion der Iteration dar, sind aber für die Annotierung gleichbedeutend mit den Routinen, weshalb sie im folgenden übergangen werden.

Auf den Routinen sind ebenfalls Relationen definiert: zum einen die Konfliktrelation, zum anderen die Konformitätsrelation. Die Signaturen zweier Routinen f und g treten genau dann in Konflikt, wenn gilt:

1. f und g haben den gleichen Namen.
2. f und g haben die gleiche Anzahl von Parametern.
3. Entweder liefern beide oder keine einen Rückgabewert.
4. Der Typ jedes Parameters von f ist gleich dem des korrespondierenden Parameters von g, oder einer der Typen ist abstrakt.

Davon unabhängig ist die Konformitätsrelation definiert. Die Signatur von f ist konform zu der von g, wenn gilt:

1. f und g haben den gleichen Namen.
2. f und g haben die gleiche Anzahl von Parametern.
3. Der Typ jedes Parameters von g ist gleich oder steht im Vererbungsgraphen unterhalb des Typs des korrespondierenden Parameters von f.
4. Entweder liefern beide oder keine einen Rückgabewert.
5. Falls f und g Werte zurückliefern, so ist der Typ des Rückgabewertes von f gleich oder steht im Vererbungsgraphen unterhalb des Typs des Rückgabewertes von g.

Die Konformität, die wegen den Forderungen 3 und 5 nicht symmetrisch ist, ist auch auf den Schnittstellen von Klassen definiert:

> Unter einer Schnittstelle versteht man die Menge aller öffentlichen Methoden, die auf ein Objekt angewandt werden dürfen. Eine Schnittstelle S_1 ist konform zu S_2, wenn es zu jeder Routine f_2 in S_2 eine einzige konforme Routine f_1 in S_1 gibt.

Als Grundregel der Vererbung kann dann formuliert werden:

> Die Schnittstelle einer Klasse muß konform zu den Schnittstellen aller ihrer Superklassen sein.

Dadurch wird erreicht, daß ein Methodenaufruf an eine Klasse auch von allen abgeleiteten Typen verarbeitet werden kann.

Abstrakte Typen

Abstrakte Typen werden in Sather-Programmen mit Identifikatoren benannt, die aus einem $-Zeichen, gefolgt von einer Sequenz aus Großbuchstaben bestehen. Sie definieren lediglich Schnittstellen und enthalten deshalb die Signaturen der Routinen, ohne eine Implementierung anzubieten. Wegen obiger Grundregel der Vererbung müssen diese Routinen von jeder abgeleiteten konkreten Klasse bereitgestellt werden. Unter *Vererbung* versteht man in Sather also lediglich die Weitergabe von Schnittstellen. Es liegt daher, anders als z. B. in C++, eine strenge Trennung von Vererbung und *code-sharing* vor.

Eine Besonderheit Sathers ist es, daß eine Vererbungsbeziehung nicht nur von der abgeleiteten Klasse, sondern auch von einer neu eingeführten Superklasse aus hergestellt werden kann. Dies ermöglicht das nachträgliche Einfügen von abstrakten Typen in die Vererbungshierarchie und somit die Definition beliebiger Mengen von Klassen.

Bei der Vererbung zwischen abstrakten Typen werden die Signaturen aller Superklassen in die Liste des abgeleiteten Typs aufgenommen. Im Konfliktfall verschatten explizit spezifizierte Signaturen die geerbten. Tritt bei der Mehrfachvererbung ein Konflikt zwischen zwei geerbten Signaturen auf, so muß eine Signatur angegeben werden, die beide abdeckt.

Ebenso wie Referenz- und Value-Klassen können abstrakte Typen parametrisiert sein. Bei der Instanziierung verhält sich die Klasse exakt wie eine textuelle Kopie der parametrisierten Klasse, bei der jedes Auftreten des Parameters durch den aktuellen Typ ersetzt wurde.

Konkrete Typen

Auf die Rolle der Referenz- und Value-Klassen wurde bereits in Abschnitt 3.3.1 eingegangen. Sather bietet dem Programmierer die Möglichkeit, sowohl Referenz- als auch Value-Klassen zu definieren. Ebenso wie abstrakte Typen werden sie mit Namen aus Großbuchstaben bezeichnet, jedoch ohne die Voranstellung eines $-Zeichens.

Referenzobjekte müssen explizit erzeugt werden. Sie werden stets über ihre Referenz angesprochen, sei es bei der Verwendung als Parameter einer Methode oder bei Zuweisungen. Value-Objekte enthalten dagegen auch ohne explizite Kreierung einen Wert, den sog. `void`-Wert, der in Abschnitt 4.1.4 näher erklärt wird. Daher können rekursive Datenstrukturen nicht als Value-Klassen realisiert werden, weil die Instanziierung nicht terminieren würde. Weitere Unterschiede zwischen Value- und Referenzklassen werden bei der Besprechung der *Klassenkomponenten* deutlich.

4.1.2 Die Klassenkomponenten

Während abstrakte Typen nur Signaturen enthalten, können konkrete Typen aus einer Reihe verschiedener Klassenkomponenten bestehen. Neben Klassen-, Objekt- und konstanten Attributen existieren Routinen- und Iteratorendefinitionen sowie *Inkludierungsbestimmungen*. Die Reihenfolge ihrer Definition hat keinen Einfluß auf die Semantik.

Komponenten, die als `private` deklariert sind, können nur innerhalb der Klasse angesprochen werden. Sie sind nicht Teil der nach außen sichtbaren Schnittstelle.

Attribute

Klassenattribute sind Attribute, die für alle Objekte einer Klasse gemeinsam zugänglich sind, d. h. ihr Wert ist für alle Objekte stets der gleiche. Ansonsten verhalten sie sich wie Objektattribute, weshalb sie im weiteren nicht mehr gesondert erwähnt werden.

Jede Definition eines Objektattributs bewirkt die Einführung einer Lese- und Schreibroutine mit gleichem Namen wie das Attribut. Für konstante Attribute wird nur eine Leseroutine erstellt, da ihr Wert ohnehin nach der Kreierung des Objekts nicht mehr verändert werden kann.

Die Leseroutine hat keine Parameter, der Rückgabewert ist vom gleichen Typ wie das Attribut. Sie ist privat, falls das Attribut als `private` deklariert wurde. Ein Aufruf dieser Routine bewirkt die Rückgabe des aktuellen Attributwertes.

Schreibroutinen nehmen für Value- und Referenztypen unterschiedliche Formen an. Die Schreibroutine eines Attributs mit Value-Charakter erwartet einen Parameter vom Typ des Attributs und gibt als Rückgabewert eine Kopie des Objekts zurück, in der das Attribut auf den neuen Wert gesetzt wurde. Dies ist erforderlich, da Value-Objekte nach ihrer Kreierung nicht mehr verändert werden können. Jede Manipulation resultiert daher in der Erzeugung eines neuen Objekts.

Für Attribute mit Referenztyp arbeiten die Schreibroutinen einfacher. Auch sie haben einen Parameter vom Typ des Attributs, geben aber keinen Wert zurück. Nach ihrem Aufruf enthält das Attribut den neuen Wert, es wird also das bestehende Objekt verändert. Schreibroutinen sind dann privat, wenn das zugehörige Attribut als `private` oder `readonly` deklariert wurde.

Routinen und Iteratoren

Die Signatur von Routinen und Iteratoren besteht aus einem Identifikator, einer optionalen Parameterliste und evtl. einem Rückgabetyp. Sie bildet zusammen mit dem Prozedurrumpf und den Vor- und Nachbedingungen die Definition der Routine bzw. des Iterators. Der Rumpf ist dabei eine Folge von Anweisungen. Auf die Vor- und Nachbedingungen wird in Abschnitt 4.1.5 eingegangen.

Innerhalb einer Klasse dürfen keine Signaturen miteinander in Konflikt treten. Dadurch wird erreicht, daß die Auswahl der zu einem Aufruf passenden Methode in der Regel eindeutig erfolgen kann.

Die Ausführung des Programms beginnt mit der Routine `main` in der Instanz einer spezifizierten Klasse. Wird keine Klasse angegeben, so wird die Routine `main` eines Objekts der Klasse `MAIN` aufgerufen.

Inkludierungsbestimmungen

Code-sharing wird in Sather mittels Inkludierungsbestimmungen realisiert. Diese ermöglichen das Einbinden von Code aus fremden Klassen. Dabei ist es möglich, einzelne Klassenkomponenten umzubenennen oder gar nicht zu übernehmen. Für die inkludierten Komponenten kann die Deklaration als `private` oder `readonly` neu entschieden werden.

Bei der Inkludierung ist darauf zu achten, daß der Graph, dessen Knoten die Klassen und dessen Kanten die Inkludierungsbestimmungen darstellen, keine Zyklen enthält.

Namenskollisionen bei der Inkludierung können durch Umbenennung der neuen Komponenten vermieden werden. Dabei wird jedoch der Identifikator nur bei der Deklaration, nicht aber bei angewandtem Auftreten verändert. Dies führt besonders bei rekursiven Routinen zur Sinnentstellung.

4.1.3 Anweisungen

An dieser Stelle sollen nur diejenigen Anweisungen des Sather-Sprachschatzes erläutert werden, deren Anwendung vom gewohnten Schema anderer Sprachen abweicht. Es sind dies Schleifen und `typecase`-Anweisungen.

Schleifen

Iterationen werden in Sather mittels `loop`-Anweisungen programmiert. Der zwischen den Schlüsselwörtern `loop` und `end` eingeschlossene Code wird in endloser Wiederholung durchlaufen. Eine Unterbrechung der Ausführung kann nur durch Aufruf eines Iterators geschehen.

Neben den benutzerdefinierten exisieren auch die vordefinierten Iteratoren `while!`, `until!` und `break!`. Dabei bewirkt `while!` das Verlassen der Schleife, falls das boolesche Argument `false` liefert, während `until!` die Schleife fortsetzt, bis dieses `true` ergibt. Keinen Parameter erwartet dagegen `break!`. Es führt immer zum Schleifenabbruch.

Die `typecase`-Anweisung

`typecase`-Anweisungen ermöglichen dem Programmierer, Anweisungen in Abhängigkeit vom dynamischen Typ einer Variable auszuführen. Der Aufbau der

Anweisung ähnelt dem der `case`-Anweisung. Nach der Nennung einer lokalen Variablen oder eines Prozedurparameters folgt eine Liste von Typbezeichnern, jeder gefolgt von einer Anweisungssequenz. Ausgeführt wird die erste Anweisungssequenz, deren Typbezeichner gleich dem dynamischen Typ der Variable ist oder über diesem in der Vererbungshierarchie steht. Wird kein passender Typ gefunden, so wird mit der Bearbeitung des `else`-Falls fortgefahren.

Die Besonderheit der `typecase`-Anweisung liegt darin, daß die untersuchte Variable innerhalb der Anweisungssequenz eines einzelnen Falles nicht vom deklarierten Typ, sondern von dem Typ ist, der in diesem Fall angegeben wurde. So können spezielle Operationen auch auf Objekten ausgeführt werden, die in einer allgemeiner deklarierten Variable gehalten werden.

4.1.4 Ausdrücke

Ebenso wie die Anweisungen sollen auch die Ausdrücke nur in Auszügen besprochen werden. Interessant sind hierbei vor allem die Verwendung des `void`-Ausdrucks, der Routinenaufruf und die Realisierung der Operatorüberlagerung.

Der `void`-Ausdruck

Alle verwendeten Attribute und Variablen werden in Sather mit `void` initialisiert. Dieser Ausdruck nimmt je nach Typ der Variablen unterschiedliche Werte an, z. B. `false` für boolesche Variablen und `0` für integer-Variablen. Allgemein erhält man den `void`-Wert eines Value-Objekts, indem man rekursiv jedes Attribut mit `void` besetzt. Für Variablen eines Referenz- oder abstrakten Typs ist `void` ein ausgezeichneter Wert, der besagt, daß kein Objekt referenziert wird. Er ist in dieser Funktion z. B. mit `nil` in Pascal vergleichbar.

Der Routinenaufruf

Syntaktisch bestehen Methodenaufrufe in Sather wie in anderen objektorientierten Sprachen aus einem Empfänger, dem Namen der Routine und einer optionalen Parameterliste. Zur Spezifikation des Empfängers stehen dabei zwei Möglichkeiten zur Wahl. Zum einen kann der Empfänger durch einen Ausdruck angegeben werden. Dieser wird ausgewertet und der Aufruf an das Ergebnisobjekt gerichtet.

Zum anderen kann der Aufruf auch direkt an eine konkrete Klasse gerichtet werden. Dabei wird die Routine im entspechenden `void`-Objekt dieser Klasse aufgerufen. Diese Lösung ist für Value-Klassen leicht zu handhaben, da hier mit dem `void`-Objekt ein Objekt kreiert wird. Für Referenzklassen beinhaltet der direkte Aufruf jedoch eine enorme Fehlerquelle, da die Methode eines Objekts ausgeführt wird, das im Speicher überhaupt nicht existiert. Der schreibende Zugriff auf ein Attribut führt daher sofort zum Programmabsturz.

Die Operatorüberlagerung

Arithmetische Operatoren oder Vergleichsoperatoren sind in Sather lediglich eine alternative Notation für Methodenaufrufe. Ein Ausdruck der Form $a + b$ wird beispielsweise in `a.plus(b)` transformiert. Dadurch wird die Überlagerung von Operatoren sehr einfach: Es genügt, eine Routine mit entsprechender Signatur zu definieren. Eine Tabelle der Operatoren mit den zugehörigen Prozedurnamen findet sich in [Omo94, S. 24]. Die Zuordnung kann auch der Parserbeschreibung in Anhang B.2.2 entnommen werden.

Neben den Operatoren existieren für eine Reihe weiterer Routinen syntaktische Kurzformen. Besonders häufige Verwendung findet der Ausdruck `#CLASS(parameterlist)` zum Kreieren eines Objekts. Er wird intern in den direkten Aufruf `CLASS::create(parameterlist)` umgeformt.

4.1.5 Annotierung und Ausnahmebehandlung

Wie bereits eingangs erwähnt, stellt Sather dem Programmierer verschiedene Möglichkeiten der Annotierung zur Verfügung. Neben Zusicherungen auf Anweisungsebene ist auch die Spezifikation von Vor- und Nachbedingungen für Routinen und Iteratoren auf Prozedurebene möglich. Die Formulierung von Objektinvarianten erfolgt mittels einer ausgezeichneten Routine, die einen booleschen Wert liefert. Diese Routine wird nach jedem öffentlichen Methodenaufruf ausgewertet. Wird `false` zurückgegeben, so wird die Ausführung des Programms abgebrochen. Eine Ausnahmebehandlung erfolgt hier nicht! Weitere Nachteile, wie die völlig fehlende Behandlung der Vererbung von Annotationen, wurden bereits in Abschnitt 1.4.1 genannt.

Hauptproblem der Ausnahmebehandlung in Sather ist der fehlende Zusammenhang zur Annotierung. Eine Ausnahmebehandlung kann nur dann erfolgen, wenn diese explizit oder vom System aufgerufen wird, nicht jedoch bei Verletzung von Annotationen. Sie ist daher im Kontext von *Design by Contract* sinnlos.

4.2 Definition des implementierten Ausschnitts

Um den Implementierungsaufwand in einem überschaubaren Rahmen zu halten, wird nicht der gesamte Sprachschatz Sathers vom Interpreter unterstützt. Es wird daher ein Ausschnitt der Sprache definiert, der folgende Bedingungen erfüllen muß:

- Der Ausschnitt muß alle typischen Merkmale einer objektorientierten Sprache aufweisen, um die Überlegungen und Ergebnisse der Arbeit auch auf andere Programmiersprachen übertragen zu können.

- Die implementierte Sprache muß den in Abschnitt 3.3 dargestellten Anforderungen an eine Basissprache genügen.
- Der Sprachschatz soll auf Wesentliches beschränkt sein, dem Programmierer aber trotzdem die Möglichkeit geben, auch größere Projekte zu realisieren, um eine Praxistauglichkeit zu erhalten.

Die entstehende Sprache muß also einerseits ohne allzu großen Aufwand implementiert werden können, soll andererseits jedoch umfangreich genug sein, um alle wichtigen Aspekte der Annotierung objektorientierter Programme demonstrieren zu können.

4.2.1 Bestandteile des Ausschnitts

Kürzungen des Sprachschatzes mußten vor allem im Bereich der Klassen und Klassenkomponenten vorgenommen werden, während Anweisungen und Ausdrücke nahezu vollständig implementiert wurden, um eine komfortable Programmierung zu ermöglichen.

- Auf externe und gebundene Klassen wurde vollständig verzichtet, ebenso wie auf parametrisierte Klassen. Die Behandlung dieser Konstrukte seitens des Interpreters gestaltet sich aufwendig, bringt jedoch keinerlei neue Erkenntnisse für die Annotierung.
- Klassenattribute und konstante Attriubte wurden ausgeklammert, da sie sich in der Annotierung nicht von Objektattributen unterscheiden.
- Iteratoren und die zugehörigen Anweisungen wurden wegen ihrer Ähnlichkeit zu Routinen weggelassen.
- Da eine eigene Annotationssprache für Sather entwickelt wird, wurde auf alle bestehenden Möglichkeiten der Annotierung ebenso verzichtet wie auf die Mechanismen der Ausnahmebehandlung.
- Die vorgegebenen Bibliotheksklassen wurden nur teilweise implementiert. Eingespart wurden vor allem die Klassen zur Behandlung von Feldern und Fließkommaberechnungen. Die Grundtypen `BOOL`, `CHAR`, `INT` und `STR` stehen jedoch ebenso wie die Klasse der systemnahen Funktionen, `SYS`, zur Verfügung.

Den Kern der Sprache stellen die Referenz-, Value- und abstrakten Klassen mit ihren Objektattributen und Methoden dar. Wegen der Bereitstellung von Mechanismen der Vererbung, des *code-sharings* und des Polymorphismus, sowie wegen des umfangreichen Angebots an Anweisungen und Ausdrücken kann der beschriebene Ausschnitt als exemplarische objektorientierte Sprache bezeichnet werden. Der Transfer der Ergebnisse dieser Arbeit auf andere Sprachen dieser Klasse sollte daher keine großen Schwierigkeiten bereiten.

4.2.2 Abweichungen von der Sather-Spezifikation

Die Implementierung des Satherausschnitts weicht in einigen Punkten von den Vorgaben der Spezifikation in [Omo94] ab. Dies geschah einerseits, um die Anforderungen aus Abschnitt 3.3 erfüllen zu können, andererseits, um die Programmierung des Interpreters zu vereinfachen.

Funktionale Routinen

Die wichtigste Veränderung ist die Einführung des Schlüsselworts `functional` zur Kennzeichnung von Signaturen und Routinen, deren Abarbeitung seiteneffektfrei verläuft. In funktionalen Routinen dürfen keine Veränderungen an Objekten im Speicher vorgenommen werden. Nicht zugelassen sind daher

- der Aufruf nicht funktionaler Routinen,
- die Kreierung von Referenzobjekten mittels `new` sowie
- die Veränderung von Referenzobjekten durch Manipulation der Attribute.

Aus der letzten Forderung läßt sich ableiten, daß die Schreibroutinen für Attribute von Referenzklassen nicht funktional sind, während die der Value-Klassen diese Eigenschaft aufweisen. Der schreibende Zugriff auf ein Attribut eines Value-Objekts führt immer zur Erzeugung eines neuen Objekts, weshalb das bestehende nicht verändert wird und somit keine Seiteneffekte auftreten können. Eine genaue Definition des Begriffs *funktionale Routine* erfolgt am Ende dieses Kapitels in Abschnitt 4.4.2.

Die Deklaration einer Signatur als `functional` wirkt sich auch auf die Vererbung aus. Schreibt eine abstrakte Klasse eine funktionale Signatur vor, so müssen alle abgeleiteten Klassen die Routine ebenfalls funktional gestalten. Der Begriff der Konformität zweier Routinen f und g aus Abschnitt 4.1.1 muß daher um einen Punkt erweitert werden. Die Signatur von f ist konform zu der von g, wenn zusätzlich gilt:

6. Ist g funktional, so muß auch f funktional sein.

Durch die Einführung der funktionalen Routinen erfüllt Sather alle genannten Anforderungen an eine Basissprache, da die Bibliotheksklasse `SYS` alle erwünschten Standardfunktionen bereitstellt (vgl. Anhang B.1).

Kleine Änderungen der Semantik

Kleine Abweichungen von der Sather-Spezifikation ermöglichen einen einfacheren Aufbau der abstrakten Syntax und ein besseres Zusammenspiel mit dem MAX System.

- Im Interpreter liefert jede Routine einen Rückgabewert. Fehlt bei der Deklaration die Angabe eines Rückgabetyps, so wird `INT` verwendet. Jede Routine, die keinen Wert zurückgibt, liefert intern den `void`-Wert. Dadurch muß auch die Forderung aufgehoben werden, daß nur Routinen, die keinen Wert zurückgeben, an Stelle einer Anweisung aufgerufen werden dürfen. Jeder Methodenaufruf ist nun zulässig. Ergebnisse, die nicht weiter verarbeitet werden, werden ignoriert.

- Der Verzicht auf eine `case`-Anweisung oder `elsif`-Zweige in der *abstrakten Syntax* führt zu einer übersichtlicheren Implementierung. `if`-Anweisungen mit `elsif`-Zweigen werden vom Parser in kaskadenartige `if-then-else`-Anweisungen transformiert, ebenso wie die `case`-Anweisung. Hierbei tritt jedoch das Problem auf, daß die Auswertung des untersuchten Ausdrucks nun bei jedem Vergleich erfolgt, statt wie üblich nur einmal am Beginn der Anweisung. Ein- und Ausgaben sollten daher beispielsweise nicht als Ausdruck in `case`-Anweisungen erscheinen. In der Regel ist es sicherer, den Ausdruck der `case`-Anweisung zuerst einer lokalen Variable zuzuweisen und dann diese mittels `case` zu inspizieren.

- Während die `STR`-Klasse in Sather als Referenzklasse angeboten wird, liegt sie im Interpreter als Value-Klasse vor. Grund dafür ist der MAX Datentyp `string`, auf den sich die Implementierung von `STR` abstützt, und der eine Value-Semantik hat.

- In Sather muß stets gewährleistet sein, daß die Vererbungsbeziehung zweier Klassen nur anhand der von ihnen erreichbaren Klassen überprüft werden kann. Wie im folgenden Abschnitt beschrieben, baut der Interpreter den gesamten Vererbungsgraphen auf und berechnet dessen transitive Hülle. Die obige Forderung ist deshalb nicht mehr notwendig, da die Beziehung zweier Klassen in der Vererbungshierarchie mit diesem Graphen leicht festgestellt werden kann.

4.2.3 Spezielle Implementierungsdetails

Im folgenden sollen einige Fragen der Implementierung besprochen werden, die zum Verständnis der MAX Spezifikation erforderlich sind. Es handelt sich hierbei um die vom Parser erledigte Transformation der konkreten in die abstrakte Syntax.

Die Symboltabelle

In der konkreten Syntax sieht der Zugriff auf lokale Variablen, Prozedurparameter und Attribute völlig identisch aus, obwohl auf Variablen und Parameter direkt, auf Attribute nur mittels Lese- und Schreibroutinen zugegriffen werden

kann. Zur Umsetzung in die abstrakte Syntax muß daher bekannt sein, welche Identifikatoren Variablen bzw. Parameter bezeichnen und welche Attribute.

Zu diesem Zweck führt der Parser eine temporäre Symboltabelle, in die alle Variablen- und Parameternamen bei ihrer Deklaration eingetragen und bei Verlassen des Gültigkeitsbereichs entfernt werden. Auf Bezeichner, die in der Tabelle enthalten sind, wird dann stets direkt, auf die nicht enthaltenen über die speziellen Routinen zugegriffen. Gleichzeitig wird so erreicht, daß lokale Variablen und Parameter gleichnamige Attribute und Routinen ohne Parameter verschatten.

Vererbung zwischen abstrakten Typen

Wie in Abschnitt 4.1.1 beschrieben, werden die Signaturen der Superklasse in die abgeleitete abstrakte Klasse übernommen. Um Konflikte zu vermeiden, besteht die Möglichkeit, geerbte Signaturen durch neu definierte zu überschreiben. Die Vererbung zwischen abstrakten Typen wird unmittelbar nach Abschluß der Syntaxanalyse behandelt, indem die Signaturen der betroffenen Klassen umkopiert werden.

Hierbei findet ein Typgraph Verwendung, der die gesamte Vererbungshierarchie in Form einer Adjazenzmatrix wiedergibt. Mit ihr ist es möglich, das Kopieren der Signaturen in der Hierarchie von oben nach unten zu steuern und Zyklen aufzudecken. Sie dient außerdem zur Überprüfung der Vererbungsbeziehung.

Diese Lösung hat den Vorteil, daß beim Suchen einer Signatur nicht immer die gesamte Vererbungshierarchie durchlaufen werden muß. Jede Klassendefinition enthält die Signaturen aller Operationen, die auf ihren Objekten ausgeführt werden dürfen.

Das Auflösen der Inkludierungsbestimmungen

Da beim Interpretieren des Sather-Programms keine Codeerzeugung stattfindet, ist auch die Beibehaltung der Inkludierungsbestimmungen nicht sinnvoll, da diese ja dem *code-sharing* dienen. Es liegt deshalb nahe, die inkludierten Programmstücke direkt in die Definitionen der Empfänger hineinzukopieren.

Dieser Mechanismus funktioniert ähnlich wie der oben beschriebene. Analog zum Typgraphen wird vom Parser ein Inkludierungsgraph in Form einer Adjazenzmatrix aufgebaut. Anhand dieses Graphen ist es dann möglich, das Kopieren der Attribute und Routinen zu steuern. Gleichzeitig werden einzelne Klassenkomponenten umbenannt oder übersprungen, falls dies in der Inkludierungsbestimmung vorgesehen ist. Hierbei wird nicht überprüft, ob inkludierte Routinen oder Attribute von explizit deklarierten überschrieben werden. Es werden grundsätzlich alle angeforderten Komponenten kopiert, was dann evtl. zur Ausgabe einer Fehlermeldung führt, da die inkludierte und die überschreibende Routine zueinander in Konflikt stehen. Dies ist zwar eine weitere Abweichung vom Sather-Standard, stellt aber durch die Möglichkeiten der Umbenennung keine wesentliche

Einschränkung dar.

Durch diese Behandlung des *code-sharings* kann beim Interpretieren immer davon ausgegangen werden, daß der Code wirklich in der Klassendefinition steht, wo er genutzt wird. Dieser Vorzug überwiegt den Nachteil eines größeren abstrakten Syntaxbaums bei weitem.

Die Komponenten zur Beschreibung einer Inkludierungsbestimmung in der abstrakten Syntax werden nur zum Zeitpunkt der Syntaxanalyse benötigt und während des Auflösens der Inkludierung entfernt. Bei Beginn der Attributauswertung dürfen sie nicht mehr im Baum enthalten sein.

Vordefinierte Klassen

Da vom Interpreter aus nicht auf Bibliotheksklassen zugegriffen werden kann, wird das Sather-Programm nach der syntaktischen Analyse um einige vordefinierte Klassen erweitert. Neben den vier Grundtypen `BOOL`, `CHAR`, `INT` und `STR` stehen die Klassen `SYS` mit systemnahen Routinen und `INOUT` mit Ein- und Ausgabefunktionen zur Verfügung.

An das Quellprogramm werden dabei nur die Definitionen der Klassen und Signaturen angehängt. Die Routinen selbst sind in C implementiert. Deshalb kann kein Sather-Code aus vordefinierten Klassen inkludiert werden. Die dynamische Auswertung des Programms steuert den Aufruf der C-Funktionen, falls ein Methodenaufruf an eine vordefinierte Klasse gerichtet wird. Ansonsten verhalten sich diese wie gewöhnliche Klassen. Ihre Definitionen können Anhang B.1 entnommen werden.

Da die Klassen zur Behandlung von Feldern nicht implementiert sind, kann die Übergabe von Parametern der Kommandozeile an die `main`-Routine nicht über Arrays erfolgen. Für jeden Parameter wird daher eine eigene Zeichenkette erzeugt und übergeben. Für unterschiedliche Parameterzahlen müssen daher verschiedene `main`-Routinen zur Verfügung stehen. Diese können die Parameterstrings mit Hilfe der Konvertierungsfunktionen der `STR`-Klasse in andere Basistypen überführen.

4.3 Ein Programmbeispiel

Um mit den Elementen der Sprache Sather vertraut zu werden und um die Programmbeispiele zur Annotationssprache verstehen zu können, wird im folgenden das Sather-Programm aus Abbildung 4.1 besprochen, das einige der wichtigsten Konstrukte dieser Sprache zeigt. Die Schlüsselwörter von Sather sind dabei fett gedruckt.

Das Programm besteht lediglich aus einer Klasse. Diese stellt eine Implementierung des Datentyps `NAT` vor. Als einzige Operation auf den natürlichen Zahlen ist die Fakultätsfunktion definiert. Außerdem steht ein Konstruktor zur Erzeugung und Initialisierung eines Objekts zur Verfügung. Die Funktion `main` steuert

```
(1)  value class NAT is
(2)     private attr x: INT;

(3)     create(n: INT): SAME is
(4)        r: SAME;
(5)        return r.x(n)
(6)     end; -- create

(7)     fac: SAME is
(8)        r: INT := 1;
(9)        n: INT := x;

(10)       if n > 1 then
(11)          loop
(12)             r := r * n;
(13)             n := n - 1;
(14)             while!(n > 1)
(15)          end -- loop
(16)       end; -- if

(17)       return create(r);
(18)    end; -- fac

(19)    main(a: STR): INT is
(20)       n: INT := a.atoi;
(21)       if n >= 0 and n < 11
(22)       then return #NAT(n).fac.x
(23)       else
(24)          INOUT::write_string("Parameter out of range").nl;
(25)          return -1
(26)       end -- if
(27)    end -- main
(28) end -- NAT
```

Abbildung 4.1: Sather Programmbeispiel

lediglich die Parameterübergabe von der Kommandozeile.

Da sich die Implementierung von `NAT` auf dem vordefinierten Typ `INT` abstützt, wurde sie ebenfalls als Value-Klasse realisiert. Der Konstruktor `create` zeigt den Gebrauch der Schreibroutinen für Attribute. Der Aufruf in Zeile 5 bewirkt die Erzeugung eines neuen `NAT`-Objekts, dessen einziges Attribut auf den Wert von `n` gesetzt ist.

An der Funktion `fac` läßt sich die korrekte Verwendung der `loop`-Anweisung erkennen. Der Rumpf der Schleife wird unendlich oft wiederholt, kann jedoch durch Aufruf des `while!`-Iterators vorzeitig verlassen werden, wenn dessen Bedingung `false` ergibt.

Die Routine `main`, mit der die Ausführung des Programms beginnt, verdeutlicht den Einsatz der vordefinierten Klassen. Ein Parameter, der beim Programmaufruf in der Kommandozeile angegeben wurde, wird der Hauptroutine als Zeichenkette übergeben. Mittels der vordefinierten Methode `atoi` kann dieser in einen `INT`-Wert umgewandelt werden. Liegt dieser Wert innerhalb der erlaubten Grenzen, so wird in Zeile 22 ein `NAT`-Objekt erzeut und dessen Fakultät berechnet. Liegt der Parameter nicht im erwünschten Bereich, so wird mit einem direkten Aufruf von Routinen der vordefinierten Klasse `INOUT` eine Fehlermeldung ausgegeben (vgl. Zeile 24).

4.4 Beschreibung der Semantik

Wie bereits eingangs erwähnt, ist die Kenntnis der Semantik von Sather für das Verständnis der Semantik der Annotationssprache unumgänglich. Im folgenden Abschnitt werden deshalb sowohl das Spezifikationsverfahren des MAX Systems als auch die operationelle Semantik der für die Annotierung wesentlichen Sather-Konstrukte dargestellt.

4.4.1 Das Spezifikationsverfahren von MAX

Bei der Semantik einer Programmiersprache unterscheidet man zwischen *statischer* und *dynamischer Semantik*. Unter statischer Semantik versteht man die Bereiche, die zur Übersetzungszeit ausgewertet werden können, wie z. B. die Identifikation eines Bezeichners mit seiner Deklaration. Aspekte der Semantik, die das Verhalten zur Laufzeit betreffen, bezeichnet man als dynamische Semantik. Hierzu zählen die Regeln zur Ausführung von Anweisungen und zur Auswertung von Ausdrücken.

Statische Semantik

Im MAX System erfolgt die Beschreibung der statischen Semantik durch die Attributierung des Syntaxbaums. Dabei ist es möglich, einen *Kontrollflußgraphen* zu

erstellen, der die Abarbeitungsreihenfolge des Programms wiedergibt. Die Verarbeitung einer Programmzeile, hier Zeile 13 des obigen Programms verläuft, dann in folgenden Schritten:

Die Anweisung `n := n - 1` wird unter Verwendung der Informationen aus der Symboltabelle, in der `n` als lokale Variable erscheint, in die Form `n := n.minus(1)` überführt, und der abstrakte Syntaxbaum erstellt. Den verwendeten Ausschnitt der abstrakten Syntax gibt Abbildung 4.2 wieder, Abbildung 4.3 stellt den Syntaxbaum mit der Anreicherung um den Kontrollflußgraphen dar[1].

```
UsedId       ( Ident Int )
Assign_Stmt  ( Local_Expr Expr )
Expr         = Literal_Expr | Local_Expr | Call_Expr | ...
Expr_List    * Expr
Literal_Expr = Bool | Char | String | Int
Local_Expr   ( UsedId )
Call_Expr    ( Receiver UsedId Expr_List )
Receiver     = Expr | Typespec
```

Abbildung 4.2: Auszug aus der abstrakten Sather-Syntax

Die mit ∘ gekennzeichneten Stellen heißen Ausführungspunkte. Sie stellen die Knoten des Graphen dar. Für jeden Knoten des abstrakten Syntaxbaums können zwei Ausführungspunkte existieren, jeweils vor und nach diesem. Auf diesen Ausführungspunkten ist eine Nachfolgerrelation definiert. Diese Relation entspricht einem *depth-first*-Durchlauf durch den Syntaxbaum, wobei auf dem Weg nach unten die Punkte vor dem Baumknoten, auf dem Weg nach oben die Punkte nach diesem besucht werden.

Der Aufbau des Kontrollflußgraphen geschieht dann hauptsächlich dadurch, daß diejenigen Produktionen spezifiziert werden, bei denen der Baumdurchlauf von der Nachfolgerrelation abweicht. Man erkennt z. B. in Abbildung 4.3, daß nach dem ersten Besuch einer `Local_Expr` direkt nach dieser fortgefahren werden kann, ohne die Söhne im Baum besucht zu haben.

Außerdem existieren Funktionen, die zu jedem Baumknoten die Ausführungspunkte liefern, die vor und nach diesem liegen, sowie Funktionen, die zu jedem Ausführungspunkt den zugehörigen Knoten zurückgeben. Dadurch können für jeden Übergang im Kontrollflußgraphen Aktionen definiert werden, die beim Durchlaufen der Kante ausgeführt werden. Diese bezeichnet man als *Tasks*. Sie beschreiben das dynamische Verhalten des Programms. Ein Beispiel hierfür ist der Task `Move` in der oben angesprochenen Produktion des Beispiels, der dafür sorgt, daß

[1] siehe hierzu auch [PH94, S. 2]

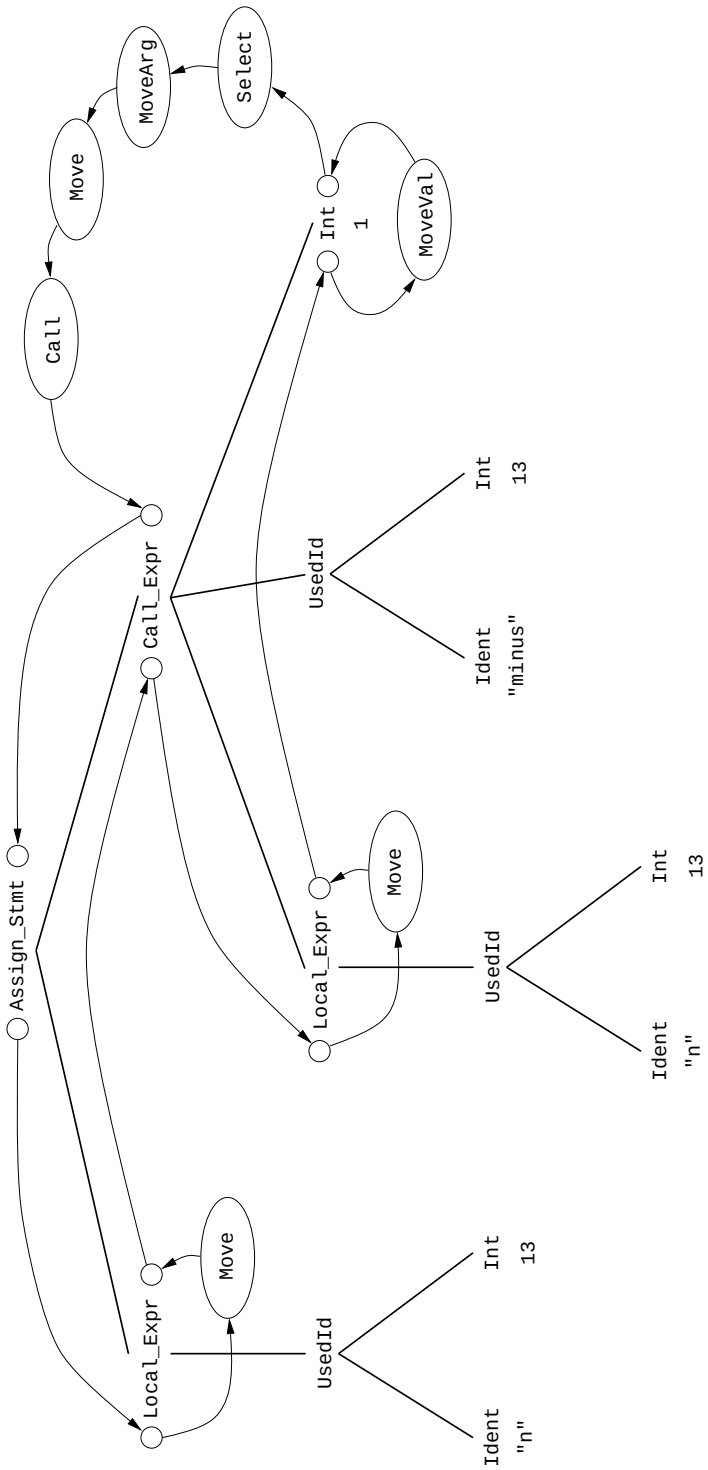

Abbildung 4.3: Syntaxbaum mit Kontrollflußgraph für `n := n - 1`

der Wert der lokalen Variablen `n` an der Stelle des angewandten Auftretens zur Verfügung steht.

Dynamische Semantik

Die Spezifikation der dynamischen Semantik geschieht in MAX über sog. *dynamische Algebren*. Im folgenden werden nur die wesentlichen Züge dieses Verfahrens vorgestellt. Eine ausführlichere Darstellung findet man in [PH94].

Unter einer *Algebra* versteht man eine Menge A — die sog. Trägermenge — und eine Folge von Funktionen, deren Argumente aus A^n stammen und die einen Wert in A liefern. Bei der Spezifikation durch dynamische Algebren werden die Zustände der Berechnung durch Algebren dargestellt. Die dynamische Algebra beschreibt dabei, in welcher Beziehung die Zustände zu ihren Folgezuständen stehen. Die Spezifikation muß also definieren, wie die Funktionen der Algebra manipuliert werden müssen, um Veränderungen des Zustands wiederzugeben. Durch diese Manipulationen der Funktionen entsteht eine Folge von Algebren, die die Zustandsänderungen beschreibt. Da die Werte der Funktionen während der Programmausführung verändert werden können, bezeichnet man diese als *dynamische Funktionen*.

Bezogen auf die MAX Spezifikation ist die Trägermenge gegeben durch die MAX-Basistypen und die Menge der in der abstrakten Syntax definierten Sorten. Die dynamischen Funktionen sind in der Spezifikation durch das Schlüsselwort `DYN` gekennzeichnet. Die Definition einer dynamischen Funktion hat die Form

`DYN ident(parameter) : value`,

wobei `ident` den Bezeichner der Funktion und `value` einen Wert aus der Trägermenge darstellt. Der Parameter ist optional. Mehrere Parameter sind nicht zulässig, können aber durch Einführung einer entsprechenden Tupelproduktion realisiert werden. Dynamische Funktionen ohne Parameter werden auch als *dynamische Variablen* bezeichnet.

Sowohl dynamische Funktionen als auch Variablen erhalten bei ihrer Definition einen Wert. Mit dieser Ausgangsbelegung stellen sie die erste Algebra in der Folge dar. Die Transformation der Algebren wird über bestimmte Regeln beschrieben, die in der MAX Spezifikation nach dem Schlüsselwort `RULES` erscheinen.

Diese Regeln geben für jeden Task des Kontrollflußgraphen an, welche Zustandsänderungen er bewirkt. Diese Änderungen erfolgen durch selektive Manipulation einzelner Funktionen an bestimmten Stellen. Syntaktisch haben sie die Form

`ident(parameter) := rechte Seite`.

Durch den Einsatz von `IF`- und `CASE`-Konstrukten ist auch eine bedingte Ausführung der Zuweisungen möglich.

Die Abarbeitung der Regeln wird in drei Schritten vollzogen[2]:

1. Die Bedingungen der Wächter werden ausgewertet. Unbewachte Zuweisungen werden wie bewachte behandelt, deren Bedingung `true` ergibt.

2. Die rechten Seiten aller Zuweisungen, deren Wächter `true` liefert, werden ausgewertet.

3. Die Zuweisungen werden ausgeführt. Im Fall widersprüchlicher Zuweisungen werden betroffene Regeln solange nicht-deterministisch entfernt, bis keine Widersprüche mehr auftreten.

Die Berechnung der Folge von Algebren terminiert, wenn für einen Task im Kontrollflußgraphen keine Transformationsregeln bekannt sind.

4.4.2 Die Semantik von Sather

Im folgenden Abschnitt soll neben den allgemeinen Bestandteilen der semantischen Beschreibung, wie den dynamischen Funktionen, vor allem die Semantik jener Sather-Konstrukte dargestellt werden, die für die Annotationssprache von Bedeutung sind. Wegen ihrer Verwendung in den Bedingungen der Annotationen spielen die Ausdrücke hierbei eine besondere Rolle.

Konstruktunabhängige Bestandteile

Der Kontrollflußgraph dient, wie im vorigen Abschnitt beschrieben, als Grundlage der dynamischen Auswertung eines Programms. Die Knoten dieses Graphs sind Ausführungspunkte, die Kanten werden durch eine Übergangsfunktion beschrieben. Bei jedem Übergang können Aktionen aufgerufen werden, die zur Transformation der aktuellen Zustandsalgebra in eine Folgealgebra führen.

Um die Formulierung der Übergangsfunktion zu vereinfachen, werden in der Implementierung die Ausführungspunkte ebenfalls zu den `Task`s gezählt. Beim Durchlaufen des Kontrollflußgraphen kann auf einen Punkt ein Folgepunkt oder ein Task folgen. Da nach einem Task ebenso mit einem Folgetask oder wiederum einem Ausführungspunkt fortgefahren werden kann, liegt diese Konstruktion nahe. Eine Beschreibung aller Tasks gibt Abbildung 4.4 wieder.

Die Tasks gliedern sich in `NonEndTasks` und `End`-Tasks. Letztere bezeichnen jene Aktionen, die zur Terminierung der Berechnung führen. Die `NonEndTasks` wiederum lassen sich in solche einteilen, die einen eindeutigen Nachfolger im Kontrollflußgraphen haben (`Seq_Task`), und solche, bei denen der Nachfolger erst dynamisch bestimmt werden kann. Die Definition der einzelnen Tasks und ihre Verwendung sind in Tabelle 4.1 zusammengefaßt.

[2] vgl. [PH94, S. 5f]

```
Task       = NonEndTask | End
NonEndTask = Seq_Task | CreateObj | MoveArg | Call | Return
           | Branch | Typecase | Logic | Invariant | Result | Error
Seq_Task   = Point | MoveVal | Select | Void_Test | Set | Move
End        = Terminated | Failed
```

Abbildung 4.4: Die Tasks des Interpreters

Die Übergangsfunktion zwischen den Ausführungspunkten des Programms stellt die Funktion `psucc` (vgl. B.2.3) dar. Sie definiert für jeden Ausführungspunkt einen Task, der entweder eine Aktion oder wiederum ein `Point` sein kann.

Task	Definition	Verwendung
`MoveVal`	`( Var Value Task )`	Wertzuweisung
`Select`	`( Call_Expr@ Task )`	Ermittlung des Empfängerobjekts
`Void_Test`	`( Expr@ Void_Test_Expr@ Task )`	Prüfung eines Ausdrucks auf `void`
`Set`	`( Set_Expr@ Task )`	Setzen eines Attributwerts
`Move`	`( Var Var Task )`	Zuweisung
`CreateObj`	`( Var Class@ Task )`	Kreierung eines Referenzobjekts
`MoveArg`	`( Call_Expr@ Task )`	Parameterübergabe
`Call`	`( Call_Expr@ )`	Routinenaufruf
`Return`	`()`	Ausführen der `return`-Anweisung
`Branch`	`( Expr@ Task Task )`	Dynamische Verzweigung
`Typecase`	`( Local_Expr@ Typespec@ Task Task )`	Ausführen der `typecase`-Anweisung
`Logic`	`( Expr@ Logic_Expr@ Task Task )`	Berechnung von `and` und `or`
`Invariant`	`()`	Prüfung von Objektinvarianten
`Result`	`( Expr@ )`	Ausgabe des Rückgabewerts des Programms
`Error`	`( String )`	Ausgabe einer Fehlermeldung
`Terminated`	`()`	Reguläres Programmende
`Failed`	`()`	Programmabbruch

Tabelle 4.1: Definition und Verwendung der Tasks

Die Algebra besteht aus der Trägermenge und dynamischen Funktionen. In der Trägermenge müssen die Datenelemente von Sather enthalten sein, um Operationen auf ihnen ausführen zu können.

Die Trägermenge Als Datenobjekte treten in Sather Referenz- und Value-Objekte auf. Um die Implementierung zu vereinfachen, werden die Basistypen

BOOL, CHAR, INT und STR auf die Basistypen des MAX Systems abgebildet, so daß die Definition eines Value wie in Abbildung 4.5 lautet. Ein Void_Value kommt zum Einsatz, wenn der Wert eines Ausdrucks nicht bestimmt werden kann, oder um das Fehlen einer Referenz anzuzeigen.

```
Value       = Bool | Int | Char | String | Object | Void_Value
Object      = RefObject | ValObject
RefObject   ( RefClass@ ObjectId )
ObjectId    = Int
ValObject   ( ValueClass@ Value_List )
Value_List  * Value
Void_Value  ()
```

Abbildung 4.5: Datenobjekte eines Sather-Programms

Der semantische Unterschied zwischen Referenz- und Value-Objekten kommt hier bereits in der Konstruktion der Datentypen zum Ausdruck. Ein Value-Objekt enthält neben dem Knoten der Klassendefinition eine Liste mit den Werten der Attribute. Dagegen besteht ein Referenzobjekt lediglich aus der Definitionsstelle und einer Objektnummer, die ähnlich wie eine Adresse im Speicher als Referenz verwendet werden kann.

Objekte können in Attributen, lokalen Variablen oder Prozedurparametern gespeichert werden. Außerdem enthält jeder Ausdruck, insbesondere der self-Ausdruck, einen Wert, sowie jede Routine ihren Rückgabewert, um diesen später weiterverarbeiten zu können.

Die Speicherung der Objekte geschieht durch eine dynamische Funktion. Als Speicherplätze finden für Attribute, Variablen, Parameter und Routinen deren Definitionsstellen im Syntaxbaum Verwendung. Der Wert von Ausdrücken wird ebenfalls an einen Baumknoten gebunden, während das self-Objekt in einer eigenen Variablen gespeichert wird. Diese Lokalisationen werden als *statische Variablen* bezeichnet. Da die Verwaltung nicht kellerartig ist, sind statische Variablen nicht eindeutig. Beispielsweise wird bei rekursiven Methodenaufrufen die gleiche lokale Variable in verschiedenen Inkarnationen verwendet. Zur Speicherung wird jedoch immer die gleiche statische Variable, nämlich die Definitionsstelle im Baum, herangezogen.

Statische Variablen müssen daher in eine eindeutige Form überführt werden. Dies geschieht durch die Bildung eines Tupels aus der statischen Variablen und einer Zusatzinformation. Diese Information ist für Attribute von Referenzobjekten die ObjectId. Attributwerte von Value-Objekten werden nicht an ihrer Definitionsstelle gespeichert, sondern stehen direkt in der Value_List des Objekts (vgl. Abb. 4.5). Alle anderen statischen Variablen werden durch die Angabe der Inkarnation einer Methode zu eindeutigen Lokalisationen. Auch die Inkarnatio-

nen werden nicht kellerartig verwaltet, sondern lediglich durchnummeriert. Dies führt zwar einerseits zu einem Mehraufwand an Speicher, da bereits abgeschlossene Inkarnationen weiterhin gespeichert werden, erleichtert aber andererseits die Implementierung des Interpreters erheblich.

Die entstehenden eindeutigen Speicherplätze werden *dynamische Variablen* genannt. Sie haben trotz der identischen Bezeichnung nichts mit den nullstelligen dynamischen Funktionen der Algebra zu tun. Die zugehörigen MAX Produktionen gibt Abbildung 4.6 wieder. Die Abbildung von statischen auf dynamische Variablen wird von der Funktion `dvar` erledigt. Für statische Variablen liefert sie die korrespondierende dynamische Variable, für dynamische Variablen ist sie die Identität. Ihre Definition kann Anhang B.2.3 entnommen werden.

```
Var        = StaticVar | DynamicVar
StaticVar  = Attr_Def@ | Typed_Ident@ | Expr@ | Self_Expr | Routine_Def@
DynamicVar = TempVar | ActVar | InstVar | SelfVar
TempVar    ( StaticVar IncarId )
ActVar     ( Typed_Ident@ IncarId )
InstVar    ( Attr_Def@ ObjectId )
SelfVar    ( IncarId )
IncarId    = Int
```

Abbildung 4.6: Die Speicherplätze von Sather-Variablen

Die dynamischen Funktionen Die dynamische Funktion zur Speicherung der Objekte heißt `cont` („*contents*“). Sie bildet dynamische Variablen auf ihre Werte ab. Alle Variablen werden mit `Void_Value` vorbelegt und erst an den entspechenden Stellen der Programmausführung initialisiert.

Um die Überführung von statischen in dynamische Variablen zu ermöglichen, müssen der Funktion `dvar` sowohl die aktuelle Inkarnationsnummer als auch die Objektnummer des `self`-Objekts übergeben werden. Für diesen Zweck exisitieren die dynamischen Funktionen `ci` („*current incarnation*“), `nextci` („*next current incarnation*“) und `no` („*next object identification*“). Eine Beschreibung der Funktionen erfolgt bei der Behandlung von Methodenaufruf und Objekterzeugung im nächsten Abschnitt. Die Definition ist ebenso wie für alle anderen dynamischen Funktionen Abbildung 4.7 zu entnehmen.

Die Funktion `ct` („*current task*“) enthält den aktuellen Task und spielt die zentrale Rolle beim Durchlaufen des Kontrollflußgraphen, da sie zur Auswahl der Transformationsregeln herangezogen wird. Die Initialisierung von `ct` bewirkt den Programmstart. Dieser wird in Abschnitt 4.4.2 näher erläutert.

Analog zu den Variablen ist teilweise auch eine Dynamisierung der Tasks notwendig. Dies ist der Fall, wenn bei einem Methodenaufruf die Position im

```
DYN cont( DynamicVar V )       Value:          Void_Value()
DYN ci()                       IncarId:        0
DYN nextci()                   IncarId:        1
DYN no()                       ObjectId:       1
DYN ct()                       Task:
    LET C == root().statement_list.fst.typespec.decl:
    LET R == root().statement_list.snd.receiver:
    IF is[C, _RefClass@]:
       CreateObj( R, C, R.after )
    ELSE
       MoveVal( R, C.build_void_value, R.after )
DYN rt( IncarId I )            DynamicTask:    nil()
DYN curr_routine( IncarId I )  Routine_Def@:   nil()
DYN pre_state( IncarId I )     Bool:           nil()
DYN dummy()                    Value:          0
```

Abbildung 4.7: Die dynamischen Funktionen der Algebra

Kontrollflußgraphen gespeichert werden soll, um die Bearbeitung nach Rückkehr aus der Methode an dieser Stelle fortzusetzen. Hier findet wiederum die Inkarnationsnummer Verwendung. Die dynamische Funktion zur Aufnahme eines dynamischen Tasks lautet `rt` („*return task*"). Auch die Funktionen `curr_routine` („*current routine*") und `pre_state` werden für die Bearbeitung von Methodenaufrufen verwendet, weshalb auch sie die Inkarnationsnummer als Parameter erwarten. `curr_routine` enthält die Definitionsstelle der zur jeweiligen Inkarnationsnummer gehörenden Routine. Sie wird beim Methodenaufruf und für den Zugriff auf den Rückgabewert benötigt. Der boolesche Wert von `pre_state` gibt an, wie nach der Überprüfung von Objektinvarianten fortgefahren werden soll. Eine detaillierte Darstellung folgt in Kapitel 5.

Die Funktion `dummy` dient zur Aufnahme der Rückgabewerte von Funktionen, die ihrer Seiteneffekte wegen aufgerufen werden. Dabei handelt es sich um C-Funktionen zur Ausgabe von Meldungen.

Die Semantik einzelner Konstrukte

Da zur Formulierung einzelner Bedingungen in Annotationen die Ausdrücke der Basissprache herangezogen werden, wird im folgenden Abschnitt vor allem deren Semantik erläutert. Die verschiedenen Ausdrücke von Sather gehen aus der Produktion in Abbildung 4.8 hervor, ihre Bedeutung wird in Tabelle 4.2 kurz beschrieben.

```
Expr = Self_Expr | Local_Expr | Void_Expr | New_Expr | Literal_Expr
     | Call_Expr | Logic_Expr | Attr_Expr | Set_Expr | Void_Test_Expr
```

Abbildung 4.8: Die Ausdrücke von Sather

Ausdruck	Definition	Verwendung
`Literal_Expr`	`= Bool | Char | String | Int`	Konstanten
`Self_Expr`	`()`	Aktuelles Objekt
`Local_Expr`	`(UsedId)`	Lokale Variablen und Parameter
`Attr_Expr`	`()`	Lesender Attributzugriff
`Void_Expr`	`()`	`void`-Wert
`Void_Test_Expr`	`(Expr)`	Test auf `void`-Wert
`New_Expr`	`()`	Erzeugung von Referenzobjekten
`Set_Expr`	`(Local_Expr)`	Schreibender Attributzugriff
`Logic_Expr`	`(Expr Operator Expr)`	Logisches *und* und *oder*
`Call_Expr`	`(Receiver UsedId Expr_List)`	Methodenaufruf

Tabelle 4.2: Verwendung der Ausdrücke

Zugriff auf Konstanten und Speicherplätze Einige Ausdrücke dienen dem Zugriff auf Werte, die bereits in Speicherplätzen stehen und nicht erst berechnet werden müssen. Hierzu zählen die Ausdrücke zur Anwendung des `self`-Objekts, von lokalen Variablen, Prozedurparametern, Konstanten sowie Attribut- und `void`-Werten. Ihre dynamische Semantik stimmt in weiten Teilen überein.

Die Funktion `psucc`, die jedem Ausführungspunkt `P` einen Folgetask zuordnet, stellt im wesentlichen den Kontrollflußgraphen dar. Sie analysiert den zugehörigen Knoten `N` des Syntaxbaums und ermittelt je nachdem, ob sich `P` vor oder nach dem Knoten befindet, eine entsprechende Aktion oder den folgenden Ausführungspunkt.

Für die oben angesprochenen Ausdrücke wird bereits bei Erreichen des Ausführungspunkts vor dem Knoten eine Aktion eingeleitet, die den benötigten Wert an der aktuellen Stelle verfügbar macht. Den entsprechenden Auszug aus `psucc` gibt Abbildung 4.9 wieder.

Da im Interpreter Konstanten der Klassen `BOOL`, `CHAR`, `INT` und `STR` möglich sind, die alle auf die Basistypen von MAX abgebildet werden, ist es ausreichend, den vom Parser gelieferten Wert der Konstanten als `Value` der entsprechenden `Literal_Expr` zuzuordnen. Diese Aufgabe übernimmt die Aktion `MoveVal` (vgl. Abbildung 4.10).

Um die Transformationsregeln für die Algebra verständlicher und besser lesbar zu machen, werden zu Beginn eines Tasks häufig verwendete Ausdrücke mit `LET`-Anweisungen aufgeführt. MAX stellt diesen Mechanismus nicht zur Verfügung. Er dient hier lediglich der Definition von Abkürzungen. Obwohl ebenfalls nicht von MAX unterstützt, werden in den Bedingungen der `IF`-Anweisungen die Ope-

```
IF P = N.before:
   IF Literal_Expr@ N:    MoveVal (N, N.term, N.after)
    | Self_Expr@ N:       Move (N, Self_Expr(), N.after)
    | Local_Expr@<UID> N: Move (N, UID.decl, N.after)
    | Attr_Expr@ N:       Move (N, N.encl_rout.lbroth, N.after)
    | Void_Expr@ N:       N.void_task
   ...
```

Abbildung 4.9: Kontrollfluß für Zugriff auf Speicher

```
CASE ct().sort
IS _MoveVal THEN
  dvar(ct().t_var, ci(), 0).cont := ct().t_value
IS _Move THEN
  LET DDEST == dvar(ct().t_dest, ci(), 0):
     // Dynamische Variable des Ziels
  LET DEST  == dvar(ct().t_dest, ci(), 0).cont:
     // Inhalt des Ziels
  LET SRC   == dvar(ct().t_src, ci(), 0).cont:
     // Inhalt der Quelle
  LET SELF  == SelfVar(ci()).cont:
     // Inhalt des self-Objekts

  IF is[ct().t_src, _Attr_Def@] THEN
     CASE SELF.sort
     IS _Void_Value THEN
        DEST := Void_Value()
     IS _RefObject THEN
        DEST := dvar(ct().t_src, ci(), SELF.objid).cont
     IS _ValObject THEN
        DEST := subterm(ct().t_src.attr_id, SELF.t_value_list)
     END
  ELSE
     IF is[SRC, _Void_Value] AND is[DDEST, _ActVar] THEN
        DEST := build_void_value(DDEST.typed_ident.class)
     ELSE
        DEST := SRC
     FI
  FI
```

Abbildung 4.10: Regeln für den Speicherzugriff

ratoren AND und OR benutzt, um den Code zu straffen. Einige weitere syntaktische Änderungen führen zusätzlich zu einer erhöhten Lesbarkeit. Alle Regeln sind in der MAX-Syntax im Anhang B.2.3 zu finden. Funktionen, deren Bezeichner mit dem Präfix t_ beginnen, kennzeichnen Selektoren für den Zugriff auf die einzelnen Komponenten der Tasks.

Im Gegensatz zu Konstanten wird bei Self_Expr, Local_Expr und Attr_Expr auf Werte zurückgegriffen, die nicht vom Parser geliefert werden, sondern während der Programmausführung vom Interpreter berechnet wurden und deshalb bereits in der dynamischen Funktion cont enthalten sind. Diese können mittels Move angesprochen werden.

Im Falle eines Zugriffs auf ein Attribut wird dabei unterschieden, ob es sich um ein Attribut einer Value- oder Referenzklasse handelt. Bei Value-Objekten sind die Attributwerte nicht in cont gespeichert, sondern in der Value_List des Objekts. Im Fall eines Referenzobjekts wird überprüft, ob die Referenz überhaupt auf ein im Speicher (d. h. in cont) existierendes Objekt verweist, also nicht den Void_Value enthält. Ist die Quelle des Move-Tasks kein Attribut, so wird ihr Wert, falls er der Void_Value ist, durch den void-Wert vom Typ des Empfängers — einer lokalen Variable oder einem formalen Parameter — ersetzt. Eine Begründung für diese Vorgehensweise wird bei der Behandlung des Methodenaufrufs (Seite 49) gegeben. Im Regelfall erfolgt eine Kopie des Wertes von der Quelle auf den Zielspeicherplatz durch Veränderung der cont-Funktion an der entspechenden Stelle.

Der Wert des void-Ausdrucks muß je nach Kontext erst ermittelt werden. Die Funktion void_task (vgl. Anhang B.2.3) erzeugt ein Objekt vom geforderten Typ und weist dieses per MoveVal dem Baumknoten der Void_Expr zu. Das Pandon zu diesem Ausdruck, die Void_Test_Expr wird durch den gleichnamigen Task ausgewertet. Dieser prüft, ob der untersuchte Ausdruck den void-Wert enthält. Die Definition ist ebenfalls im Anhang zu finden.

Objekterzeugung und -manipulation Der Kontrollflußgraph für die Ausdrücke zur Objekterzeugung und -manipulation ist in Abbildung 4.11 dargestellt. Set_Expr ist wie die Attr_Expr ein Element der abstrakten Syntax, das die Funktion von Schreib- bzw. Leseroutinen beschreibt. Da sich Schreibroutinen für Attribute von Referenz- und Value-Objekten unterschiedlich verhalten, muß diese Unterscheidung auch bei den Regeln für die Set-Aktion getroffen werden (vgl. Abbildung 4.12).

Für Referenzobjekte erfolgt der schreibende Attributzugriff durch Veränderung der cont-Funktion für dessen Definitionsstelle. Falls die Referenz nicht auf ein Objekt zeigt, also gleich Void_Value ist, kann keine Änderung eines Objekts erfolgen. In beiden Fällen wird der Void_Value als Ergebnis geliefert.

```
IF P = N.after:
   IF Set_Expr@<N> S:  Set (S, S.after)
   ...
IF P = N.before:
   IF New_Expr@ N:     CreateObj (N, N.encl_class, N.after)
   ...
```

Abbildung 4.11: Kontrollfluß für Objekterzeugung und -manipulation

Da bestehende Value-Objekte nicht verändert werden können, sondern jede Manipulation die Erzeugung eines neuen Objekts herbeiführt, muß in den Regeln für `Set` ein `ValObject` kreiert werden, dessen `Value_List` den veränderten Attributwert enthält. Die Konstruktion dieser neuen Liste wird von der Funktion `set_value` erledigt.

```
CASE ct().sort
IS _Set THEN
  DEST  == dvar(ct().t_expr, ci(), 0).cont:
     // Inhalt der Set-Ausdrucks
  ATTR  == ct().t_expr.encl_rout.lbroth.lbroth:
     // Definitionsstelle des Attributs
  PAR   == dvar(ct().t_expr.local_expr, ci(), 0).cont:
     // Wert des Parameters der Schreibroutine
  CLASS == ct().t_expr.encl_class:
     // Umgebende Klasse des Attributs
  SELF  == SelfVar(ci()).cont:
     // Inhalt des self-Objekts

  IF is[CLASS, _RefClass@] THEN
     IF SELF # Void_Value() THEN
        dvar(ATTR, 0, SELF.objid).cont := PAR
     FI
     DEST := Void_Value()
  ELSE
     DEST := ValObject (CLASS,
     set_value (SELF.t_value_list, ATTR.attr_id, PAR))
  FI
```

Abbildung 4.12: Regeln für Attributveränderung

Value-Objekte können nicht explizit erzeugt werden. Sie entstehen bei der Auswertung von Konstanten, Parametern oder lokalen Variablen bzw. durch

schreibende Zugriffe auf bestehende Value-Objekte. Im Gegensatz dazu lassen sich Referenzobjekte mit dem `new`-Ausdruck erzeugen. Wie aus Abbildung 4.13 ersichtlich ist, muß dem Ausdruck dabei eine neue Referenz, bestehend aus dem Knoten der Klassendefinition und einer neuen Objektnummer, zugeordnet werden. Die weitere Aufgabe des `CreateObj`-Tasks liegt in der Initialisierung der Attribute des neuen Objekts.

```
CASE ct().sort
IS _CreateObj THEN
  LET FIRST == ct().t_class.class_element_list.fst:
     // Erstes Element der instanziierten Klasse

  no() := no() + 1
  dvar(ct().t_var, ci(), 0).cont := RefObject(ct().t_class, no())
  IF FIRST # nil() THEN
     ct() := init_attr (FIRST, no(), ct().succ)
  ELSE
     ct() := ct().succ
  FI
```

Abbildung 4.13: Regeln für Objekterzeugung

Die Initialisierung von Objekten erfolgt mit den Funktionen `build_void_value` und `init_attr`. Die erste berechnet für jede Klasse denjenigen Wert, mit dem ein entsprechendes Objekt vorbelegt werden muß. Dieser Wert ist für Referenzklassen der `Void_Value`, für Value-Klassen ein Objekt, in dem die Attribute rekursiv wieder von `build_void_value` initialisert werden. Die `void`-Werte der Basistypen sind explizit erwähnt (vgl. Anhang B.2.3).

Bei der Instanziierung von Referenzklassen wird mittels `init_attr` eine Folge von `MoveVal`-Tasks erzeugt, die jedem Attribut des neuen Objekts den mit `build_void_value` erzeugten Wert zuweisen.

Die verbleibenden beiden Ausdrücke des Sather-Sprachschatzes spielen eine herausragende Rolle bei der Formulierung von Annotationen. Es handelt sich hierbei um die logischen Verknüpfungen `and` und `or` sowie um den Methodenaufruf.

Logische Verknüpfung Der Kontrollfluß für `and` und `or`, die in der abstrakten Syntax als `Logic_Expr` dargestellt werden, ist etwas komplexer. Wie aus Abbildung 4.14 hervorgeht, werden nicht zuerst beide Argumente der Verknüpfung ausgewertet, sondern der Aufruf der `Logic`-Aktion erfolgt bereits nach Berechnung des ersten Arguments.

```
IF P = N.after:
   IF Logic_Expr@<N, O, E> L:  Logic (N, L, L.after, E.before)
    | Logic_Expr@<E, O, N> L:  Logic (N, L, L.after, L.after)
   ...
```

Abbildung 4.14: Kontrollfluß für logisches *und* und *oder*

Die Betrachtung der zugehörigen Transformationsregeln in Abbildung 4.15 zeigt, daß der zweite Ausdruck der Verknüpfung nur ausgewertet wird, falls er zur Findung des Ergebnisses benötigt wird. Nach Untersuchung des ersten Ausdrucks kann daher zum Ausführungspunkt vor dem zweiten Ausdruck oder nach der `Logic_Expr` verzweigt werden. Nach Auswertung des zweiten Arguments hingegen ist die Berechnung des logischen Ausdrucks abgeschlossen, weshalb die Programmausführung immer an der gleichen Stelle fortgesetzt wird.

```
CASE ct().sort
IS _Logic THEN
  LET OP   == ct().t_logic_expr.operator:
     // Logischer Operator
  LET EXPR == dvar(ct().t_expr, ci(), 0).cont:
     // Inhalt des Arguments der Verknüpfung
  LET RES  == dvar(ct().t_logic_expr, ci(), 0).cont:
     // Inhalt des Ergebnisses der Verknüpfung

  RES := EXPR
  IF is[OP, _And_Op@] THEN
     IF EXPR = true() THEN
        ct() := ct().t_else_task
     ELSE
        ct() := ct().t_then_task
     FI
  ELSE
     IF EXPR = true() THEN
        ct() := ct().t_then_task
     ELSE
        ct() := ct().t_else_task
     FI
  FI
```

Abbildung 4.15: Regeln für logisches *und* und *oder*

Die Beschreibung der Semantik zeigt, daß die Berechnung logischer Ausdrücke in Sather nicht strikt ist. Wie in Abschnitt 5.3.3 gezeigt wird, läßt sich dieser Umstand bei der Annotierung ausnutzen, um elegantere Formulierungen zu erhalten.

Methodenaufruf und -rückkehr Der komplexeste und zugleich bedeutendste Ausdruck in Sather ist der Methodenaufruf. Der Kontrollfluß wurde in Abbildung 4.3 bereits kurz dargestellt und soll jetzt detailliert besprochen werden. Gleichzeitig wird auch die Behandlung der `return`-Anweisung beschrieben, um den vollständigen Ablauf einer Routinenausführung erkennen zu können.

```
IF P = N.after:
   IF Call_Expr@<N, _, <E, *>>:  E.before
    | Call_Expr@<N, _, <>> CE:   Select(CE, Call(CE))
    | Call_Expr@<*, <*, N>> CE:  Select(CE, MoveArg(CE, Call(CE)))
    | Return_Stmt@<N>:           Move(N.encl_rout, N, Return())
   ...
```

Abbildung 4.16: Kontrollfluß für Methodenaufruf und `return`

In Sather kann jedem Methodenaufruf bereits statisch eine Routine oder abstrakte Signatur eindeutig zugeordnet werden. Zur Auswahl werden neben dem Bezeichner Anzahl und Typen der Parameter herangezogen. Werden für einen Aufruf keine oder mehrere passende Routinen bzw. Signaturen gefunden, so wird die semantische Analyse unter Ausgabe einer Fehlermeldung abgebrochen. Die Zuordnung des Aufrufs zur Definitionsstelle der Routine oder Signatur geschieht durch das Attribut `target` (vgl. Anhang B.2.3).

Die dynamische Behandlung einer `Call_Expr` beginnt mit der `Select`-Aktion (vgl. Abbildung 4.16). Falls das `target`-Attribut bereits eine Routine liefert, also *statisches Binden* vorliegt, wird diese lediglich in die dynamische Funktion `curr_routine` übernommen. Andernfalls muß die benötigte Routine erst dynamisch ermittelt werden (*dynamisches Binden*). Das Vererbungskonzept von Sather stellt sicher, daß jede abgeleitete Klasse genau eine zur korrespondierenden Signatur der Superklasse konforme Routine bereitstellt. Diese wird von der Funktion `select_rout` gefunden und `curr_routine` zugewiesen (vgl. Abbildung 4.17).

Wie aus der Kontrollflußfunktion `psucc` in Abbildung 4.16 zu entnehmen ist, wird die Bearbeitung nach dem `Select`-Task mit der `MoveArg`-Aktion fortgesetzt. Aufgabe dieser Aktion ist das Kopieren der aktuellen Parameter in die Speicherplätze der formalen Parameter der aufgerufenen Methode. `MoveArg` kann daher bei Routinen ohne Argumentliste entfallen.

Das Übertragen der aktuellen Parameter wird von der Funktion `move_arg` gesteuert. Diese erzeugt für jedes Argument eine `Move`-Aktion (vgl. Abbildung 4.10).

Jetzt läßt sich auch erkennen, warum die Regeln für `Move` eine spezielle Behandlung für das Kopieren eines `Void_Value` in eine `Act_Var` vorsehen. Wird `void` als aktueller Parameter bei einem Routinenaufruf benutzt, so kann sein Wert erst nach der dynamischen Ermittlung der aufgerufenen Methode, also nach der Ausführung der `Select`-Aktion, bestimmt werden, da erst dann der Typ dieses Parameters feststeht. Beim Kopieren des aktuellen Parameters `void` wird dieser durch den zum formalen Parameter passenden `void`-Wert ersetzt.

```
CASE ct().sort
IS _Select THEN
  LET TAR  == ct().t_call_expr.target:
     // Definition der aufgerufenen Routine oder Signatur
  LET CALL == ct().t_call_expr:
     // Die Call_Expr@
  LET CR   == curr_routine(nextci()):
     // Definition der aufgerufenen Routine

  IF is[TAR, _Routine_Def@] THEN
     CR := TAR
  ELSE
     CR := select_rout(CALL.ident, CALL.expr_list,
     dvar(CALL.receiver, ci(), 0).cont.type_of_value )
  FI

IS _MoveArg THEN
  LET ARG   == curr_routine(nextci()).argument_list.fst:
     // Erster formaler Parameter der aufgerufenen Routine
  LET FIRST == ct().t_call_expr.expr_list.fst:
     // Erster aktueller Parameter

  ct() := move_arg(FIRST, ARG, nextci(), ci(), ct().succ)
```

Abbildung 4.17: Regeln für Methodenauswahl und Parameterübergabe

Nachdem die Methodenauswahl durch `Select` und gegebenenfalls das Übertragen der Parameter mittels `MoveArg` erfolgt ist, wird der Aufruf der gewählten Routine durchgeführt. Die zugehörige Aktion, der `Call`-Task, ist in Abbildung 4.18 dargestellt.

Syntaktisch können als Empfänger eines Methodenaufrufs Objekte und Klassen auftreten. Handelt es sich beim `Receiver` um einen `Typespec` und damit um einen direkten Aufruf an eine Klasse, so wird zuerst der `void`-Wert dieser Klasse berechnet. Dieser ist bei Value-Klassen ein Objekt, bei Referenzklassen der `Void_Value` zur Kennzeichnung, daß kein Verweis auf ein exisitierendes Objekt

vorliegt. Dieser von `build_void_value` gelieferte Wert wird als `self`-Objekt der nächsten Inkarnation eingetragen. Ist der `Receiver` hingegen ein Ausdruck, so kann dessen Wert direkt als `self`-Objekt übernommen werden.

```
CASE ct().sort
IS _Call THEN
  LET CR    == curr_routine(nextci()):
     // Definition der aufgerufenen Routine
  LET CALL  == ct().t_call_expr:
     // Die Call_Expr@
  LET REC   == ct().t_call_expr.receiver:
     // Empfänger des Aufrufs
  LET CLASS == curr_routine(nextci()).encl_class.ident.term:
     // Umgebende Klasse der aufgerufenen Routine
  LET FIRST == curr_routine(nextci()).statement_list.fst:
     // Erste Anweisung der aufgerufenen Routine

  IF is[REC, _Typespec@] THEN
     SelfVar(nextci()).cont := build_void_value(REC.typeclass)
  ELSE
     SelfVar(nextci()).cont := dvar(REC, ci(), 0).cont
  FI

  IF CLASS.predecl = true() THEN
     dvar(CALL, ci(), 0).cont := eval_predecl(CLASS.idtos, CALL)
     ct() := CALL.after
  ELSE
     ct() := init_locals(FIRST, FIRST.before)
     rt(nextci()) := DynamicTask(CALL.after, ci())
     ci() := nextci()
     nextci() := nextci() + 1
  FI
```

Abbildung 4.18: Regeln für Sather-Methodenaufruf

Da die vordefinierten Klassen des Interpreters keine Implementierung enthalten, muß bei der Ausführung des Sprungs zwischen diesen und normalen Klassen unterschieden werden. Die Methodenausführung für vordefinierte Klassen erfolgt durch die C-Funktion `eval_predecl`. Sie liefert das Ergebnis der Methode als `Value` zurück, der direkt in den Speicherplatz der `Call_Expr` übernommen werden kann. Die Bearbeitung kann daher am Ausführungspunkt nach dem Methodenaufruf fortgesetzt werden.

Die Ausführung einer in Sather implementierten Routine gestaltet sich etwas aufwendiger: Der Ausführungspunkt, an dem das Programm nach der Rückkehr aus der Methode fortgesetzt werden soll, wird zusammen mit der aktuellen Inkarnationsnummer als `DynamicTask` in der Funktion `rt` gesichert. Anschließend werden die Inkarnationsnummern `ci` und `nextci` erhöht.

Bevor mit der Ausführung der ersten Anweisung der aufgerufenen Routine begonnen werden kann, müssen sämtliche lokalen Variablen dieser Methode mit ihrem `void`-Wert initialisiert werden. Diese Vorbelegung erfolgt durch eine Sequenz von `MoveVal`-Aktionen, die von der Funktion `init_locals` erzeugt wird. Abschließend erfolgt der Sprung zum Ausführungspunkt vor der ersten Anweisung der behandelten Routine.

```
CASE ct().sort
IS _Return THEN
  LET TASK == rt(ci()).dyn_task:
     // Dynamischer Task, zum Fortsetzen der Bearbeitung
  LET INC  == rt(ci()).incid:
     // Inkarnationsnummer der aufrufenden Routine
  LET CR   == curr_routine(ci()):
     // Aktuelle Routine

  ct() := TASK
  ci() := INC
  dvar(TASK.node, INC, 0).cont := dvar(CR, ci(), 0).cont
```

Abbildung 4.19: Regeln für Methodenrückkehr

Aus der Behandlung der `return`-Anweisung in Abbildung 4.16 erkennt man, daß deren Ausführung zwei Aktionen erfordert. Zum einen muß der Rückgabewert gesichert und zum anderen die Weiterführung der aufrufenden Routine ermöglicht werden.

Als Speicherplatz für den Rückgabewert dient die Definitionsstelle der Routine. Die Übertragung erfolgt wie bei allen Speicherzugriffen über eine `Move`-Aktion. Bei der Ausführung des `Return`-Tasks wird der Wert von dort geholt und dem Methodenaufruf zugewiesen. Diese Zwischenspeicherung ist nötig, um den Rückgabewert in Nachbedingungen ansprechen zu können.

Neben dem Weiterreichen des Ergebnisses erfolgt durch die Regeln der `Return`-Aktion auch das Herstellen einer konsistenten Situation zur weiteren Programmauswertung. Dazu wird die Inkarnationsnummer restauriert und die Ausführung an der in `rt` gespeicherten Stelle fortgesetzt (vgl. Abbildung 4.19).

Objektgleichheit

Wegen ihrer großen Bedeutung für die Formulierung von Bedingungen in Annotationen soll die Semantik der Objektgleichheit noch exakt definiert werden. Sie wird von der Routine `ob_eq` der vordefinierten Klasse `SYS` überprüft. Als Regel für die Objektgleichheit kann formuliert werden:

> Zwei Objekte sind gleich, wenn ihre MAX-Darstellungen als `Values` (vgl. Abbildung 4.5) gleich sind. Das bedeutet:
>
> 1. Sie gehören der gleichen Klasse an.
> 2. Gleiche Referenzobjekte tragen die gleiche Objektnummer. Die Werte der Attribute spielen keine Rolle.
> 3. Gleiche Value-Objekte haben gleiche Attributwerte, d. h. für je zwei korrespondierende Attributwerte gilt die Objektgleichheit.

Da diese Semantik bereits aus den Produktionen für `Values` folgt, kann die Objektgleichheit sehr einfach durch Anwendung des `eq`-Prädikats von MAX festgestellt werden.

Funktionale Methoden

Bei der Beschreibung der funktionalen Methoden wurden bisher die Begriffe *Zustand* und *Zustandsänderung* verwandt, ohne ihre genaue Bedeutung definiert zu haben.

> Im Zusammenhang mit funktionalen Methoden versteht man unter *Zustand* den Inhalt aller dynamischen Funktionen außer `ct`, `ci` und `nextci`, soweit dieser mit Inkarnationsnummern bis zur aktuellen `ci` und mit Objektnummern bis zu `no - 1` ansprechbar ist.

Daraus lassen sich mehrere Folgerungen für funktionale Methoden ableiten:

- Die Ausführung aller Aktionen, die diesen Zustand nicht verändern, insbesondere Methodenaufrufe, ist möglich.
- In funktionalen Methoden können lokale Variablen und Parameter benutzt und verändert werden, da ihre Speicherung mit höherer Inkarnationsnummer erfolgt und somit keine Auswirkungen auf den relevanten Teil der `cont`-Funktion haben.
- Die Erzeugung neuer Objekte mittels `new` ist nicht möglich, da der zugehörige `CreateObj`-Task die `no` verändert.

- Die Manipulation von Referenzobjekten ist nicht möglich, die `Set`-Aktion führt Änderungen der `cont`-Funktion durch. Die Schreibroutinen in Referenzklassen sind daher nicht funktional.

- Schreibroutinen für Attribute von Value-Objekten sind funktional, da die `cont`-Funktion hier keine entscheidenden Änderungen erfährt.

Die Routinen der Klasse `INOUT` zur Ein- und Ausgabe sind per definitionem nicht funktional.

Mit dieser Definition der funktionalen Methoden wird erreicht, daß die Programmausführung durch Annotationen nicht beeinflußt werden kann, da alle Objekte, die bereits vor Ausführung einer solchen Funktion existieren, hinterher unverändert vorliegen. Auch durch Ein- oder Ausgaben läßt sich das Verhalten eines Programms nicht verändern.

Da Sather über Mechanismen der Speicherbereinigung verfügt, wäre es prinzipiell möglich, die Kreierung neuer Referenzobjekte zuzulassen und diese zur Veränderung freizugeben, solange keine Referenzen auf diese Objekte nach außen gebracht werden. Dies führt jedoch dazu, daß die Schreibroutinen in Referenzklassen für manche Objekte funktional und für andere nicht funktional wären. Die Unterscheidung ließe sich nur dynamisch treffen. Um die Verständlichkeit der Programme nicht zu beeinträchtigen und den Aufwand der Implementierung nicht unnötig zu erhöhen, ist daher die Erzeugung von Referenzobjekten in funktionalen Methoden und in Annotationen generell untersagt.

Programmstart und Programmende

Der Programmstart erfolgt durch die Vorbelegung der dynamischen Funktion `ct` (vgl. Abbildung 4.7). Vor Beginn der Ausführung wird ein Objekt derjenigen Klasse erzeugt, die der Benutzer beim Aufruf des Interpreters spezifiziert. Fehlt diese Angabe, so wird die Klasse `MAIN` verwendet. Die Instanziierung geschieht für Referenzklassen durch die `CreateObj`-Aktion, bei Value-Klassen genügt es, den `void`-Wert zu berechnen.

Jedes Sather-Program wird vom Interpreter um folgende Anweisungen ergänzt:

```
r: MAIN;   -- bzw. spezifizierte Klasse
r.main (parlist);
```

Die lokale Variable `r` enthält das oben beschriebene Objekt der `MAIN`-Klasse. Die aktuellen Parameter für die `main`-Funktion werden aus der Kommandozeile gewonnen. Jeder Text nach der Angabe des Sather-Quelltextes wird als Argument für die Routine `main` interpretiert und dieser als Parameter vom Typ `STR` übergeben.

Nach der Kreierung des `MAIN`-Objekts wird das Programm, beginnend am Ausführungspunkt nach dem Empfänger des Methodenaufrufs, ausgeführt. Dieser Aufruf wird bereits wie ein regulärer Methodenaufruf behandelt.

Das Programmende ist erreicht, wenn der Kontrollfluß nach Ausführung der `main`-Routine zu deren Aufruf zurückkehrt. Dies wird durch folgende Zeilen der Funktion `psucc` erkannt:

```
IF P = N.after:
   IF Program@<_, <_, N>>:   Result (N)
```

Die `Result`-Aktion bewirkt die Ausgabe des Ergebnisses der `main`-Funktion durch den Interpreter. Für den darauffolgenden `Terminated`-Task existieren weder Transformationsregeln noch ein Nachfolger im Kontrollflußgraphen, die Programmausführung ist beendet.

Ein vorzeitiger Programmabbruch kann durch Aufruf der `Error`-Aktion erreicht werden. Sie führt zur Ausgabe einer Fehlermeldung und über den `Failed`-Task zum Ende der Bearbeitung.

Kapitel 5

SALSA — Die Annotationssprache

Die im vorangegangenen Abschnitt beschriebene objektorientierte Programmiersprache Sather wird im folgenden Kapitel um die *Sather Annotation Language for Specification* — SALSA — erweitert. Dabei wird der in Kapitel 3 dargestellte Entwurf einer Annotationssprache in die Praxis umgesetzt. Neben der Beschreibung der Sprachkonstrukte mit ihrer Syntax und Semantik wird das Programmbeispiel aus Abbildung 4.1 schrittweise um Annotationen angereichert, um den praktischen Einsatz der neuen Mechanismen zu demonstrieren.

5.1 Der Aufbau einer Annotation

In SALSA findet der in Abschnitt 3.1.1 vorgeschlagene Aufbau der Annotationen Verwendung. Demnach bestehen diese aus Listen einzelner Bedingungen, diese wiederum aus einem Identifikator und einem booleschen Ausdruck der Basissprache.

5.1.1 Einzelne Bedingungen

Einzelne Bedingungen der Annotationen haben in SALSA die Form

`ident : expression`,

wobei `ident` ein Bezeichner für die jeweilige Bedingung ist, und `expression` einen booleschen Ausdruck darstellt. Da für die Annotierung die Forderung gestellt worden ist, den Systemzustand nicht zu verändern, dürfen in `expression` nur diejenigen Sather-Konstrukte erscheinen, die auch in funktionalen Methoden erlaubt sind. Verboten sind also im wesentlichen der Aufruf nicht funktionaler Routinen und die Erzeugung von Referenzobjekten mittels `new`.

In der abstrakten Syntax werden die Einzelbedingungen als `Condition` (vgl. Abbildung 5.1) dargestellt. Ihr Aufbau entspricht dem der Bedingungen in der konkreten Syntax.

```
Condition_List   * Condition
Condition        ( DeclId Expr )
```

Abbildung 5.1: Die Produktionen für Bedingungen

5.1.2 Zusätzliche Ausdrücke

Zusätzliche Ausdrücke sind, wie bereits in Abschnitt 3.1 erwähnt, erforderlich, um bei der Annotierung z. B. Werte des Initialzustandes einer Prozedur oder deren Rückgabewert ansprechen zu können. Die Sather-Ausdrücke werden daher mit SALSA um den `initial`- und den `result`-Ausdruck erweitert. Analog erhält die Produktion für `Expr` (siehe Abbildung 4.8) zwei zusätzliche Varianten:

`Expr = ... | Initial_Expr | Result_Expr`

Der initial-Ausdruck wird verwendet, wenn Werte des Initialzustandes, wie z. B. die Belegung der Parameter, während der Prozedurausführung oder im Terminalzustand zur Annotierung benötigt werden. Seine konkrete Syntax lautet:

initial (`expression`),

die Darstellung in der abstrakten Syntax daher:

`Initial_Expr ( Expr )`

Eine geschachtelte Anwendung von `initial` ist nicht erlaubt, da das äußerste Auftreten ohnehin die Berechnung des gesamten Ausdrucks im Initialzustand bewirkt, die inneren `initial`-Ausdrücke also bedeutungslos sind.

Die Beschreibung der dynamischen Semantik von `initial` erfolgt am Ende dieses Kapitels, wenn alle Konstrukte, die die Abwicklung eines Methodenaufrufs beeinflussen, besprochen sind.

Der result-Ausdruck kann nur in den Nachbedingungen einer Methode benutzt werden und liefert deren Rückgabewert. In der konkreten Syntax wird er durch das Schlüsselwort **result** dargestellt, die abstrakte Syntax lautet:

`Result_Expr ()`

Selbstverständlich darf der `result`-Ausdruck nicht innerhalb eines `initial`-Ausdrucks erscheinen, da der Rückgabewert der Methode im Initialzustand nicht bekannt ist.

Die dynamische Semantik des `result`-Ausdrucks ist sehr einfach. Da bei Ausführung der `return`-Anweisung der Rückgabewert der Methode an deren Definitionsstelle gespeichert wird, genügt es, diesen an der Stelle des `result`-Ausdrucks zugänglich zu machen. Dies kann durch eine `Move`-Aktion erfolgen (vgl. Abbildung 5.2).

```
IF P = N.after:
   IF Result_Expr@ N:     Move (N, N.encl_rout, N.after)
   ...
```

Abbildung 5.2: Kontrollfluß für `result`-Ausdruck

Die Implikation ist eine zusätzliche Erweiterung der Sather-Ausdrücke. Ausdrücke der Form `a => b` werden vom Parser in `a.not or b` transformiert und somit durch bereits existierende Ausdrücke ersetzt. Die Implikation stellt daher lediglich eine Abkürzung dar, die die Lesbarkeit der Annotationen erhöht.

Abgeschwächte und verschärfte Bedingungen

Abgeschwächte und verschärfte Bedingungen erlauben eine besondere Behandlung der Vererbung seitens der Annotationssprache. Wie bereits in Abschnitt 3.2.2 beschrieben, erfordert die Vererbung auf Prozedurebene, aber auch die Vererbung auf Objektebene, das Erkennen von Abschwächungen und Verschärfungen zur Übersetzungszeit bzw. während der statischen Analyse.

In SALSA können daher nur die logische Und- und Oder-Verknüpfung verwendet werden, um derartige Effekte zu erzielen. Die in Abbildung 5.3 dargestellte Funktion `is_weaker` ermittelt, ob die Bedingung *E1* schwächer als *E2* ist, also *E1* aus *E2* folgt. Diese Definition läßt erkennen, daß sich hinter der Bezeichnung *schwächer* eigentlich *schwächer oder identisch* verbirgt.

Dies ist der Fall, wenn beide Bedingungen identisch sind. Ist andernfalls *E1* eine Oder-Verknüpfung, die in einem Zweig *E2* oder eine Abschwächung davon enthält, so ist *E1* schwächer als *E2*. Dies ist auch dann gegeben, wenn *E2* aus einer Und-Verknüpfung besteht, die in einem Zweig *E1* oder eine stärkere Bedingung enthält. Die Funktion `cond_equal`, die zwei Bedingungen auf Gleichheit überprüft, ist im Anhang B.2 zu finden.

```
FCT is_weaker(Expr@ E1, Expr@ E2) Bool:
    IF cond_equal(E1, E2) = true(): true()
    ELSE
        LET H1 ==
        IF Logic_Expr@ <O1, OP, O2> E1:
            IF is[OP, _Or_Op@]
            && {is_weaker(O1, E2) = true()
            || is_weaker(O2, E2) = true()}: true()
            ELSE false()
        ELSE false():
        IF H1 = true(): true()
        ELSE
            IF Logic_Expr@ <O1, OP, O2> E2:
                IF is[OP, _And_Op@]
                && {is_weaker(E1, O1) = true()
                || is_weaker(E1, O2) = true()}: true()
                ELSE false()
            ELSE false()
```

Abbildung 5.3: Erkennung von abgeschwächten und verschärften Bedingungen

5.1.3 Listen von Bedingungen

In der Regel bestehen Annotationen nicht nur aus einzelnen Bedingungen, sondern enthalten mehrere dieser Ausdrücke. In der konkreten Syntax sind diese durch Kommata voneinander getrennt. Ihre Repräsentation in der abstrakten Syntax gibt Abbildung 5.1 wieder. Innerhalb einer Bedingungsliste darf jeder Bezeichner nur einmal verwendet werden, um genaue Angaben über verletzte Annotationen zu ermöglichen.

Die Konformitätsrelation ist nicht nur auf Signaturen (vgl. Abschnitt 4.1.1) beschränkt, sondern kann auch auf Bedingungslisten definiert werden:

> Die Bedingungsliste B_1 ist konform zu B_2, wenn jede Bedingung aus B_2 direkt oder in verschärfter Form in B_1 enthalten ist.

B_1 stellt also in seiner Gesamtheit eine stärkere Forderung dar als B_2, da sie zum einen Bedingungen enthalten kann, die nicht in B_2 vorkommen, und zum anderen gemeinsame Bedingungen verschärft sein können.

Die dynamische Semantik von Bedingungen ist sehr einfach, da sie im wesentlichen nur die Auswertung des booleschen Ausdrucks enthält. Diese ist für

die Sather-Ausdrücke bereits in Kapitel 4 ausführlich dargestellt worden. Die dynamische Semantik des `result`-Ausdrucks ist in Abschnitt 5.1.2 dargestellt, die des `initial`-Ausdrucks wird später (Abschnitt 5.3.2) erklärt.

Der Kontrollflußgraph für Annotationen steuert die Auswertung dieser Ausdrücke. Beim Aufruf des Interpreters wird vom Benutzer spezifiziert, in welchem Umfang eine dynamische Abprüfung der Annotationen erfolgen soll. Zur Auswahl stehen fünf Stufen:

Level 0: Es werden keine dynamischen Abprüfungen vorgenommen.

Level 1: Die Vorbedingungen der Methoden werden geprüft.

Level 2: Vor- und Nachbedingungen werden geprüft.

Level 3: Vor- und Nachbedingungen sowie Objektinvarianten werden geprüft.

Level 4: Alle Bedingungen werden geprüft.

Fehlt die Angabe des Benutzers, so wird Stufe 1 gewählt. Sie stellt einen guten Kompromiß zwischen Ausführungsgeschwindigkeit und einem Mindestmaß an Kontrolle des Programmverhaltens dar.

Die gewählte Stufe wird vom Interpreter gespeichert. Mit Hilfe der C-Funktion `check_level` kann überprüft werden, ob der eingestellte Level größer oder gleich dem Parameter der Funktion ist.

```
IF P = N.after:
   IF Condition_List@<*> N: N.after_cond_list
    | Condition@<DID, N> C: Branch (N, C.after, Error (
    `"### " C.loc ": assertion " DID.ident.term.idtos " violated"'))
   ...
IF P = N.before:
   IF Condition_List@ N:    LET R == N.encl_rout:
                            LET LEVEL ==
                            IF N = R.pre:                  1
                             | N = R.post:                 2
                             | N = N.encl_class.invariant: 3
                            ELSE                           4:
                            IF check_level(LEVEL) = true(): P.next
                            ELSE                           N.after
   ...
```

Abbildung 5.4: Kontrollfluß für die Überprüfung von Bedingungen

Erreicht der Kontrollfluß den Ausführungspunkt vor einer Bedingungsliste, so wird überprüft, ob es sich dabei um die Liste einer Objektinvarianten, einer Vor-

oder Nachbedingung oder einer Zusicherung handelt (vgl. Abbildung 5.4). Mit der Funktion check_level wird dann entschieden, ob die Ausdrücke der Bedingungen ausgewertet werden sollen, oder ob die Liste vollständig übersprungen werden kann.

Soll eine dynamische Abprüfung der Bedingungen erfolgen, so verläuft der Kontrollfluß weiter zur ersten Bedingung der Liste, deren Ausdruck wie gewohnt ausgewertet wird. Nach der Berechnung erfolgt ein Aufruf der `Branch`-Aktion, die folgendermaßen aufgebaut ist:

`Branch ( Expr@ Task Task )`

Die Regeln für diesen Task sind in Abbildung 5.5 dargestellt. Sie bewirken je nachdem, ob der Wert des Ausdrucks `true` oder `false` ist, ein Fortfahren der Auswertung nach der aktuellen `Condition` oder einen Programmabbruch durch eine `Error`-Aktion. Dieser erfolgt unter Angabe des Bezeichners der verletzten Annotation und deren Position im Quelltext.

```
CASE ct().sort
IS _Branch THEN
  IF dvar(ct().t_expr, ci(), 0).cont = true() THEN
     ct() := ct().t_then_task
  ELSE
     ct() := ct().t_else_task
  FI
```

Abbildung 5.5: Regeln für `Branch`-Task

Ist die Annotation erfüllt, so werden die anderen Bedingungen der Liste in der Reihenfolge ihres Erscheinens im Quelltext ausgewertet. In korrekten Programmen sind stets alle Bedingungen erfüllt, so daß der Ausführungspunkt nach der `Condition_List` erreicht wird. Die Funktion `after_cond_list` liefert den nächsten Task der Programmausführung. Sie wird später im Zusammenhang mit dem Methodenaufruf besprochen (vgl. Abschnitt 5.3.2).

Die Auswertungsstrategie der Bedingungslisten zeigt, daß das Aufführen mehrerer Bedingungen einer logischen Und-Verknüpfung zwischen diesen entspricht, da nur bei Erfüllung aller Annotationen mit der Ausführung des Programms fortgefahren wird. Analog zum `and`-Ausdruck Sathers ist auch diese Verknüpfung nicht strikt. Bereits die erste verletzte Bedingung führt zum Programmabbruch. Die folgenden werden nicht mehr ausgewertet.

5.2 Objektinvarianten

Objektinvarianten werden in SALSA durch das Schlüsselwort `invariant` eingeleitet. Sie sind optional und stehen vor der Definition der Klassenelemente, also unmittelbar vor dem Schlüsselwort `is`. Invarianten können sowohl in konkreten als auch in abstrakten Klassen verwendet werden.

5.2.1 Vererbung

Wie in Abschnitt 3.2.3 erwähnt, müssen abgeleitete Klassen die Invarianten aller ihrer Superklassen enthalten. Objektinvarianten in abstrakten Klassen haben daher die gleiche Funktion wie die Definition abstrakter Signaturen: Sie geben den abgeleiteten Klassen gewisse Verpflichtungen auf, die von diesen erfüllt werden müssen. Abgeleitete Klassen dürfen die geerbten Bedingungen um eigene erweitern. Bei Vererbung gilt demnach: Die Objektinvarianten von abgeleiteten Klassen sind konform zu den Invarianten ihrer Superklassen.
Demzufolge ändert sich die Definition der Konformität von Schnittstellen aus Abschnitt 4.1.1:

> Unter einer Schnittstelle versteht man die Menge aller öffentlichen Methoden, die auf ein Objekt angewandt werden dürfen, **und die Objektinvarianten der zugehörigen Klasse**. Eine Schnittstelle S_1 aus Klasse K_1 ist konform zu S_2 aus K_2 , wenn es zu jeder Routine f_2 in S_2 eine einzige konforme Routine f_1 in S_1 gibt, **und die Objektinvarianten von K_1 konform zu denen von K_2 sind.**

Mit dieser Änderung kann die Grundregel der Vererbung beibehalten werden. Sie wird weiterhin von der Funktion `interface_conform` überprüft.

Geerbte Invarianten werden in SALSA nicht automatisch in die abgeleiteten Klassen eingefügt. Der Programmierer hat darauf zu achten, daß jede Bedingung einer Superklasse in der neuen Bedingungsliste enthalten ist. Dadurch wird ein höheres Maß an Dokumentation erreicht, da alle Invarianten einer Klasse explizit in deren Definition erscheinen.

5.2.2 Semantik

Die Bedingungen der Objektinvarianten müssen in jedem Zustand, in dem das Objekt für andere sichtbar ist, erfüllt sein. Das bedeutet, daß sich während des Aufenthalts innerhalb eines Objekts durchaus Inkonsistenzen ergeben können. Die dynamische Abprüfung der Bedingungen erfolgt daher beim Betreten und Verlassen des Objekts.

Die formale Semantik dieser Abprüfungen wird in Abschnitt 5.3.2 besprochen, da sie unmittelbar mit der dort dargestellten Semantik von Methodenaufrufen verknüpft ist.

5.2.3 Beispiel

In Abschnitt 4.3 wurde bereits ein Beispielprogramm vorgestellt, das eine Implementierung der Klasse `NAT` mit der Fakultätsberechnung als einziger Operation enthält (vgl. Abbildung 4.1). Dieses Programm wird während der Besprechung der SALSA-Konstrukte schrittweise annotiert.

Die Klasse `NAT` soll die Menge der natürlichen Zahlen N_0 darstellen. Um den Umfang des Beispiels gering zu halten, stützt sich die Implementierung auf die Klasse `INT` ab, so daß z. B. Vergleichs- und arithmetische Operationen nicht bereitgestellt werden müssen.

Da das Attribut `x` der Klasse `NAT` vom Typ `INT` ist, ist es prinzipiell möglich, auch negative Werte als `NAT` darzustellen. Dies kann durch die Einführung einer entsprechenden Invariante verhindert werden. Die Klassendefinition ändert sich demnach, wie in Abbildung 5.6 dargestellt. Annotationen sind dort fett gedruckt.

```
value class NAT
invariant
   is_nat: x >= 0
is
   private attr x: INT;
   ...
```

Abbildung 5.6: Programmbeispiel mit Objektinvariante

Bei der Formulierung von Objektinvarianten ist darauf zu achten, daß die `void`-Belegung des Objekts möglichst nicht zu einer Verletzung führen sollte. Wollte man beispielsweise in `NAT` nur positive Zahlen zulassen, sollte die Invariante dennoch nicht `is_pos: x > 0` lauten, denn jeder direkte Methodenaufruf an die Klasse, wie er vor allem bei der Objekterzeugung (`NAT::create(n)`) üblich ist, führt zur Erzeugung des `void`-Wertes. Die gewünschte Operation kann jedoch nicht ausgeführt werden, wenn dieser Wert (im Fall von `NAT` der Wert 0) die Bedingungen der Invariante verletzt.

Die verschärfte Bedingung `is_pos: x > 0` müßte daher als Vor- und Nachbedingung formuliert werden, um den Konstruktor `create` von der Annotierung ausnehmen zu können.

5.3 Vor- und Nachbedingungen

Die Annotierung von Routinen erfolgt in SALSA durch Vor- und Nachbedingungen. Beide stehen in der Definition der Methode vor deren Rumpf. Vorbedingungen werden durch das Schlüsselwort `requires`, Nachbedingungen durch `ensures` eingeleitet. Beide Möglichkeiten der Annotierung sind optional, fehlende Angaben

werden wie Bedingungen behandelt, die stets `true` liefern. In den Ausdrücken der Nachbedingungen dürfen die SALSA-Ausdrücke `initial` und `result` verwendet werden.

5.3.1 Vererbung

Wie bereits mehrfach erwähnt, besteht bei der Vererbung die Möglichkeit, Vorbedingungen abzuschwächen und Nachbedingungen zu verschärfen. Dies bedeutet, daß der Begriff der Konformität zweier Routinen aus Abschnitt 4.1.1 nochmals um zwei Punkte erweitert werden muß:

7. Die Liste der Vorbedingungen von g ist konform zu der von f.

8. Die Liste der Nachbedingungen von f ist konform zu der von g.

Die Überprüfung dieser Bedingungen wird weiterhin von der Funktion `rout_conform` vorgenommen. Da die formalen Parameter einer Signatur in der abgeleiteten Klasse und der Superklasse unterschiedliche Bezeichner haben können, müssen zwei Bedingungen nicht wörtlich identisch sein, um die gleiche Bedeutung zu haben. Dieser Aspekt wird von `rout_conform` berücksichtigt.

5.3.2 Semantik

Im folgenden Abschnitt wird ein großer Teil der Behandlung von Methodenaufrufen nochmals beschrieben, da sich hier durch den Einsatz von Objektinvarianten, Vor- und Nachbedingungen und den neuen SALSA-Ausdrücken wesentliche Veränderungen ergeben.

Den Ablauf eines Methodenaufrufs eines annotierten Programms gibt Abbildung 5.7 wieder. Der Initialzustand ist erst erreicht, wenn die Vorbelegung der lokalen Variablen erfolgt ist. Dadurch können diese auch in `initial`-Ausdrücken verwendet werden.

Wie aus Abbildung 3.1 ersichtlich ist, wird beim Aufruf einer Methode eines anderen Objektes zuerst das Objekt und dann die Methode betreten. Daher erfolgt in SALSA die Prüfung der Objektinvarianten beim Betreten eines Objekts vor der Prüfung der Vorbedingungen. Beim Verlassen werden umgekehrt zuerst die Nachbedingungen und dann die Invarianten untersucht. Dies spiegelt wider, daß Methoden Teil eines Objekts sind, Objekte diese also umgeben.

Der Methodenaufruf

Bei der formalen Beschreibung der Semantik des Methodenaufrufs ist nur der dynamische Teil von Interesse. Statische Aspekte, wie die Methodenauswahl, werden durch die Annotierung nicht beeinflußt. Auch die dynamische Methodenauswahl der `Select`-Aktion und das Kopieren der aktuellen Parameter bleiben

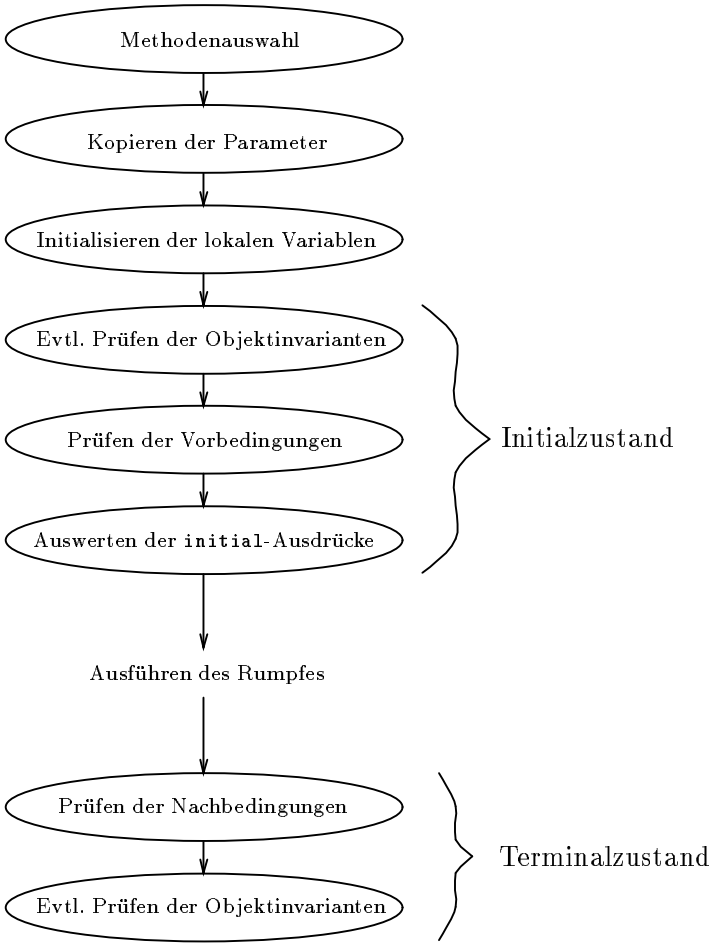

Abbildung 5.7: Behandlung eines Methodenaufrufs in SALSA

```
CASE ct().sort
IS _Call THEN
  LET CR    == curr_routine(nextci()):
     // Definition der aufgerufenen Routine
  LET CALL  == ct().t_call_expr:
     // Die Call_Expr@
  LET REC   == ct().t_call_expr.receiver:
     // Empfänger des Aufrufs
  LET CLASS == curr_routine(nextci()).encl_class.ident.term:
     // Umgebende Klasse der aufgerufenen Routine
  LET FIRST == curr_routine(nextci()).statement_list.fst:
     // Erste Anweisung der aufgerufenen Routine
  LET PRE == curr_routine(nextci()).pre
     // Vorbedingungen der Routine
  LET INV == curr_routine(nextci()).encl_class.invariant
     // Objektinvarianten von CLASS

  IF is[REC, _Typespec@] THEN
     SelfVar(nextci()).cont := build_void_value(REC.typeclass)
  ELSE
     SelfVar(nextci()).cont := dvar(REC, ci(), 0).cont
  FI

  IF CLASS.predecl = true() THEN
     dvar(CALL, ci(), 0).cont := eval_predecl(CLASS.idtos, CALL)
     ct() := CALL.after
  ELSE
     IF is[REC, _Self_Expr@] THEN
        ct() := init_locals(FIRST, PRE.before)
     ELSE
        pre_state(nextci()) := true()
        ct() := init_locals(FIRST, INV.before)
     FI
     rt(nextci()) := DynamicTask(CALL.after, ci())
     ci() := nextci()
     nextci() := nextci() + 1
  FI
```

Abbildung 5.8: Regeln für SALSA-Methodenaufruf

unverändert, so daß es ausreicht, mit der Betrachtung des `Call`-Tasks zu beginnen.

Abbildung 5.8 zeigt die Regeln für die `Call`-Aktion in SALSA. Die Veränderungen gegenüber dem Sather-Methodenaufruf (vgl. Abbildung 4.18) sind fett gedruckt. Unverändert ist die Behandlung der vordefinierten Klassen. Die Auswertung ihrer Annotierung wird wie die Ausführung des Rumpfes von der C-Funktion `eval_predecl` übernommen.

Für benutzerdefinierte Klassen wird in dem Fall, daß der Empfänger des Methodenaufrufs das `self`-Objekt ist, nach der Initialisierung der lokalen Variablen unmittelbar mit der Auswertung der Vorbedingungen fortgefahren. Andernfalls wird zur Auswertung der Objektinvarianten des Empfängerobjekts verzweigt.

Die dynamische Funktion `pre_state` wird auf `true` gesetzt. Sie wird benötigt, um nach der Abarbeitung der Invarianten entscheiden zu können, ob sich die Programmausführung im Initial- oder im Terminalzustand befindet.

Auswertung der Objektinvarianten

Die Auswertung der Objektinvarianten geschieht, wie bereits oben geschildert, durch Berechnung der Ausdrücke und Anwendung des `Branch`-Tasks. Nach der vollständigen Überprüfung aller Bedingungen steuert die Funktion `after_cond_list` (vgl. Abbildung 5.9) das weitere Vorgehen.

```
FCT after_cond_list(Condition_List@ CL) Task:
    LET R == CL.encl_rout:
    IF CL = R.pre:                      next_initial (CL.after, R)
     | CL = R.post:                     Return ()
     | CL = CL.encl_class.invariant:    Invariant ()
    ELSE CL.after.next
```

Abbildung 5.9: Programmfortsetzung nach Überprüfung der Annotationen

Die Aktion `Invariant` wird stets nach Abarbeitung der Objektinvarianten aufgerufen. Wie aus Abbildung 5.10 ersichtlich ist, nutzt sie die von `Call` versorgte Funktion `pre_state`. Liefert diese den Wert `true`, so befindet sich die Programmausführung im Initialzustand und muß mit der Auswertung der Vorbedingungen fortgesetzt werden. Andernfalls ist die Behandlung des Methodenaufrufs abgeschlossen. `Invariant` übernimmt für diesen Fall die Regeln für die Methodenrückkehr, die in Sather von der `Return`-Aktion ausgeführt wurden.

```
CASE ct().sort
IS _Invariant THEN
  LET PRE  == curr_routine(ci()).pre
     // Vorbedingungen der Routine
  LET TASK == rt(ci()).dyn_task:
     // Dynamischer Task, zum Fortsetzen der Bearbeitung
  LET INC  == rt(ci()).incid:
     // Inkarnationsnummer der aufrufenden Routine
  LET CR   == curr_routine(ci()):
     // Aktuelle Routine

  IF pre_state(ci()) = true() THEN
     ct() := PRE.before
  ELSE
     ct() := TASK
     ci() := INC
     dvar(TASK.node, INC, 0).cont := dvar(CR, ci(), 0).cont
  FI
```

Abbildung 5.10: Regeln für Invariantenbehandlung

Auswertung der Vorbedingungen und Initialausdrücke

Die Auswertung der Vorbedingungen geschieht analog. Anschließend müssen jedoch als letzte Aktion im Initialzustand die Werte der `initial`-Ausdrücke berechnet werden. Die in Abbildung 5.11 dargestellte Funktion `next_initial` durchläuft den abstrakten Syntaxbaum und veranlaßt die Berechnung der Initialausdrücke.

```
FCT next_initial(Point P, Routine_Def@ R) Task:
    LET N == P.node:
    IF {check_level(4) = true() && N.encl_rout = R}
    || {check_level(2) = true() &&
       {is[N, _Condition_List@] || N.encl_condition # nil()}}:
       IF is[N, _Initial_Expr@] && P = N.before: N.expr.before
       ELSE next_initial(P.next, R)
    ELSE
       R.statement_list.fst.before
```

Abbildung 5.11: Steuerung der Auswertung der Initialausdrücke

Da dieser Durchlauf äußerst zeitraubend ist, wird er nur in dem Maß ausgeführt, wie die Initialausdrücke tatsächlich benötigt werden. Das bedeutet, daß

der Rumpf der Routine nur betrachtet wird, wenn die Abprüfung der Zusicherungen gefordert wird. Gleiches gilt für die Nachbedingungen. Nachdem alle benötigten `initial`-Ausdrücke berechnet wurden, wird die Auswertung am Ausführungspunkt vor der ersten Anweisung des Methodenrumpfes fortgeführt.

Den zugehörigen Auszug aus der Kontrollflußfunktion `psucc` zeigt Abbildung 5.12. Nach der Berechnung eines als `initial` deklarierten Ausdrucks wird dessen Wert mit einer `Move`-Aktion an der Stelle des Initialausdrucks gespeichert und die Ausführung von `next_initial` fortgesetzt. Beim Durchlaufen der Zusicherungen oder Nachbedingungen können `initial`-Ausdrücke dann einfach übersprungen werden, da ihr Wert bereits vorliegt.

```
IF P = N.after:
   IF Initial_Expr@<N> I:    Move (I, N, next_initial (P, N.encl_rout))
   ...
 | P = N.before:
   IF Initial_Expr@ N:       N.after
   ...
```

Abbildung 5.12: Kontrollfluß für Initialausdrücke

Methodenrückkehr

Die Bearbeitung des Methodenrumpfes wird stets mit der Ausführung einer `return`-Anweisung abgeschlossen. Diese bewirkt weiterhin die Speicherung des Rückgabewerts an der Definitionsstelle der Routine. Anschließend erfolgt jedoch nicht sofort die Rückkehr zum Aufrufer, da im Terminalzustand zuerst noch die Annotierungen überprüft werden müssen. Die Bearbeitung wird daher mit der Auswertung der Nachbedingungen fortgesetzt (vgl. Abbildung 5.13).

```
IF P = N.after:
   IF Return_Stmt@<N> R:     LET H == R.encl_rout:
                             Move (H, N, H.post.before)
   ...
```

Abbildung 5.13: Kontrollfluß für `return`-Anweisung

Die Funktion `after_cond_list` bewirkt anschließend den Aufruf der `Return`-Aktion (vgl. Abbildung 5.14). Diese überprüft wiederum, ob beim Aufruf der Methode ein Wechsel in ein anderes Objekt erfolgt ist. Ist dies der Fall, so wird dieses Objekt bei der Rückkehr aus der Methode wieder verlassen, weshalb erneut die Objektinvarianten geprüft werden müssen.

```
CASE ct().sort
IS _Return THEN
  LET INV == curr_routine(ci()).encl_class.invariant:
     // Objektinvarianten der aufgerufenen Klasse
  LET TASK == rt(ci()).dyn_task:
     // Dynamischer Task, zum Fortsetzen der Bearbeitung

  IF is[TASK.node.receiver, _Self_Expr@] THEN
     ct() := Invariant()
  ELSE
     ct() := INV.before
  FI
  pre_state(ci()) := false()
```

Abbildung 5.14: Regeln für SALSA-Methodenrückkehr

Der `Invariant`-Task wird aufgerufen, wenn diese Abprüfung beendet bzw. gar nicht erforderlich ist. Da nach Abarbeitung der Regeln für `Return` die dynamische Funktion `pre_state` den Wert `false` enthält, wird die Behandlung des Methodenaufrufs mit der Rückkehr zum Aufrufer abgeschlossen (vgl. Abbildung 5.10).

5.3.3 Beispiel

Das Programmbeispiel der Klasse `NAT` soll im folgenden um die oben beschriebenen Konstrukte zur Annotierung erweitert werden.

Da die Voraussetzungen beim Aufruf der Methode `main` bekannt sind, braucht diese keine Vorbedingungen zu stellen, so daß sich die Annotierung auf die Routinen `create` und `fac` beschränkt.

```
functional create(n: INT): SAME
requires
   not_negative: n >= 0
is
   r: SAME;
   return r.x(n)
end; -- create
```

Abbildung 5.15: Annotierte `create`-Routine

`create` dient der Erzeugung neuer `NAT`-Objekte aus `INT`-Werten. Um die Invariante der `NAT`-Klasse erfüllen zu können, muß `create` dafür sorgen, daß dem

Attribut `x` nur nicht-negative Zahlen zugewiesen werden. Dies kann durch die Formulierung einer Vorbedingung wie in Abbildung 5.15 erfolgen.

Eine Nachbedingung wird hier nicht angegeben, da der korrekte Inhalt des Attributs bereits durch die Objektinvariante gesichert ist. Eine Wiederholung dieser Forderung als Nachbedingung hätte den Effekt, daß diese auch bei Aufrufen innerhalb des Objekts geprüft würde.

Im Gegensatz zu `create` lassen sich für die Fakultätsfunktion `fac` umfangreichere Annotationen angeben (vgl. Abbildung 5.16). Die Objektinvariante sorgt zwar bereits dafür, daß der Wert des `self`-Objekts in der Definitionsmenge der Fakultätsfunktion liegt, eine zusätzliche Einschränkung nach oben hin verhindert jedoch auch das Auftreten eines arithmetischen Überlaufs.

```
functional fac: SAME
requires
   no_overflow: x < 11
ensures
   func1: initial(x) = 0 => result.x = 1,
   func2: initial(x) > 0 =>
          result.x = initial(x) * create(initial(x) - 1).fac.x
is
   ...
```

Abbildung 5.16: Annotierte `fac`-Routine

Um das Verhalten von `fac`, die den Wert der Fakultät iterativ berechnet, näher zu beschreiben, wird die Definition der Fakultät in die Nachbedingung integriert. Diese zeigt die Verwendung von Implikationen und Initialausdrücken.

Die Implikationen werden vom Interpreter in eine Oder-Verknüpfung transformiert. Da für jedes `x` >= 0 genau eine der beiden linken Seiten der Implikationen wahr ist, wird auch genau eine der beiden rechten Seiten ausgewertet. Nur diese Bedingung ist von Interesse, da die andere Implikation `true` liefert und deshalb innerhalb der Und-Verknüpfung einer Bedingungsliste keine Bedeutung hat.

Ist der ursprüngliche Wert von `x` gleich `0`, so wird überprüft, ob das Attribut des Ergebnisses den Wert `1` hat. Der Zugriff auf das Attribut ist hier notwendig, da für die Klasse `NAT` keine Vergleichsoperatoren implementiert wurden und deshalb die der `INT`-Klasse verwendet werden müssen. Für alle anderen Werte von `x` wird der Wert der Fakultät rekursiv berechnet und mit dem Ergebnis von `fac` verglichen.

Diese elegante Formulierung wird durch die nicht strikte Semantik der logischen Ausdrücke ermöglicht. Im Fall `x = 0` wird die rechte Seite der zweiten Implikation nicht ausgewertet und so ein Aufruf der Funktion `fac` für den Wert `-1` vermieden.

Die Einbindung der rekursiven Fakultätsdefinition in die Nachbedingung hat zur Folge, daß die dynamische Abprüfung sehr zeitaufwendig ist. Andererseits garantiert sie bei einem Aufruf von `n.fac`, daß die Funktion für alle $m \leq n$ korrekt arbeitet. Die Verwendung rekursiver Funktionen zur Annotierung ist auch deshalb problematisch, weil deren Terminierung in Zusammenhang mit den Vor- und Nachbedingungen oft nur schwer zu durchschauen ist.

5.4 Zusicherungen

Zusicherungen stellen die einfachste Form von Annotierung dar. Da sie auf Anweisungsebene arbeiten, sind sie in SALSA als Anweisung implementiert.

5.4.1 Syntax und Semantik

In der konkreten Syntax bestehen Zusicherungen aus dem Schlüsselwort `assert`, gefolgt von einer Bedingungsliste. Sie dürfen überall dort stehen, wo auch Sather-Anweisungen erlaubt sind. Die Produktionen der abstrakten Syntax für Anweisungen werden deshalb, wie in Abbildung 5.17 gezeigt, erweitert.

```
Statement     = ...| Assert_Stmt
Assert_Stmt   ( Condition_List )
```

Abbildung 5.17: Produktionen für Zusicherungen

Die Semantik der Zusicherungen ergibt sich aus der Beschreibung der Anweisungen und der Bedingungslisten. Für Zusicherungen existiert weder ein spezieller Eintrag in der Kontrollflußfunktion `psucc` noch ein eigener Task. Der Kontrollfluß durchläuft die Ausführungspunkte, bis die Bedingungsliste erreicht ist. Diese wird wie bei Invarianten oder Vor- und Nachbedingungen behandelt. Anschließend werden die Ausführungspunkte nach der `assert`-Anweisung weiterverfolgt.

5.4.2 Beispiel

Am Beispiel des Rumpfes der Funktion `fac` soll die Verwendung von Zusicherungen gezeigt werden (vgl. Abbildung 5.18). Die lokale Variable `r` dient zur Aufnahme des Rückgabewerts der Methode. Da es sich dabei um den Wert der Fakultät handelt, muß stets `r > 0` gelten. Das Einbringen dieser Bedingung in die Schleife der `fac`-Funktion ermöglicht unter anderem die Erkennung einer fehlerhaften Abbruchbedingung. Läuft die Schleife zu weit, so unterschreitet die Variable `n` den Wert 1, was sich durch die Multiplikation in einem fehlerhaften Wert von `r` niederschlägt und somit zur Verletzung der Zusicherung führt.

```
...
r: INT := 1;
n: INT := x;

if n > 1 then
   loop
      assert correct_r: r > 0;
      r := r * n;
      n := n - 1;
      while!(n > 1)
   end -- loop
end; -- if
...
```

Abbildung 5.18: Verwendung von Zusicherungen

Wie in diesem Beispiel dienen Zusicherungen häufig der Einfügung von Redundanz in den Programmcode. Die Abprüfung, ob ein Zeiger tatsächlich ein Objekt im Speicher referenziert, gehört zu den Sicherheitsmaßnahmen, die viele Programmierer stets in ihre Programme integrieren. Im Gegensatz zu Zusicherungen führen solche Abprüfungen mittels `if`-Anweisungen jedoch zu einer Verlangsamung der Programmausführung, während die dynamische Abprüfung der `assert`-Anweisungen nach einer angemessenen Testphase abgeschaltet werden kann.

Kapitel 6

Der praktische Einsatz von SALSA

Nachdem im vorangegangenen Kapitel Syntax und Semantik von SALSA dargestellt wurden, soll im folgenden Abschnitt untersucht werden, inwieweit sich die Annotationssprache zur Spezifikation und Dokumentation im praktischen Einsatz eignet.

6.1 Das Programmbeispiel

Um herauszufinden, ob die Mächtigkeit von SALSA tatsächlich ausreicht, die Eigenschaften von Programmen zu spezifizieren, wurde ein nicht triviales Programmbeispiel entworfen und annotiert. Das Programm dient der Berechnung von Bruchtermen. Es erwartet als einzigen Parameter einen Term in Postfixnotation, dessen Wert berechnet werden soll. Als Werte dienen ganze Zahlen und Brüche, die als `Zähler:Nenner` geschrieben werden. Erlaubte Operationen sind die vier Grundrechenarten `+`, `-`, `*` und `/`. Die einzelnen Bestandteile des Terms werden durch Kommata voneinander getrennt. Leerzeichen in der Eingabe sind zulässig und werden ignoriert. Der Ausdruck `1:4, 1:4, +` ergibt beispielsweise `1:2` als Ergebnis.

Bei der Berechnung findet ein Keller Verwendung, auf dem Operanden und Zwischenergebnisse abgelegt werden. Jede Rechenoperation holt die beiden obersten Werte vom Keller und legt das Ergebnis wieder dort ab.

Der Quellcode des Programms besteht aus zwei abstrakten und vier konkreten Klassen. Ihre Bedeutung gibt Tabelle 6.1 wieder. Die Vererbungs- und Inkludierungsbeziehungen zwischen den Klassen sind in Abbildung 6.1 dargestellt. Durchgezogene Linien deuten dabei auf Vererbung, gestrichelte Linien auf Inkludierung hin. Der gesamte Quelltext des Beispiels ist in Anhang A zu finden. Aus Platzgründen wird bei der Besprechung des Programms auf eine Abbildung der relevanten Teile verzichtet und lediglich unter Angabe der Zeilennummern auf

den Anhang verwiesen.

Klasse	Funktion
`$IS_EQ`	Menge aller Klassen, auf denen die Gleichheit definiert ist
`$STACK`	Abstrakter Datentyp für einen Keller
`FRACTION`	Datentyp für Brüche mit vier Grundrechenarten
`STACK`	Datentyp für Keller
`CALC`	Berechnung der Bruchterme
`MAIN`	Analyse des Postfixausdrucks

Tabelle 6.1: Die Klassen des Programmbeispiels

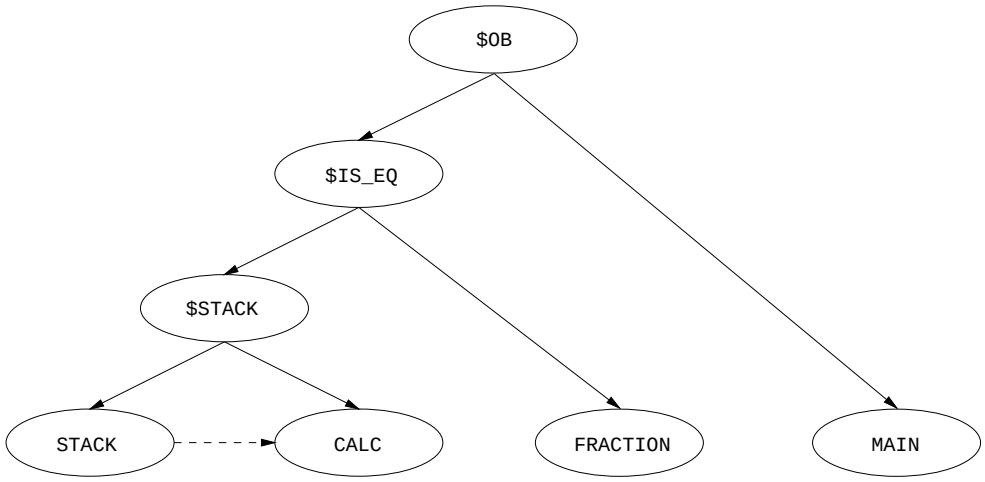

Abbildung 6.1: Vererbungs- und Inkludierungsbeziehungen

6.2 Die Spezifikation

In den folgenden Abschnitten werden die Klassen des Progammbeispiels mit ihren Annotationen besprochen. Dabei wird zwischen den abstrakten und den konkreten Klassen unterschieden, da in beiden Fällen unterschiedliche Ziele verfolgt werden.

6.2.1 Die abstrakten Typen

Während bei der Annotierung konkreter Klassen meist das Verhalten des Programmstücks beschrieben werden soll, geht es bei abstrakten Klassen um die

Spezifikation abstrakter Datentypen. Für diese Aufgabe wird oft die Methode der *algebraischen Spezifikation* herangezogen.

Die Einbindung algebraischer Spezifikationen

Bei der algebraischen Spezifikation werden die Rechenstrukturen als Algebren — bestehend aus einer Trägermenge und Operationen — aufgefaßt. Die Eigenschaften dieser Operationen werden durch algebraische Gleichungen, sog. Gesetze, ausgedrückt. Eine ausführliche Behandlung dieser Thematik erfolgt in [BW81].

Algebraische Spezifikationen ermöglichen die exakte und vollständige Erfassung der Eigenschaften einer abstrakten Datenstruktur. Bei der Erstellung abstrakter Klassen wird deshalb angestrebt, diese Spezifikationen möglichst vollständig in den Programmcode zu integrieren. Während bei der Verwendung nicht annotierter Klassen lediglich Trägermenge und Operationen nachgebildet werden können, erlauben Annotationssprachen zusätzlich die Formulierung der Gesetze.

Die Übertragung von algebraischen Spezifikationen auf abstrakte Klassen kann weitgehend schematisch erfolgen. Im folgenden soll dies am Beispiel der Spezifikation des Datentyps *stack*χ[1] aus Abbildung 6.2 gezeigt werden.

mode stackχ,
funct stackχ *empty*,
funct (stackχ**) bool** *isempty*,
funct $(\{$**stack**χ $x : \neg isempty(x)\})$ χ *top*,
funct $(\{$**stack**χ $x : \neg isempty(x)\})$ **stack**χ *rest*,
funct (stackχ**,** χ**) stack**χ *append*,
funct (stackχ**) nat** *length*

law R: $rest(append(a, x)) = a$,
law T: $top(append(a, x)) = x$,
law A: $\neg isempty(a) \Rightarrow append(rest(a), top(a)) = a$,
law L1: $length(empty) = 0$,
law L2: $length(append(a, x)) = length(a) + 1$,
law E1: $isempty(empty)$,
law E2: $\neg isempty(append(a, x))$

Abbildung 6.2: Algebraische Spezifikation des Datentyps *stack*χ

Die Trägermenge besteht aus den Sorten *bool*, *nat*, χ und *stack*χ. Diese müssen bei der Einbindung in SALSA durch entsprechende Typen ersetzt werden. Die Operationen werden in Sather als Routinen dargestellt. Parameter- und Ergebnistypen können übernommen werden. Lediglich der Parameter vom Typ

[1] siehe [BW81, S. 205]

stackχ, der bei allen Funktionen mit Ausnahme von *empty* auftritt, kann bei der Implementierung der abstrakten Klasse entfallen. An seine Stelle tritt das `self`-Objekt, das als erster Parameter interpretiert werden kann.

Partielle Funktionen werden in algebraischen Spezifikationen hervorgehoben, indem die Einschränkung des Definitionsbereichs explizit erwähnt wird (vgl. Funktionen `top` und `rest` in Abbildung 6.2). Diese Einschränkungen können unmittelbar als Vorbedingungen der entsprechenden Sather-Routinen formuliert werden.

Schwieriger gestaltet sich die Umsetzung der einzelnen Gesetze. Auf beiden Seiten dieser Gleichungen können Anwendungen der Operationen des spezifizierten Datentyps auftreten. Als Parameter erscheinen freie Variablen oder wiederum Funktionsterme. Die meisten der Gesetze können schematisch in Annotationen umgeformt werden. Da sie Aussagen über das Ergebnis von Funktionen treffen, werden sie in der Regel als Nachbedingungen formuliert.

Funktionsapplikationen in den Gesetzen entsprechen Methodenaufrufen in der Annotationssprache. Die Überprüfung der Gleichheit stellt ebenfalls einen Aufruf einer Routine (`is_eq`) dar. Daher kann jedes Gesetz der Spezifikation in eine Folge von Methodenaufrufen transformiert werden. Für die Gleichung

$$linke_Seite = rechte_Seite$$

kann diese Folge dann

```
linke_Seite.is_eq(rechte_Seite)
```

oder

```
rechte_Seite.is_eq(linke_Seite)
```

lauten, wobei sowohl `linke_Seite` als auch `rechte_Seite` wiederum aus mehreren Aufrufen besetehen können. Um ein Gesetz schematisch in eine Annotation transformieren zu können, ist die Umformung so zu wählen, daß der insgesamt erste Aufruf an eine Routine der der zu spezifizierenden Klasse gerichtet ist und nur freie Variablen als aktuelle Parameter enthält.

Das Gesetz wird dann als Nachbedingung der zuerst aufgerufenen Methode formuliert, wobei deren angewandtes Auftreten in der Folge durch den `result`-Ausdruck ersetzt wird. Die aktuellen Parameter des Aufrufs werden bei dessen Ersetzung durch `result` an die formalen Parameter der Routine gebunden. Alle anderen aktuellen Parameter werden mit einem `initial`-Ausdruck versehen, um im Terminalzustand die ursprünglichen Werte ansprechen zu können. Alle verbleibenden Methodenaufrufe müssen an funktionale Methoden gerichtet sein. Dieses Schema ist anwendbar, wenn

1. eine Transformation des Gesetzes in eine Folge von Methodenaufrufen exisitiert, die außer dem ersten Folgenglied nur Aufrufe funktionaler Methoden enthält und

2. der erste Aufruf dieser Folge an eine Methode der spezifizierten Klasse gerichtet ist und

3. der erste Aufruf dieser Folge nur freie Variablen als Parameter enthält.

Andernfalls läßt sich das Gesetz nicht unter Anwendung dieses Schemas — in manchen Fällen sogar überhaupt nicht — in eine Annotation transformieren.

Das dargestellte Schema zur Transformation von Gesetzen der algebraischen Spezifikation in Nachbedingungen von Routinen funktioniert in vielen Fällen. Dennoch treten Schwierigkeiten auf, die meist nur durch eine Abschwächung der Gesetze zu beheben sind.

- Gesetze mit zwei oder mehreren verschiedenen Anwendungen nicht funktionaler Methoden können nicht transformiert werden, da nur einer dieser Ausdrücke mit einem `result`-Ausdruck identifiziert werden kann.

- Das Auftreten von freien Variablen, die nicht an formale Parameter gebunden werden können, kann nicht adäquat behandelt werden. Möchte man beispielsweise das Assoziativgesetz einer Gruppe — $(a \circ b) \circ c = a \circ (b \circ c)$ — als Nachbedingung einer Methode `conc` formulieren, so können lediglich zwei der Parameter — z. B. a und b — gebunden werden. Das Gesetz muß jedoch für jedes c gelten und erfordert daher die Unterstützung von Quantoren seitens der Annotationssprache. Umgangen werden kann dieses Problem in SALSA nur dadurch, daß auf die Quantifizierung verzichtet wird und statt dessen ein bestimmtes, nicht näher spezifiziertes c zur Formulierung des Gesetzes herangezogen wird. Es muß also eine Abschwächung des Gesetzes in Kauf genommen werden.

- Die Gleichheit zwischen Elementen der Trägermenge ist in einer Algebra exakt definiert. In Sather stellt die Überprüfung der Gleichheit zweier Objekte jedoch lediglich einen Aufruf der Routine `is_eq` dar. Die Gleichheit im Sinn der Spezifikation kann nicht immer festgestellt werden. Soll z. B. die Gleichheit zweier Geflechte, die aus Referenzobjekten aufgebaut sind, über verschiedene Zustände der Ausführung hinweg überprüft werden, so kann lediglich die Gleichheit der Referenzen, nicht jedoch die der ganzen Geflechte festgestellt werden. Abhilfe schaffen kann hier die Bildung eines Value-Objekts, das die im Geflecht enthaltenen Werte aufnimmt. Da Value-Klassen keine rekursiven Datenstrukturen modellieren können, ist dieser Weg jedoch nur in wenigen Ausnahmefällen praktizierbar.

Die Klasse $IS_EQ

Die Klasse `$IS_EQ` wurde lediglich eingeführt, um eine Menge von Klassen, auf denen die Gleichheitsfunktion `is_eq` definiert ist, bezeichnen zu können. In der

Implementierung des Kellers stellt sie die Sorte χ dar. Der Keller kann somit Objekte aller von `$IS_EQ` abgeleiteten Klassen aufnehmen.

Stünde im Interpreter der volle Sprachschatz Sathers zur Verfügung, so würde man den Stack sicherlich als parametrisierte Klasse gestalten. Die hier gezeigte Methode mit der Einführung einer abstrakten Klasse soll zum einen demonstrieren, wie auch ohne parametrisierte Typen sinnvolle Implementierungen erstellt werden können, und zum anderen die Beschreibung von Typabfragen in SALSA vorführen.

Eine noch allgemeinere Version eines Kellers kann man erreichen, indem man alle Typen, also $OB, für χ zuläßt. Die Formulierung der Gesetze der algebraischen Spezifikation erfordert jedoch, daß die Gleichheit zwischen Elementen des Stacks definiert ist, weshalb `$IS_EQ` die Definition der entsprechenden Methode vorschreibt.

Da `$IS_EQ` keinen abstrakten Datentyp im eigentlichen Sinn, sondern vielmehr eine Menge von Typen darstellt, wurde auf eine Annotierung dieser Klasse verzichtet. Da sie sehr allgemein verwendet wird, ließen sich ohnehin nur minimale Forderungen stellen.

Die Klasse $STACK

Die Klasse `$STACK` stellt die Transformation der algebraischen Spezifikation aus Abbildung 6.2 in ein Sather/SALSA-Programm dar.

Bei der Umsetzung nach Sather werden *bool* auf `BOOL` und *nat* auf `INT` abgebildet. Als Datentyp χ dient der abstrakte Typ `$IS_EQ`. Den neu zu erstellenden Typ *stack*χ stellt schließlich die Klasse `$STACK` dar.

Da *nat* und `INT` unterschiedliche Mengen von Zahlen bezeichnen, wurde durch Einführung der Objektinvarianten (vgl. Anhang A, Z. 69) der Wertebereich der Funktion `length` auf die nicht-negativen Zahlen beschränkt, so daß sich durch den Übergang von *nat* zu `INT` keinerlei Veränderungen der Spezifikation ergeben.

Die Operationen von *stack*χ wurden unmittelbar in Sather übernommen, lediglich *empty* wurde in `create` umbenannt, um die syntaktischen Abkürzungen Sathers verwenden zu können. Die Ausführung der Routinen `create` und `append` wird mit der Erzeugung bzw. Veränderung von Objekten einhergehen, weshalb diese Methoden nicht als funktional gekennzeichnet wurden. Alle anderen Routinen tragen hingegen dieses Attribut.

Wie bereits erwähnt, können Einschränkungen des Definitionsbereichs von Methoden durch Vorbedingungen ausgedrückt werden. Die Operationen `top` und `rest` sind nur auf nicht leeren Stacks erlaubt, weshalb die zugehörigen Methoden folgende Annotation erhalten (vgl. Anhang A, Z. 80, 82):

```
requires not_empty : isempty.not
```

Die Umsetzung der Gesetze der Spezifikation erfolgt nach dem im vorangegangenen Abschnitt beschriebenen Schema und soll hier lediglich ausschnittsweise vorgeführt werden.

Das Gesetz `law R` enthält einen Aufruf der nicht funktionalen Methode `append`, wobei keine Einschränkungen seitens der Parameter vorhanden sind. Das Gesetz kann somit als Nachbedingung dieser Routine formuliert werden. Bei der Identifizierung des `append`-Aufrufs mit `result` wird der Parameter a an das `self`-Objekt und x an den Parameter `x` der Routine gebunden. Die Annotation lautet somit (vgl. Anhang A, Z. 75):

$$\texttt{law_R} : \texttt{result.rest} = \texttt{initial}(\texttt{self})$$

Die Anwendung des `initial`-Ausdrucks ist nicht unbedingt erforderlich, da sich der Wert des `self`-Ausdrucks in einer Routine ohnehin nicht verändern kann. Dennoch verdeutlicht der Initialausdruck, daß in der algebraischen Spezifikation a den Stack vor Ausführung der Operation bezeichnet, sich also auf den Initialzustand bezieht.

Analog gestaltet sich die Einbindung von Gesetz `law L1`, das keine Aufrufe nicht funktionaler Methoden enthält. Es wird als Nachbedingung der Routine `create` formuliert (vgl. Anhang A, Z. 71):

$$\texttt{law_L1} : \texttt{result.length} = 0$$

Ebenso wie `law R` und `law L1` lassen sich die weiteren Gesetze in Annotationen überführen. Lediglich Gesetz `law A` läßt sich nicht in gewohnter Weise transformieren, da beim Übergang von *append(rest(a), top(a))* zu `result` die Information über die Parameter verlorengeht. Mit den Mitteln von SALSA kann dieses Gesetz nicht als Annotation dargestellt werden.

6.2.2 Die konkreten Typen

Im folgenden Abschnitt wird auf die Implementierung und Annotierung der konkreten Typen des Programmbeispiels eingegangen.

Die Klasse STACK

Die Klasse `STACK` ist vom abstrakten Typ `$STACK` abgeleitet. Sie stellt die Implementierung des dort spezifizierten Kellers dar. Da es sich dabei um einen rekursiven Datentyp handelt, wurde `STACK` als Referenzklasse realisiert. Diese stellt neben den von `$STACK` vorgeschriebenen Operationen auch die von `$IS_EQ` geforderten Routinen zur Verfügung. Die Operation `length` wurde nicht als Methode, sondern als Attribut implementiert. Die Leseroutine für dieses Attribut entspricht jedoch der von `$STACK` verlangten Signatur, so daß die Vererbungsregel Sathers erfüllt ist.

Die Annotationen des abstrakten Datentyps `$STACK` sind durch eine zusätzliche Objektinvariante ergänzt. Diese beschreibt, in welcher Beziehung die einzelnen Objekte des Geflechts zueinander stehen. Sie garantiert, daß die Liste, die den Stack repräsentiert, azyklisch ist. Da SALSA keine Konstrukte zur Verfügung stellt, die erkennen lassen, ob zwei Geflechte gemeinsame Speicherbereiche nutzen, ist hierfür die Hilfsfunktion `is_referenced` erforderlich, die rekursiv nach Zyklen in der Liste sucht (vgl. Anhang A, Z. 91–95). Dabei wird lediglich überprüft, ob das vorderste Element der Liste von einem der hinteren Elemente referenziert wird. Somit werden nur Zyklen aufgedeckt, die den Kopf der Liste enthalten. Da der Keller jedoch nur um einzelne Elemente erweitert werden kann, ist garantiert, daß auch im hinteren Teil keine Zyklen auftreten.

Da zusätzliche Einschränkungen im Bereich der Vorbedingungen, also Verschärfungen, nicht zulässig sind, wurden mit Hilfe von `assert`-Anweisungen einige weitere Bedingungen in Form von Zusicherungen eingefügt (vgl. Anhang A, Z. 97, 111, 119). Diese verhindern, daß bei direkten Methodenaufrufen an die Klasse der Rumpf der Prozedur ausgeführt wird, ohne daß ein entsprechendes `self`-Objekt exisitert.

Diese Bedingungen können weder als Objektinvarianten formuliert werden, da sonst ein direkter Aufruf der `create`-Routine nicht mehr möglich ist, noch kann die Annotierung in der Superklasse erfolgen, wo noch keine Details der Implementierung bekannt sind.

Die als Nachbedingungen formulierten Gesetze der algebraischen Spezifikation erhalten ihre Semantik erst durch die Definition der Gleichheitsfunktion `is_eq`. Diese erwartet zwei Referenzen auf Objekte vom Typ `STACK` und prüft rekursiv die Gleichheit aller im Keller enthaltenen Elemente (vgl. Anhang A, Z. 96–103). Durch diese Konstruktion ist ein Vergleich von Stacks aus verschiedenen Zuständen der Programmausführung nicht möglich, da vom vergangenen Zustand lediglich die Referenz erhalten ist, nicht jedoch der Rest des Geflechts. Es tritt also die in Abschnitt 6.2.1 angesprochene Abschwächung der Gesetze ein.

Die Klasse FRACTION

Zur Darstellung von Brüchen dient die Klasse `FRACTION`. Sie zeigt einige typische Anwendungen und Probleme von SALSA.

Der Bruch wird in drei Attributen gespeichert, wobei Vorzeichen, Zähler und Nenner getrennt werden. Wegen der expliziten Darstellung des Vorzeichens müssen Zähler und Nenner stets nicht-negativ sein, was sich in entsprechenden Objektinvarianten niederschlägt (vgl. Anhang A, Z. 6f). Die Tatsache, daß der Nenner eines Bruches stets ungleich 0 ist, läßt sich nicht als Invariante formulieren, da bei Aufrufen der `create`-Methode kurzzeitig ein mit `void` initialisiertes Objekt erzeugt wird, bei dem sowohl Zähler als auch Nenner 0 sind. Hierbei wird wieder deutlich, daß das Fehlen expliziter Konstruktorroutinen in Sather eine große Einschränkung der Annotierbarkeit darstellt.

Durch eine weitere Invariante (vgl. Anhang A, Z. 8) wird ausgedrückt, daß der Bruch stets in vollständig gekürzter Form gespeichert wird. Dies wird durch Anwendung der Methode `ggT` zur Berechnung des größten gemeinsamen Teilers (Algorithmus vgl. [Knu69, S. 297]) überprüft. Die Annotierung dieser Methode (vgl. Anhang A, Z. 19–22) gibt das Verhalten nur unzureichend wieder. Die Nachbedingungen garantieren lediglich, daß der Rückgabewert Teiler beider Argumente ist. Auch die konstante Funktion `1` würde diese Bedingungen erfüllen.

Zur Definition des größten gemeinsamen Teilers ist eigentlich der Einsatz von Quantoren erforderlich. Es muß sichergestellt werden, daß kein größerer gemeinsamer Teiler als der gefundene exisitiert. Da die Methode `ggT` auf nicht-negative Argumente beschränkt ist, muß der größte gemeinsame Teiler zwischen dem gefundenen Wert und dem Minimum der beiden Argumente liegen. Es kommen also nur endlich viele Werte in Betracht, so daß in einer Schleife überprüft werden kann, ob einer von ihnen ebenfalls gemeinsamer Teiler der Argumente ist.

Durch eine Hilfsfunktion kann hier auch ohne Einsatz von Quantoren eine vollständige Spezifikation der Routine erfolgen. Die Durchführung dieser Idee ist trivial und wird deshalb hier nicht vorgeführt.

Die weiteren Annotationen der Klasse `FRACTION` sind leicht verständlich. Sie dienen der Formulierung von allgemein bekannten Bedingungen, wie dem Verbot der Division durch 0 (vgl. Anhang A, Z. 63) oder der Forderung, daß der Nenner stets ungleich 0 sein muß, die aus oben dem beschriebenen Grund nicht als Invariante, sondern als Nachbedingung angegeben wird (vgl. Anhang A, Z. 38, 51).

Die Semantik der Gleichheit ist bei Objekten der Klassen `FRACTION` einfacher, da es sich hierbei um Value-Objekte handelt. Die Gleichheitsfunktion kann sich auf die Definition der Objektgleichheit (vgl. Anhang A, Z. 44) abstützen. Auch der Vergleich von Objekten aus verschiedenen Zuständen ist problemlos möglich, da hier wirklich alle relevanten Teile der Datenstruktur — in diesem Fall lediglich die Attribute — aus beiden Zuständen vorliegen.

Die Annotierung von `FRACTION` zeigt, daß sich mit SALSA ein sehr großer Teil der Verträge bei der Erstellung konkreter Klassen formulieren läßt. Sie ist jedoch keineswegs vollständig, da beispielsweise auf die Einbindung der Körper-Gesetze verzichtet wurde. Das dabei auftretende Problem mit dem Assoziativgesetz wurde bereits in Abschnitt 6.2.1 behandelt.

Die Klasse CALC

Die eigentliche Berechnung der Bruchterme vollzieht die Klasse `CALC`. Sie ist ein Stack, der um die Routine `compute` erweitert wurde. Durch die Anwendung einer Inkludierungsbestimmung (vgl. Anhang A, Z. 136) stehen alle Methoden und Attribute der `STACK`-Klasse vollständig, d. h. einschließlich der Annotationen, für `CALC` zur Verfügung.

Aufgrund der Zulassung aller Objekte vom Typ `$IS_EQ` im Keller muß durch geeignete `typecase`-Anweisungen (vgl. Abschnitt 4.1.3) sichergestellt werden,

daß die Operationen wirklich auf Objekten vom Typ `FRACTION` ausgeführt werden. Bei Verwendung einer parametrisierten `STACK`-Klasse könnten diese Abfragen entfallen.

Die Annotierung der Routine `compute` stellt im wesentlichen den Rahmen sicher, in dem die Berechnungen ausgeführt werden können. Interessant ist die Möglichkeit, Aussagen über die Typen von Objekten zu treffen. In SALSA geschieht dies durch die Methoden der Bibliotheksklasse `SYS`:

$$\texttt{SYS} :: \texttt{str_for_tp}(\texttt{SYS} :: \texttt{tp}(\texttt{top})) = \text{"}\texttt{FRACTION}\text{"}$$

Im Beispiel dienen sie der Überprüfung, ob die Operanden der arithmetischen Operation tatsächlich Brüche, d. h. vom Typ `FRACTION`, sind. Gleiches wird für das Ergebnis der Berechnung zugesichert (vgl. Anhang A, Z. 142, 143, 146).

Die Klasse MAIN

Die Analyse der Eingabe und die Steuerung der Berechnung wird von den Methoden der Klasse `MAIN` vorgenommen. Um das Beispiel nicht unnötig aufzublähen, wurde auf eine Annotierung dieser Klasse verzichtet. Es wird lediglich demonstriert, wie die Einhaltung der Vertragsverpflichtungen durch geeignete Abfragen im Programm sichergestellt werden kann.

6.2.3 Grenzen der Ausdrucksmöglichkeit

Die Behandlung des obigen Beispiels zeigt, wo die Grenzen der Spezifikationsmöglichkeiten von SALSA liegen. Zusammenfassend läßt sich feststellen, daß sich praxisnahe Forderungen aus den Verträgen der Softwareentwicklung weitgehend mit den Mitteln von SALSA ausdrücken lassen.

Dazu zählt beispielsweise die Formulierung partieller Funktionen: Die Einschränkung des Definitionsbereichs durch Vorbedingungen ist in der Regel problemlos möglich, wie die Annotierung der Routine `compute` (vgl. Anhang A, Z. 138–146) zeigt. Ebenso lassen sich die Seiteneffekte und der Rückgabewert einer Methode beschreiben (vgl. `append`-Routine). Die Objektinvarianten der `STACK`-Klasse (vgl. Anhang A, Z. 86, 87) machen deutlich, daß sich unter Zuhilfenahme von Sather-Routinen Integritätsbedingungen für Objekte und ganze Geflechte einbinden lassen.

Problematisch sind eher diejenigen Bedingungen, die mehr von theoretischem Interesse als von praktischer Bedeutung sind. Beispiel hierfür sind die Körpergesetze für die Klasse `FRACTION`, die eigentlich unmittelbar aus der Implementierung der Grundoperationen folgen. Bei der expliziten Aufführung als Annotation stößt man hingegen schnell an die Grenzen von SALSA.

Besondere Schwierigkeiten bereitet der Vergleich von Geflechten über verschiedene Zustände der Ausführung hinweg. Das Verbot, in Annotationen neue

Geflechte zu erzeugen, verhindert das Zwischenspeichern rekursiver Datenstrukturen und somit deren Vergeich.

Geringere Auswirkungen hat hingegen das Fehlen von SALSA-Ausdrücken zur Beschreibung der Beziehungen zwischen verschiedenen Objekten. Die Annotierung der `STACK`-Klasse mit der Funktion `is_referenced` (vgl. Abschnitt 6.2.2) demonstriert, wie man durch Formulierung zusätzlicher Funktionen auf solche Konstrukte verzichten kann.

Die beiden gewichtigsten Probleme von SALSA — fehlende Allquantoren und schlechte Vergleichbarkeit von Geflechten — spielen nur im Zusammenhang mit den dynamischen Abprüfungen eine Rolle. Betrachtet man hingegen die Spezifikation als formale Dokumentation der Programmeigenschaften, so treten diese Schwierigkeiten in den Hintergrund. In einer Dokumentation kann davon ausgegangen werden, daß jede freie Variable allquantifiziert ist. Außerdem kann man festlegen, daß hier die Gleichheit von Geflechten die ganze Datenstruktur und nicht nur die Referenzen betreffen soll.

6.3 Die automatische formale Dokumentation

Eines der Ziele bei der Entwicklung von SALSA war, die automatische Erstellung einer formalen Dokumentation zu ermöglichen. Unter *Dokumentation* versteht man hierbei eine lesbare Spezifikation der Programmeigenschaften. Da diese in Form einer Annotierung gegeben ist, kann die Dokumentation unmittelbar aus dem Programmtext extrahiert werden.

Diese Aufgabe erfüllt das Programm *AutoDoc*. Es ähnelt in weiten Teilen dem `short`-Kommando der Eiffel-Umgebung (vgl. [Mey92b, S. 103ff]). AutoDoc baut auf den Quellen des SALSA-Interpreters auf. Es nutzt Scanner und Parser des Interpreters, um einen abstrakten Syntaxbaum zu erzeugen. Durch Einsatz verschiedener Ausgabefunktionen wird dieser anschließend in ein Programm mit konkreter Syntax transformiert. Die Quellen von AutoDoc finden sich in Anhang C.

Durch die Nutzung von Scanner und Parser des Interpreters werden alle Transformationen des Programms, die der Interpreter durchführt, auch von AutoDoc vollzogen. Hierzu zählen vor allem die Auslösung von Inkludierungsbestimmungen, die Umsetzung von konkreter Notation — etwa arithmetische oder Vergleichsoperatoren — in Methodenaufrufe oder die Einbindung der vordefinierten Klassen.

Als Ausgabe erzeugt AutoDoc eine Beschreibung der vom Benutzer definierten Klassen. Diese enthalten die Deklarationen der Attribute und Signaturen sowie Objektinvarianten und Vor- bzw. Nachbedingungen der Routinen. Auf Wunsch kann die Ausgabe auf einzelne Klassen und gegebenenfalls deren Superklassen eingeschränkt werden. Dadurch entsteht eine vollständige Dokumentation der Klasse, die sowohl inkludierten Programmtext als auch geerbte Eigenschaften

berücksichtigt.

Abbildung 6.3 gibt die mit AutoDoc erzeugte Dokumentation der Klasse `STACK` wieder. Zum besseren Verständnis des Interpreters hält AutoDoc noch zwei weitere Optionen bereit: Die Dokumentation kann um die vordefinierten Klassen erweitert werden. Außerdem ist es möglich, auch die Rümpfe der Methoden ausgeben zu lassen, um die Transformationen, die der Parser durchführt, nachvollziehen zu können. In diesem Modus arbeitet AutoDoc wie ein *pretty-printer*.

Mit SALSA ist eine Sprache entstanden, die den Programmierer in die Lage versetzt, einen großen Teil der Eigenschaften des entwickelten Programms zu spezifizieren und automatisch zu dokumentieren. Inwieweit die eingangs gesteckten Ziele, die mit der Entwicklung der Annotationssprache verknüpft sind, durch den Entwurf von SALSA erreicht wurden, wird im nun folgenden abschließenden Kapitel der Arbeit erörtert.

```
class STACK < $STACK, $IS_EQ
invariant
  len: self.length.is_geq(0),
  wellformed: (self.length.is_gt(1).not or
       self.is_referenced(self.r).not) is
  readonly attr length: INT;
  private attr elem: $IS_EQ;
  private attr r: SAME;
  private functional is_referenced(back: SAME): BOOL;
  functional is_eq(a: $OB): BOOL;
  functional is_neq(a: $OB): BOOL;
  create: SAME
  ensures
    law_L1:  result.length.is_eq(0),
    law_E1:  result.isempty;
  functional isempty: BOOL;
  append(a: $IS_EQ): SAME
  ensures
    law_R:  result.rest.is_eq(initial(self)),
    law_T:  result.top.is_eq(initial(a)),
    law_L2: result.length.is_eq(initial(self.length).plus(1)),
    law_E2: result.isempty.not;
  functional rest: SAME
  requires
    not_empty: self.isempty.not;
  functional top: $IS_EQ
  requires
    not_empty: self.isempty.not;
end;
```

Abbildung 6.3: AutoDoc-Dokumentation der STACK-Klasse

Kapitel 7

Beurteilung der Ergebnisse

Zur Bewertung von SALSA werden die in Abschnitt 1.2 formulierten Ziele, die beim Einsatz einer Annotationssprache angestrebt werden, herangezogen. Diese sind, wie die folgenden Abschnitte zeigen, in großen Teilen erreicht worden. Dennoch hat der praktische Einsatz von SALSA (vgl. Kapitel 6) offenbart, daß das hier verfolgte Konzept in manchen Bereichen noch verbesserungswürdig ist. Hierfür sollen am Ende des Kapitels Anregungen gegeben werden.

7.1 Das Erreichen der Ziele

Die drei Ziele — Unterstützung bei Spezifikation, Dokumentation und Fehlersuche —, die sich aus der Forderung nach korrekter und robuster Software ergeben, waren Richtlinie der gesamten Entwicklung und Implementierung von SALSA. Inwieweit die einzelnen Forderungen durch den Entwurf erfüllt werden, wird im folgenden untersucht.

7.1.1 Die formale Spezifikation von Programmeigenschaften

Wie im vorangegangenen Kapitel gezeigt, eignet sich SALSA gut zur Spezifikation von Programmeigenschaften. Abstrakte Datentypen können durch Einbindung algebraischer Spezifikationen weitgehend vollständig beschrieben werden. Die Stärken von SALSA liegen vor allem im Bereich der Spezifikation von konkreten Implementierungen, deren korrekte Nutzung durch die Formulierung von Vertragsbedingungen sichergestellt werden kann. Ein großer Teil der in der praktischen Softwareentwicklung auftretenden Regeln für das Zusammenspiel verschiedener Programmkomponenten kann mit den Mitteln von SALSA erfaßt und formalisiert werden.

Durch die Unterstützung von Vererbung und *code-sharing* ermöglicht SALSA im Bereich der objektorientierten Softwareentwicklung sinnvolle Spezifikationen,

da sich beide Konzepte fortan nicht mehr ausschließlich auf Signaturen und Befehlsfolgen, sondern auch auf die Beschreibung derer Eigenschaften erstreckt. Somit wird die Konsistenz des objektorientierten Ansatzes durch die Anreicherung der Programme nicht verletzt.

Gegenüber anderen Methoden zur Annotierung von objektorientierten Sprachen (vgl. Abschnitt 1.4) ist SALSA vor allem durch die formale Beschreibung seiner Semantik im Vorteil. Diese ermöglicht ein genaues Verstehen der Spezifikationen und legt den Grundstein für spätere Arbeiten, in denen der Aspekt der Verifikation stärker betont werden soll.

Die beiden Hauptschwierigkeiten, die eine vollständige Spezifikation des annotierten Programmtextes oft unmöglich machen, wurden bei der Besprechung des Beispiels in Kapitel 6 aufgezeigt. Ansätze, wie die Mankos fehlender Quantoren und der mangelhaften Vergleichbarkeit von Geflechten verschiedener Zustände beseitigt werden können, werden am Ende dieses Kapitels erläutert.

7.1.2 Die Automatisierung einer formalen Dokumentation

AutoDoc zeigt, wie mit einfachsten Mitteln aus annotierten Programmen formale Dokumentationen erzeugt werden können. Diese Dokumentationen bestehen lediglich aus den Schnittstellen der Klassen. Für den praktischen Einsatz wären einige Erweiterungen wünschenswert, wie z. B. die Wiedergabe gewisser Kommentare zur informellen Beschreibung der Klassen.

Durch die Verwendung der Spezifikation als Dokumentation wird erreicht, daß dem Anwender von Bibliotheksklassen das genaue Verhalten der Routinen beschrieben werden kann, ohne Details der Implementierung preisgeben zu müssen. Dadurch wird die korrekte Nutzung von Bibliotheken ermöglicht, ohne vom Grundsatz der *Kapselung* abzuweichen.

7.1.3 Die Unterstützung bei der Fehlersuche

Die Unterstützung des Programmierers in der Testphase durch SALSA ist ebenso einfach wie effektiv. Da sämtliche Bedingungen der Annotationen zur Laufzeit ausgewertet und überprüft werden können, werden Vertragsverletzungen frühzeitig erkannt. Durch Fehlermeldungen der Form

`Evaluation interrupted near ROUTINE` *CLASS::rout*
`ROUTINE` *CLASS::rout* `: assertion` *ident* `violated`

wird er auf die genaue Lokalisation der verletzten Bedingung hingewiesen. Die Ursache des Fehlers kann so meist sehr schnell gefunden werden.

Die Entwicklung der Programmbeispiele für die Arbeit hat gezeigt, daß dieser einfache Mechanismus geeignet ist, auch normalerweise schwer zu entdeckende

Fehler in kurzer Zeit aufzuspüren. Im Gegenzug darf jedoch nicht unerwähnt bleiben, daß die Annotierung gleichzeitig auch eine neue Fehlerquelle darstellt. Besonders beim Einsatz rekursiver Funktionen innerhalb von Vor- und Nachbedingungen treten schwer zu durchschauende Abläufe auf. Beispiel hierfür ist die annotierte Fakultätsfunktion aus Abbildung 5.16. Die Auswertung der Nachbedingungen bewirkt einen rekursiven Aufruf der Methode, für die dann wiederum die Nachbedingungen geprüft werden. Ein Aufruf der iterativen Methode für das Argument n zieht somit einen Folge von n rekursiven Methodenaufrufen nach sich.

Trotz dieser Problematik überwiegen die Vorteile, die die Annotierung für die Fehlersuche bringen, bei weitem. Dabei gilt: Je vollständiger die Spezifikation der Programmeigenschaften ist, desto wahrscheinlicher ist es, daß vorhandene Fehler während der Testphase aufgedeckt werden.

7.2 Ausblick

Die Betrachtungen des vorangegangenen Abschnitts haben deutlich gemacht, daß die Ziele, die in dieser Arbeit die Entwicklung der Annotationssprache geleitet haben, im großen und ganzen erreicht worden sind. Gleichzeitig hat der praktische Einsatz gezeigt, in welchen Bereichen die Annotationssprache und deren Laufzeitumgebung noch erweitert werden müssen, um eine komfortablere Handhabung und vollständigere Spezifikationsmöglichkeiten zu erlangen.

7.2.1 Verbesserungen des Spezifikationsverfahrens

Im Mittelpunkt der Verbesserungsvorschläge steht die Forderung, die Erzeugung und Veränderung von Referenzobjekten und somit Geflechten zu ermöglichen, ohne darauf zu verzichten, daß die Auswertung von Annotationen seiteneffektfrei verläuft. Denkbar ist die Zulassung von temporären Referenzobjekten, die in keinerlei Beziehung zu den Objekten des Programms stehen. Daher unterliegt die Manipulation dieser Objekte strengen Regeln.

Hauptschwierigkeit dieses Ansatzes ist, wie bereits erwähnt, daß die Überprüfung des funktionalen Charakters von Annotationen statisch erfolgen soll. Eine Annotationssprache, die einen geeigneten Mechanismus bereitstellt, ermöglicht jedoch unter anderem den Vergleich von Geflechten über verschiedene Zustände der Ausführung hinweg, was den Gehalt der Spezifikation wesentlich erhöht.

Eine weitere Verbesserung der Spezifikationsmöglichkeiten kann durch die Kennzeichnung der Konstruktormethoden erzielt werden. Da in Sather Konstruktoren beliebige Namen haben können, muß diese Markierung durch die Annotationssprache erfolgen. Beim Aufruf von Konstruktoren werden die Objektinvarianten dann nicht für das `self`-Objekt, sondern für das erzeugte Objekt geprüft.

Dadurch können auch solche Invarianten angegeben werden, die vom `void`-Objekt verletzt werden, was bisher nicht möglich ist.

Besonders bei der Einbindung algebraischer Spezifikationen ist häufig die Verwendung von Quantoren erforderlich. Dabei spielt vor allem der Allquantor eine bedeutende Rolle, da der Exisitenzquantor meist durch das Objekt, dessen Existenz nachgewiesen werden soll, ersetzt werden kann. Bedingungen, die für jedes Objekt einer Klasse gelten müssen, werden in abgeschwächter Form als Annotation eingebracht, indem die Gültigkeit für ein bestimmtes Objekt gezeigt wird (vgl. Abschnitt 6.2.1).

Da die Menge der verschiedenen Objekte einer Klasse meist extrem groß ist, können Allquantoren nicht in die Annotationssprache integriert werden, ohne die dynamische Auswertbarkeit der Bedingungen zu beeinträchtigen. Denkbar wäre jedoch z. B. die Erzeugung zufälliger Objekte, um nach einer gewissen Anzahl von Test wenigstens statistische Aussagen über die Erfüllung der Annotation treffen zu können.

Des weiteren ist es wünschenswert, neben Initial- und Terminalzustand weitere ausgezeichnete Zustände definieren und ansprechen zu können. Interessant wäre es z. B. in diesem Zusammenhang, bei einem Schleifendurchlauf auf die Werte früherer Durchläufe zurückgreifen zu können.

Zur besseren Unterstützung der Iteration können auch Schleifeninvarianten und -varianten nach dem Vorbild von Eiffel dienen. Sie ermöglichen eine Beschreibung des Verhaltens der Iteration und erleichtern dadurch z. B. Terminierungsbeweise.

Die Einführung von Konstrukten zur Behandlung von Geflechten bringt zwar keine neuen Spezifikationsmöglichkeiten, vereinfacht jedoch die Handhabung der Annotationssprache. Verfahren zur Feststellung von Überlappungen zweier Geflechte im Speicher oder ähnlichen Beziehungen zwischen Objekten könnten von SALSA bereitgestellt oder aber als Bibliotheksfunktionen angeboten werden.

7.2.2 Verbesserungen der Laufzeitumgebung

Die Auswertung der Annotationen zur Laufzeit wird bisher lediglich dazu genutzt, den Entwickler über Annotationsverletzungen zu informieren. Der Interpreter reagiert stets mit dem Abbruch der Programmausführung. Für die Erstellung robuster Softwaresysteme ist dieses Verhalten keineswegs ausreichend. Nur durch Einsatz von Mechanismen der Ausnahmebehandlung kann sinnvoll auf Programm- und Systemfehler reagiert werden.

Die Erweiterung des SALSA-Sytems um eine Ausnahmebehandlung, die sich nahtlos in das Konzept von *Design by Contract* einfügt, ermöglicht weitergehende Untersuchungen über die Arbeitsweise von Annotationssprachen. Auch das Auftreten von Systemfehlern könnte dann von Annotationen erkannt und vom System behandelt werden.

7.2.3 Weiterführung des SALSA-Konzeptes

Die vorliegende Arbeit zeigt, wie Annotationssprachen für objektorientierte Progammiersprachen entworfen, implementiert und eingesetzt werden können. Zielrichtung bei der Entwicklung war dabei die Unterstützung des Programmierers durch dynamische Abprüfungen.

Die Verifikation der Programme — ein mindestens ebenso interessanter Aspekt der Annotierung — erfordert zum Teil andere Konzepte, die die automatische Führung eines Korrektheitsbeweises erleichtern, im Gegenzug jedoch die dynamische Auswertung der Annotationen verhindern. Beiden Ansätzen gemein sind die formale Spezifikation und Dokumentation der Programmeigenschaften.

In weiterführenden Arbeiten kann durch nicht allzugroße Abwandlungen des SALSA-Konzepts die Schwerpunktsetzung auf den Bereich der Verifikation verlagert werden. Durch die formale Beschreibung der Semantik von Annotations- und Basissprache wurden hierfür bereits wesentliche Vorarbeiten geleistet.

Anhang A

Programmbeispiele

A.1 Die Klasse $IS_EQ

```
 (1) type $IS_EQ
 (2) is functional is_eq(x: $OB): BOOL;
 (3)    functional is_neq(x: $OB): BOOL
 (4) end; -- $IS_EQ
```

A.2 Die Klasse FRACTION

```
 (5) value class FRACTION < $IS_EQ
 (6) invariant  pos_numerator:   num >= 0,
 (7)            pos_denominator: den >= 0,
 (8)            wellformed:      den /= 0 => ggT(num, den) = 1
 (9) is private attr sign: BOOL;
(10)    private attr num: INT;
(11)    readonly attr den: INT;

(12)    numerator: INT
(13)    ensures  abs:  result.abs = num,
(14)             sign: result < 0 = sign
(15)    is if sign then return -num
(16)       else return num end
(17)    end; -- numerator

(18)    private functional ggT(a, b: INT): INT
(19)    requires pos_a: a >= 0,
(20)             pos_b: b >= 0
(21)    ensures  divides_a: initial(a) % result = 0,
(22)             divides_b: initial(b) % result = 0
(23)    is min: INT := b.min(a);
(24)       max: INT := b.max(a);
(25)       if min = 0 then return max
(26)       else return ggT(min, max % min) end
(27)    end; -- ggT
```

```
(28)    functional create(n, d: INT): SAME
(29)    requires correct_denominator: d /= 0
(30)    ensures  correct_denominator: result.den /= 0,
(31)             correct_sign:        result.sign =
(32)                 ((initial(n) > 0 and initial(d) < 0) or
(33)                 (initial(n) < 0 and initial(d) > 0))
(34)    is k: INT := ggT(n.abs, d.abs);
(35)       return FRACTION::num(n.abs / k).den(d.abs / k).sign(n * d < 0);
(36)    end; -- create

(37)    functional create(n: INT): SAME
(38)    ensures  correct_denominator: result.den /= 0,
(39)             correct_sign:        result.sign = (initial(n) < 0)
(40)    is return create(n, 1)
(41)    end; -- create

(42)    functional is_eq(a: $OB): BOOL
(43)    is typecase a
(44)       when SAME then return SYS::ob_eq(self, a)
(45)       when INT  then return self = create(a)
(46)       else return false end
(47)    end; -- is_eq

(48)    functional is_neq(a: $OB): BOOL
(49)    is return ~is_eq(a) end;

(50)    plus(a: SAME): SAME
(51)    ensures  correct_denominator: result.den /= 0
(52)    is return create(numerator * a.den + a.numerator * den, den * a.den)
(53)    end; -- plus

(54)    minus(a: SAME): SAME
(55)    ensures  correct_denominator: result.den /= 0
(56)    is return create(numerator * a.den - a.numerator * den, den * a.den)
(57)    end; -- minus

(58)    times(a: SAME): SAME
(59)    ensures  correct_denominator: result.den /= 0
(60)    is return create(numerator * a.numerator, den * a.den)
(61)    end; -- times

(62)    div(a: SAME): SAME
(63)    requires div_by_zero: a /= create(0)
(64)    ensures  correct_denominator: result.den /= 0
(65)    is return create(numerator * a.den, den * a.numerator)
(66)    end; -- div
(67) end; -- FRACTION
```

A.3 Die Klasse $STACK

```
 (68) type $STACK < $IS_EQ
 (69) invariant  len: length >= 0
 (70) is create: $STACK
 (71)    ensures  law_L1: result.length = 0,
 (72)             law_E1: result.isempty;

 (73)    functional isempty: BOOL;

 (74)    append(x: $IS_EQ): $STACK
 (75)    ensures  law_R:  result.rest = initial(self),
 (76)             law_T:  result.top = initial(x),
 (77)             law_L2: result.length = initial(length) + 1,
 (78)             law_E2: ~result.isempty;

 (79)    functional rest: $STACK
 (80)    requires not_empty: ~isempty;

 (81)    functional top: $IS_EQ
 (82)    requires not_empty: ~isempty;

 (83)    functional length: INT;
 (84) end; -- $STACK
```

A.4 Die Klasse STACK

```
 (85) class STACK < $STACK, $IS_EQ
 (86) invariant  len:        length >= 0,
 (87)            wellformed: length > 1 => ~is_referenced(r)
 (88) is readonly attr length: INT;
 (89)    private  attr elem:   $IS_EQ;
 (90)    private  attr r:      SAME;

 (91)    private functional is_referenced(back: SAME): BOOL
 (92)    is if back.isempty then return false
 (93)       elsif SYS::ob_eq(self, back) then return true
 (94)       else return is_referenced(back.r) end
 (95)    end; -- is_referenced

 (96)    functional is_eq(a: $OB): BOOL
 (97)    is assert not_void: ~void(self);
 (98)       typecase a
 (99)       when SAME then
(100)         if isempty then return a.isempty
(101)         else return elem = a.elem and r = a.r end
(102)       else return false end;
(103)    end; -- is_eq

(104)    functional is_neq(a: $OB): BOOL
(105)    is return ~is_eq(a) end;
```

```
(106)    create: SAME
(107)    ensures  law_L1: result.length = 0,
(108)             law_E1: result.isempty
(109)    is return new end;

(110)    functional isempty: BOOL
(111)    is assert not_void: ~void(self);
(112)       return length = 0
(113)    end; -- isempty

(114)    append(a: $IS_EQ): SAME
(115)    ensures  law_R:  result.rest = initial(self),
(116)             law_T:  result.top = initial(a),
(117)             law_L2: result.length = initial(length) + 1,
(118)             law_E2: ~result.isempty
(119)    is assert not_void: ~void(self);
(120)       s: SAME  := new;
(121)       s.elem   := a;
(122)       s.r      := self;
(123)       s.length := length + 1;
(124)       return s;
(125)    end; -- append

(126)    functional rest: SAME
(127)    requires not_empty: ~isempty
(128)    is return r end;

(129)    functional top: $IS_EQ
(130)    requires not_empty: ~isempty
(131)    is return elem end;
(132) end; -- STACK
```

A.5 Die Klasse CALC

```
(133) class CALC < $STACK
(134) invariant  len:        length >= 0,
(135)            wellformed: length > 1 => ~is_referenced(r)
(136) is include STACK;

(137)    compute(op: CHAR): SAME
(138)    requires
(139)      operation:       op = '+' or op = '-' or op = '*' or op = '/',
(140)      operands:        length >= 2,
(141)      div_by_zero:     op = '/' => top /= #FRACTION(0),
(142)      partype1:        SYS::str_for_tp(SYS::tp(top)) = "FRACTION",
(143)      partype2:        SYS::str_for_tp(SYS::tp(rest.top)) = "FRACTION"
(144)    ensures
(145)      remove_operands: result.length = initial(length) - 1,
(146)      result_type:     SYS::str_for_tp(SYS::tp(result.top)) = "FRACTION"
(147)    is f: FRACTION;
```

```
(148)       b: $IS_EQ := top;
(149)       a: $IS_EQ := r.top;
(150)       typecase a
(151)       when FRACTION then
(152)         typecase b
(153)         when FRACTION then
(154)           case op
(155)           when '+' then f := a + b
(156)           when '-' then f := a - b
(157)           when '*' then f := a * b
(158)           when '/' then f := a / b
(159)           end end end;
(160)       return r.r.append(f)
(161)    end; -- compute
(162) end; -- CALC
```

A.6 Die Klasse MAIN

```
(163) class MAIN
(164) is private attr input: STR;

(165)    main(s: STR)
(166)    is if s.is_empty then return end;
(167)       input := s;
(168)       c: CALC := #CALC; c := parse(c);
(169)       if c.length = 1 then
(170)         r: $IS_EQ := c.top;
(171)         typecase r
(172)         when FRACTION then
(173)           INOUT::write_string("Result: ").write_int(r.numerator);
(174)           INOUT::write_char(':').write_int(r.den).nl
(175)         end -- typecase
(176)       else INOUT::write_string("Incomplete term").nl; end
(177)    end; -- main

(178)    private parse(c: CALC): CALC
(179)    is if is_operator(lookahead) and
(180)          (input.rest.is_empty or ~is_digit(input.rest.first))
(181)       then c := comp(c, getchar)
(182)       else c := c.append(fraction) end;
(183)       if input.is_empty then return c
(184)       elsif getchar = ',' then c := parse(c)
(185)       else error("Comma or end of input expected") end;
(186)       return c
(187)    end; -- parse

(188)    private error(s: STR)
(189)    is INOUT::write_string(s).nl;
(190)       SYS::exit(0)
(191)    end; -- error
```

```
(192)    private is_operator(c: CHAR): BOOL
(193)    is case c
(194)       when '+', '-', '*', '/' then return true
(195)       else return false end;
(196)    end; -- is_operator
(197)    private is_digit(c: CHAR): BOOL
(198)    is case c
(199)       when '0', '1', '2', '3', '4', '5', '6', '7', '8', '9'
(200)       then return true
(201)       else return false end
(202)    end; -- is_digit

(203)    private lookahead: CHAR
(204)    is if input.is_empty then error("Unexpected end of line")
(205)       elsif input.first = ' ' then
(206)         input := input.rest; return lookahead
(207)       else return input.first end
(208)    end; -- lookahead

(209)    private getchar: CHAR
(210)    is r: CHAR := lookahead;
(211)       input := input.rest; return r
(212)    end; -- getchar

(213)    private number: INT
(214)    is temp: STR;
(215)       if lookahead = '-' then temp := temp + getchar end;
(216)       loop
(217)         if ~input.is_empty and is_digit(lookahead)
(218)         then temp := temp + getchar
(219)         else break! end
(220)       end; -- loop
(221)       return temp.atoi
(222)    end; -- number

(223)    private fraction: FRACTION
(224)    is d: INT := number;
(225)       n: INT;
(226)       if ~input.is_empty and lookahead = ':'
(227)       then getchar; n := number
(228)       else n := 1 end;
(229)       return #FRACTION(d, n)
(230)    end; -- fraction

(231)    comp(c: CALC, op: CHAR): CALC
(232)    is if c.length < 2 then error("Not enough operands")
(233)       elsif op = '/' and c.top = #FRACTION(0) then
(234)         error("Division by zero")
(235)       else return c.compute(op) end
(236)    end; -- comp
(237) end -- MAIN
```

Anhang B

Die Implementierung des Interpreters

B.1 Die vordefinierten Klassen

```
value class INT is
    functional plus(x: SAME): SAME;
    functional minus(x: SAME): SAME;
    functional times(x: SAME): SAME;
    functional div(x: SAME): SAME
    requires
        div_by_zero: x.is_neq(0);
    functional pow(x: SAME): SAME
    requires
        pos_exponent: x.is_geq(0);
    functional mod(x: SAME): SAME
    requires
        mod_by_zero: x.is_neq(0);
    functional is_lt(x: SAME): BOOL;
    functional is_leq(x: SAME): BOOL;
    functional is_eq(x: SAME): BOOL;
    functional is_neq(x: SAME): BOOL;
    functional is_gt(x: SAME): BOOL;
    functional is_geq(x: SAME): BOOL;
    functional negate: SAME;
    functional abs: SAME;
    functional min(x: SAME): SAME;
    functional max(x: SAME): SAME;
end;

value class BOOL is
    functional not: SAME;
    functional is_eq(x: SAME): SAME;
    functional is_neq(x: SAME): SAME;
end;
```

```
value class CHAR is
    functional ctoa: STR;
    functional is_lt(x: SAME): BOOL;
    functional is_leq(x: SAME): BOOL;
    functional is_eq(x: SAME): BOOL;
    functional is_neq(x: SAME): BOOL;
    functional is_gt(x: SAME): BOOL;
    functional is_geq(x: SAME): BOOL;
end;

value class STR is
    functional plus(x: SAME): SAME;
    functional plus(x: CHAR): SAME;
    functional is_lt(x: SAME): BOOL;
    functional is_leq(x: SAME): BOOL;
    functional is_eq(x: SAME): BOOL;
    functional is_neq(x: SAME): BOOL;
    functional is_gt(x: SAME): BOOL;
    functional is_geq(x: SAME): BOOL;
    functional atoi: INT;
    functional first: CHAR
    requires
        not_empty: self.is_empty.not;
    functional rest: SAME
    requires
        not_empty: self.is_empty.not;
    functional strlen: INT;
    functional is_empty: BOOL;
end;

class INOUT is
    write_int(x: INT): SAME;
    write_char(x: CHAR): SAME;
    write_string(x: STR): SAME;
    read_int: INT;
    read_char: CHAR;
    read_string: STR;
    nl: SAME;
end;

class SYS is
    functional ob_eq(x: $OB, y: $OB): BOOL;
    functional tp(x: $OB): INT;
    functional str_for_tp(x: INT): STR;
    exit(x: INT): INT;
end;

type $OB is
end;
```

B.2 Der Quellcode

B.2.1 Der Scanner

```
/*************************************************************************
 * Module:              SALSA_scan.l                                     *
 * Author:              Peter Müller                                     *
 * Description:         Scanner specification of                         *
 *                      Sather/SALSA interpreter and AutoDoc             *
 *************************************************************************/
%{
#include <stdlib.h>
#ifdef AUTO
    #include "AutoDoc_spec.h"
#else
    #include "SALSA_spec.h"
#endif
#include "y.tab.h"

char cltoc(char *str);

int yylineno = 1;
%}

comment                 --.*
whitespace              [ \t\v\b\r\f]|{comment}
newline                 [\n]

lowercase_letter        [a-z]
uppercase_letter        [A-Z]
letter                  {lowercase_letter}|{uppercase_letter}
escape_seq              [abfnrtv\'"]|({octal_digit}+)

decimal_digit           [0-9]
octal_digit             [0-7]
binary_digit            [01]
hex_digit               [a-f0-9]

binary_int              0b({binary_digit}|[_])*
octal_int               0o({octal_digit}|[_])*
decimal_int             {decimal_digit}({decimal_digit}|[_])*
hex_int                 0x({hex_digit}|[_])*

identifier              {letter}({letter}|{decimal_digit}|[_])*
uppercase_identifier    {uppercase_letter}({uppercase_letter}
                        |{decimal_digit}|[_])*
abstract_type_name      \${uppercase_identifier}

char_literal_expression '(.|\\{escape_seq})'
str_literal_expression  \"[^"]*\"
```

```
%%

{whitespace}    /* ignore */ ;
{newline}       { yylineno++; }
{binary_int}    { yylval.s = itoe(any_to_decimal(yytext, 2));
                  return (INTCONST); }
{octal_int}     { yylval.s = itoe(any_to_decimal(yytext, 8));
                  return (INTCONST); }
{decimal_int}   { yylval.s = itoe(atoi(yytext)); return (INTCONST); }
{hex_int}       { yylval.s = itoe(any_to_decimal(yytext, 16));
                  return (INTCONST); }
"and"           { return (AND); }
"assert"        { return (ASSERT); }
"attr"          { return (ATTR); }
"break!"        { return (BREAK); }
"case"          { return (CASE); }
"class"         { return (CLASS); }
"else"          { return (ELSE); }
"elsif"         { return (ELSIF); }
"end"           { return (END); }
"ensures"       { return (ENSURES); }
"false"         { yylval.s = btoe(0); return (BOOLCONST); }
"functional"    { return (FUNCTIONAL); }
"if"            { return (IF); }
"include"       { return (INCLUDE); }
"initial"       { return (INITIAL_SY); }
"is"            { return (IS); }
"invariant"     { return (INVARIANT); }
"loop"          { return (LOOP); }
"new"           { return (NEW); }
"or"            { return (OR); }
"private"       { return (PRIVATE); }
"readonly"      { return (READONLY); }
"requires"      { return (REQUIRES); }
"result"        { return (RESULT); }
"return"        { return (RETURN); }
"SAME"          { return (SAME); }
"self"          { return (SELF); }
"then"          { return (THEN); }
"true"          { yylval.s = btoe(1); return (BOOLCONST); }
"type"          { return (TYPE); }
"typecase"      { return (TYPECASE); }
"value"         { return (VALUE); }
"void"          { return (VOID); }
"when"          { return (WHEN); }
"while!"        { return (WHILE); }
"("             { return ('('); }
")"             { return (')'); }
"["             { yylval.s = itoe(yylineno); return ('['); }
"]"             { yylval.s = itoe(yylineno); return (']'); }
","             { return (','); }
```

```
"."             { return ('.'); }
";"             { return (';'); }
":"             { return (':'); }
":"/":="        { return (':'); }
"+"             { yylval.s = itoe(yylineno); return ('+'); }
"-"             { yylval.s = itoe(yylineno); return ('-'); }
"*"             { yylval.s = itoe(yylineno); return ('*'); }
"/"             { yylval.s = itoe(yylineno); return ('/'); }
"="             { yylval.s = itoe(yylineno); return ('='); }
"<"             { yylval.s = itoe(yylineno); return ('<'); }
">"             { yylval.s = itoe(yylineno); return ('>'); }
"#"             { yylval.s = itoe(yylineno); return ('#'); }
"^"             { yylval.s = itoe(yylineno); return ('^'); }
"%"             { yylval.s = itoe(yylineno); return ('%'); }
"~"             { yylval.s = itoe(yylineno); return ('~'); }
"/="            { yylval.s = itoe(yylineno); return (NEQ); }
"<="            { yylval.s = itoe(yylineno); return (LEQ); }
">="            { yylval.s = itoe(yylineno); return (GEQ); }
":="            { return (BECOMES); }
"::"            { return (SCOPE); }
"->"            { return (ARROW); }
"=>"            { yylval.s = itoe(yylineno); return (IMPLICATION); }

{uppercase_identifier}      { yylval.i.value = stoid(atoe(yytext));
                              yylval.i.line = itoe(yylineno);
                              return (UPPERCASE_IDENTIFIER); }
{identifier}                { yylval.i.value = stoid(atoe(yytext));
                              yylval.i.line = itoe(yylineno);
                              return (IDENTIFIER); }
{abstract_type_name}        { yylval.i.value = stoid(atoe(yytext));
                              yylval.i.line = itoe(yylineno);
                              return (ABSTRACT_TYPE_NAME); }
{char_literal_expression}   { yylval.s = ctoe(cltoc(yytext));
                              return (CHARCONST); }
{str_literal_expression}    { yylval.s = atoe(yytext); return (STRCONST); }

%%

/*
 * Signature:   pot(int base, int exp)
 * Purpose:     computes base^exp
 */
int pot(int base, int exp)
{
int res = 1;

for (; exp>0; exp--)
    res*=base;

return res;
}
```

```
/*
 * Signature:   any_to_decimal(char *bin_str, int base)
 * Purpose:     converts binary, octal and hexadecimal to decimal integers
 */
int any_to_decimal(char *bin_str, int base)
{
int sign = 1;
int res = 0;
int stelle = 0;
char *ptr;

if (*bin_str == '-')
    {
    sign = -1;
    bin_str++;
    }

bin_str +=2;                    /* skip 0_  */
for (ptr = bin_str+strlen(bin_str)-1; ptr>= bin_str; ptr--)
    if (*ptr >= '0' && *ptr <= '9')
        {
        res += (int)(*ptr - '0') * pot(base,stelle);
        stelle++;
        }
    else
        if (*ptr >= 'a' && *ptr <= 'f')
            {
            res += ((int)(*ptr - 'a') +10) * pot(base,stelle);
            stelle++;
            }

return sign * res;
}

/*
 * Signature:   cltoc(char *str)
 * Purpose:     turns character sequences into special characters
 */
char cltoc(char *str)
{
str++;
if (*str == '\\')
    {
    str++;
    switch (*str)
        {
        case 'a':     return '\a';
        case 'b':     return '\b';
        case 'f':     return '\f';
        case 'n':     return '\n';
```

```
        case 'r':     return '\r';
        case 't':     return '\t';
        case 'v':     return '\v';
        case '\\':    return '\\';
        case '\'':    return '\'';
        case '\"':    return '\"';
        default:      return (char)any_to_decimal(str, 8);
        }
    }
else
    return *str;
}
```

B.2.2 Der Parser

```
/*************************************************************************
 * Module:              SALSA_pars.y                                     *
 * Author:              Peter Müller                                     *
 * Description:         yacc specification of parser of                  *
 *                      Sather/SALSA interpreter and AutoDoc             *
 *************************************************************************/

%{
#include <stdio.h>
#include <stdlib.h>
#ifdef AUTO
    #include "AutoDoc_spec.h"
#else
    #include "SALSA_spec.h"
#endif

/*************************************************************************
 * macro definitions
 *************************************************************************/

#define ARRAY_SIZE 7

/*************************************************************************
 * global variables
 *************************************************************************/

static ELEMENT syntaxtree = nil();
struct bst_Elem *symtab = NULL;

/*************************************************************************
 * data types for symbol table
 *************************************************************************/

struct List_Elem
    {
    char *id;
    struct List_Elem *next;
    };

struct bst_Elem
    {
    struct List_Elem *hash_table[ARRAY_SIZE];
    struct bst_Elem *next;
    };
```

```
/*************************************************************************
 * data types for type graph and include graph
 *************************************************************************/

typedef struct edge
    {
    ELEMENT super;
    ELEMENT sub;
    struct edge *next;
    } edge;

typedef struct node
    {
    ELEMENT type;
    int id;
    struct node *next;
    } node;

typedef int *graph_matrix;

typedef struct
    {
    node *first_node;
    int nodes;
    edge *first_edge;
    graph_matrix matrix;
    graph_matrix trans;
    ELEMENT *class_name;
    } graph;

graph type_graph = {NULL, 0, NULL, NULL, NULL};
graph include_graph = {NULL, 0, NULL, NULL, NULL};

#define TYPE_GRAPH      0
#define INCLUDE_GRAPH   1

/*************************************************************************
 * external functions
 *************************************************************************/

extern ELEMENT add_predecl_classes (ELEMENT s);
extern char cltoc(char *str);
extern ELEMENT concat(ELEMENT s1, ELEMENT s2);
extern char *yytext;
extern int yylineno,yyleng;
extern FILE* yyin;
```

```
/*************************************************************************
 * function proto types
 *************************************************************************/

ELEMENT idlist_to_arglist(ELEMENT idlist, ELEMENT type);
ELEMENT build_attr_def_list(ELEMENT, ELEMENT, ELEMENT);
ELEMENT change_writers(ELEMENT cel);
ELEMENT decllist_to_stmtlist(ELEMENT idlist, ELEMENT type);
ELEMENT remove_elseif(ELEMENT i, ELEMENT t, ELEMENT list, ELEMENT e);
ELEMENT build_variante_list(ELEMENT list, ELEMENT sl);
ELEMENT remove_case(ELEMENT e, ELEMENT vlist, ELEMENT sl);
ELEMENT access_read(ELEMENT id, ELEMENT receiver);
ELEMENT access_write(ELEMENT id, ELEMENT receiver, ELEMENT par);
ELEMENT insert_ident(ELEMENT ident);
char lookup_ident(ELEMENT ident);
void open_block(void);
void close_block(void);
void insert_node(ELEMENT type, int counter, graph *g);
void insert_edge(ELEMENT super, ELEMENT sub, graph *g);
void add_subtyping_clause_to_type_graph(ELEMENT, ELEMENT);
void add_supertyping_clause_to_type_graph(ELEMENT, ELEMENT);
void insert_including_class(ELEMENT class);
void build_graph(graph *g, char *text);
int yyparse(void);
%}

/*************************************************************************
 * data type for scanner and parser
 *************************************************************************/

%union {
    ELEMENT s;
    struct {
        ELEMENT value;
        ELEMENT line;
        } i;
    }

/*************************************************************************
 * tokens
 *************************************************************************/

%token<s> INTCONST
%token<s> BOOLCONST CHARCONST STRCONST
%token    ASSERT ATTR BREAK CASE CLASS ELSE ELSIF END ENSURES
%token    FUNCTIONAL IF INCLUDE INITIAL_SY IS INVARIANT LOOP NEW
%token    PRIVATE READONLY REQUIRES RESULT RETURN SAME SELF THEN TYPE
%token    TYPECASE VALUE VOID WHEN WHILE
%token    ',' ';' ':'
%token<s> '#'
%token    BECOMES ARROW
```

```
%token<i> UPPERCASE_IDENTIFIER IDENTIFIER ABSTRACT_TYPE_NAME
%token<s> '[' ']'

/*****************************************************************************
 * Precedence ordering according to p. 24
 *****************************************************************************/

%left<s> IMPLICATION
%left    AND OR
%left<s> '<' LEQ '=' NEQ GEQ '>'
%left<s> '+' '-'
%left<s> '*' '/' '%'
%left<s> '~'
%left<s> '^'
%left    '.' SCOPE '(' ')' '[' ']'

/*****************************************************************************
 * types
 *****************************************************************************/

%type<s>  typedef_or_class_list typedef_or_class
%type<s>  abstract_type_definition subtyping_clause supertyping_clause
%type<s>  abstract_signature_list abstract_signature
%type<s>  argument_list argument_seq argument
%type<s>  type_specifier type_specifier_seq type_specifier_list
%type<s>  class private_or_public access_mode return_type
%type<s>  class_element_list class_element
%type<s>  used_ident decl_ident identifier_list
%type<s>  attr_definition routine_definition
%type<s>  include_clause
%type<s>  pre_condition post_condition
%type<s>  statement_list statement_seq statement
%type<s>  declaration_statement assign_statement if_statement
%type<s>  return_statement case_statement loop_statement
%type<s>  typecase_statement assert_statement
%type<s>  expression_statement
%type<s>  expression_seq expression_list expression
%type<s>  self_expression local_expression call_expression
%type<s>  void_test_expression new_expression create_expression
%type<s>  sugar_expression and_expression or_expression
%type<s>  initial_expression result_expression while_expression
%type<s>  break_expression implication_expression void_expression
%type<s>  bool_literal_expression int_literal_expression
%type<s>  char_literal_expression str_literal_expression
%type<s>  feature_modifier_list feature_modifier_seq feature_modifier
%type<s>  then_branch else_branch elseif_list
%type<s>  when_type_list when_expr_list act_parlist
%type<s>  condition_list condition invariant_clause function_mode
%type<i>  identifier

%start source_file
```

```
%%

/*************************************************************************
 * productions
 *************************************************************************/

source_file
: typedef_or_class_list
  { insert_node(stoid(atoe("$OB")), TYPE_GRAPH, &type_graph);
    insert_node(stoid(atoe("$OB")), INCLUDE_GRAPH, &include_graph);
    insert_node(stoid(atoe("BOOL")), TYPE_GRAPH, &type_graph);
    insert_node(stoid(atoe("BOOL")), INCLUDE_GRAPH, &include_graph);
    insert_edge(stoid(atoe("$OB")), stoid(atoe("BOOL")), &type_graph);
    insert_node(stoid(atoe("INT")), TYPE_GRAPH, &type_graph);
    insert_node(stoid(atoe("INT")), INCLUDE_GRAPH, &include_graph);
    insert_edge(stoid(atoe("$OB")), stoid(atoe("INT")), &type_graph);
    insert_node(stoid(atoe("INOUT")), TYPE_GRAPH, &type_graph);
    insert_node(stoid(atoe("INOUT")), INCLUDE_GRAPH, &include_graph);
    insert_edge(stoid(atoe("$OB")), stoid(atoe("INOUT")), &type_graph);
    insert_node(stoid(atoe("SYS")), TYPE_GRAPH, &type_graph);
    insert_node(stoid(atoe("SYS")), INCLUDE_GRAPH, &include_graph);
    insert_edge(stoid(atoe("$OB")), stoid(atoe("SYS")), &type_graph);
    insert_node(stoid(atoe("CHAR")), TYPE_GRAPH, &type_graph);
    insert_node(stoid(atoe("CHAR")), INCLUDE_GRAPH, &include_graph);
    insert_edge(stoid(atoe("$OB")), stoid(atoe("CHAR")), &type_graph);
    insert_node(stoid(atoe("STR")), TYPE_GRAPH, &type_graph);
    insert_node(stoid(atoe("STR")), INCLUDE_GRAPH, &include_graph);
    insert_edge(stoid(atoe("$OB")), stoid(atoe("STR")), &type_graph);

    build_graph(&type_graph, "type");
    build_graph(&include_graph, "include");
    syntaxtree = add_predecl_classes($1); }
;

typedef_or_class_list
: typedef_or_class
  { $$ = ($1 == nil()) ? Sourcefile() : appback(Sourcefile(), $1); }
| typedef_or_class_list ';' typedef_or_class
  { $$ = ($3 == nil()) ? $1 : appback($1, $3); }
;

typedef_or_class
: /* empty */
  { $$ = nil(); }
| abstract_type_definition
  { $$ = $1; }
| class
  { $$ = $1; }
;
```

```
/*************************************************************************
 * types
 *************************************************************************/

abstract_type_definition
: TYPE ABSTRACT_TYPE_NAME subtyping_clause supertyping_clause
  invariant_clause IS abstract_signature_list END
  { insert_node($2.value, TYPE_GRAPH, &type_graph);
    insert_edge(stoid(atoe("$OB")), $2.value, &type_graph);
    add_subtyping_clause_to_type_graph($2.value, $3);
    add_supertyping_clause_to_type_graph($2.value, $4);
    $$ = AbstractClass(DeclId($2.value, $2.line), $3, $4, $5, $7); }
;

subtyping_clause
: /* empty */
  { $$ = Typespec_List(); }
| '<' type_specifier_list
  { $$ = $2; }
;

supertyping_clause
: /* empty */
  { $$ = Typespec_List(); }
| '>' type_specifier_list
  { $$ = $2; }
;

invariant_clause
: /* empty */
  { $$ = Condition_List(); }
| INVARIANT condition_list
  { $$ = $2; }
;

abstract_signature_list
: abstract_signature
  { $$ = $1; }
| abstract_signature_list ';' abstract_signature
  { $$ = conc($1, $3); }
;

abstract_signature
: /* empty */
  { $$ = Signature_List(); }
| function_mode decl_ident argument_list return_type
  pre_condition post_condition
  { close_block(); $$ = appback(Signature_List(),
                        Signature($2, $1, $3, $4, $5, $6)); }
;
```

```
argument_list
: /* empty */
  { open_block(); $$ = Argument_List(); }
| '(' argument_seq argument ')'
  { $$ = conc($2, $3); }
;

argument_seq
: /* empty */
  { open_block(); $$ = Argument_List(); }
| argument_seq argument ','
  { $$ = conc($1, $2); }
;

argument
: identifier_list ':' type_specifier
  { $$ = idlist_to_arglist($1, $3); }
;

/*****************************************************************************
 * Identifiers
 *****************************************************************************/

identifier_list
: decl_ident
  { $$ = appback(DeclId_List(), $1) ; }
| identifier_list ',' decl_ident
  { $$ = appback($1, $3); }
;

identifier
: IDENTIFIER
  { $$ = $1; }
| UPPERCASE_IDENTIFIER
  { $$ = $1; }
;

used_ident
: identifier
  { $$ = UsedId($1.value, $1.line); }
;

decl_ident
: identifier
  { $$ = DeclId($1.value, $1.line); }
;
```

```
/*************************************************************************
 * conditions
 *************************************************************************/

condition_list
: condition
  { $$ = appback(Condition_List(), $1); }
| condition_list ',' condition
  { $$ = appback($1, $3); }
;

condition
: decl_ident ':' expression
  { $$ = Condition($1, $3); }
;

/*************************************************************************
 * classes
 *************************************************************************/

class
: /* Reference*/ CLASS UPPERCASE_IDENTIFIER subtyping_clause
  invariant_clause IS class_element_list END
  { insert_node($2.value, TYPE_GRAPH, &type_graph);
    insert_node($2.value, INCLUDE_GRAPH, &include_graph);
    insert_edge(stoid(atoe("$OB")), $2.value, &type_graph);
    add_subtyping_clause_to_type_graph($2.value, $3);
    insert_including_class($2.value);
    $$ = RefClass(DeclId($2.value, $2.line), $3, $4, $6); }
| VALUE CLASS UPPERCASE_IDENTIFIER subtyping_clause
  invariant_clause IS class_element_list END
  { insert_node($3.value, TYPE_GRAPH, &type_graph);
    insert_node($3.value, INCLUDE_GRAPH, &include_graph);
    insert_edge(stoid(atoe("$OB")), $3.value, &type_graph);
    add_subtyping_clause_to_type_graph($3.value, $4);
    insert_including_class($3.value);
    $$ = ValueClass(DeclId($3.value, $3.line), $4, $5, change_writers($7)); }
;

class_element_list
: class_element
  { $$ = $1; }
| class_element_list ';' class_element
  { $$ = conc($1, $3); }
;

class_element
: /* empty */
  { $$ = Class_Element_List(); }
| attr_definition
  { $$ = $1; }
```

```
| routine_definition
  { $$ = appback(Class_Element_List(), $1); }
| include_clause
  { $$ = appback(Class_Element_List(), $1); }
;

/*************************************************************************
 * type specifiers
 *************************************************************************/

type_specifier_list
: type_specifier_seq type_specifier
  { $$ = appback($1, $2); }
;

type_specifier_seq
: /* empty */
  { $$ = Typespec_List(); }
| type_specifier_seq type_specifier ','
  { $$ = appback($1, $2); }
;

type_specifier
: SAME
  { $$ = Same_Type(); }
| UPPERCASE_IDENTIFIER
  { $$ = UsedId($1.value, $1.line); }
| ABSTRACT_TYPE_NAME
  { $$ = UsedId($1.value, $1.line); }
;

/*************************************************************************
 * class elements
 *************************************************************************/

private_or_public
: /* public */
  { $$ = Public(); }
| PRIVATE
  { $$ = Private(); }
;

access_mode
: /* read & write */
  { $$ = Public(); }
| PRIVATE
  { $$ = Private(); }
| READONLY
  { $$ = Readonly(); }
;
```

```
function_mode
: /* not functional */
  { $$ = Not_Functional(); }
| FUNCTIONAL
  { $$ = Functional(); }
;

attr_definition
: access_mode ATTR identifier_list ':' type_specifier
  { $$ = build_attr_def_list($3, $5, $1); }
;

routine_definition
: private_or_public function_mode decl_ident argument_list return_type
  pre_condition post_condition IS statement_list END
  { close_block();    /* Parameter list */
    close_block();    /* Statement list */
    if ($9 != Statement_List() && is(subterm(-1, $9), _Return_Stmt))
      $$ = Routine_Def(FeatureId($1, $3), $2, $4, $5, $6, $7, $9);
    else
      $$ = Routine_Def(FeatureId($1, $3), $2, $4, $5, $6, $7,
           appback($9, Return_Stmt(Void_Expr()))); }
;

return_type
: /* empty */
  { $$ = UsedId(stoid(atoe("INT")), itoe(-1)); }
| ':' type_specifier
  { $$ = $2; }
;

pre_condition
: /* empty */
  { $$ = Condition_List(); }
| REQUIRES condition_list
  { $$ = $2; }
;

post_condition
: /* empty */
  { $$ = Condition_List(); }
| ENSURES condition_list
  { $$ = $2; }
;

include_clause
: private_or_public INCLUDE type_specifier feature_modifier_list
  { insert_edge(NULL, subterm(1, $3), &include_graph);
    $$ = Include_Clause( $1, $3, $4 ); }
;
```

```
feature_modifier_list
: /* empty */
  { $$ = Feature_Modifier_List(); }
| feature_modifier_seq feature_modifier
  { $$ = appback($1, $2); }
;

feature_modifier_seq
: /* empty */
  { $$ = Feature_Modifier_List(); }
| feature_modifier_seq feature_modifier ','
  { $$ = appback($1, $2); }
;

feature_modifier
: used_ident ARROW
  { $$ = $1; }
| used_ident ARROW access_mode decl_ident
  { $$ = Mod_Rename( $1, FeatureId($3, $4) ); }
;

/*************************************************************************
 * Statements
 *************************************************************************/

statement_list
: statement_seq statement
  { $$ = conc( $1, $2 ); }
| statement_seq
  { $$ = $1; }
;

statement_seq
: /* empty */
  { open_block(); $$ = Statement_List(); }
| statement_seq statement ';'
  { $$ = conc( $1, $2 ); }
| statement_seq ';'
  { $$ = $1; }
;

statement
: declaration_statement
  { $$ = $1; }
| assign_statement
  { $$ = $1; }
| if_statement
  { $$ = appback(Statement_List(), $1); }
| loop_statement
  { $$ = appback(Statement_List(), $1); }
| return_statement
```

```
  { $$ = appback(Statement_List(), $1); }
| case_statement
  { $$ = $1; }
| typecase_statement
  { $$ = appback(Statement_List(), $1); }
| assert_statement
  { $$ = appback(Statement_List(), $1); }
| expression_statement
  { if (is($1, _Statement))
      $$ = appback( Statement_List(), $1 );
    else {
      yyerror("expression found where statement expected");
      exit(EXIT_FAILURE); } }
;

declaration_statement
: identifier ',' identifier_list ':' type_specifier
  { $$ = appfront(Typed_Ident(insert_ident(DeclId($1.value, $1.line)), $5),
         decllist_to_stmtlist($3, $5)); }
| identifier ':' type_specifier
  { $$ = appback(Statement_List(), Typed_Ident(insert_ident(
         DeclId($1.value, $1.line)), $3) ); }
;

assign_statement
: expression BECOMES expression
  { if (sort($1) == _Local_Expr)
      $$ = appback(Statement_List(), Assign_Stmt($1, $3));
    else {
      if (sort($1) == _Call_Expr && subterm(3, $1) == Expr_List())
        $$ = appback(Statement_List(), Call_Expr(subterm(1, $1),
             subterm(2, $1), appback(Expr_List(), $3)));
      else {
        fprintf(stderr, "### Illegal lefthand side in assignment\n");
        exit (EXIT_FAILURE); } }
  }
| identifier ':' type_specifier BECOMES expression
  { $$ = appback( appback(Statement_List(), Typed_Ident(
         insert_ident(DeclId($1.value, $1.line)), $3) ),
         access_write(UsedId($1.value, $1.line), Self_Expr(), $5 ) ); }
;

if_statement
: IF expression then_branch elseif_list else_branch END
  { $$ = remove_elseif($2, $3, $4, $5); }
;

then_branch
: THEN statement_list
  { close_block(); $$ = $2; }
;
```

```
elseif_list
: /* empty */
  { $$ = Variante_List(); }
| elseif_list ELSIF expression THEN statement_list
  { close_block(); $$ = appback($1, Variante($3, $5)); }
;

else_branch
: /* empty */
  { $$ = Statement_List(); }
| ELSE statement_list
  { close_block(); $$ = $2; }
;

loop_statement
: LOOP statement_list END
  { close_block(); $$ = Loop($2); }
;

return_statement
: RETURN
  { $$ = Return_Stmt(Void_Expr()); }
| RETURN expression
  { $$ = Return_Stmt($2); }
;

case_statement
: CASE expression when_expr_list else_branch END
  { $$ = remove_case($2, $3, $4); }
;

when_expr_list
: /* empty */
  { $$ = Variante_List(); }
| when_expr_list WHEN expression_seq expression THEN statement_list
  { close_block();
    $$ = conc($1, build_variante_list(appback($3, $4), $6)); }
;

expression_seq
: /* empty */
  { $$ = Expr_List(); }
| expression_seq expression ','
  { $$ = appback($1, $2); }
;

typecase_statement
: TYPECASE identifier when_type_list else_branch END
  { $$ = Typecase_Stmt(Local_Expr(UsedId($2.value, $2.line)), $3, $4);}
;
```

```
when_type_list
: /* empty */
  { $$ = Alternative_List(); }
| when_type_list WHEN type_specifier THEN statement_list
  { close_block(); $$ = appback($1, Alternative($3, $5) ); }
;

assert_statement
: ASSERT condition_list
  { $$ = Assert_Stmt($2); }
;

expression_statement
: expression
  { $$ = $1;}
;

/*************************************************************************
 * Expressions
 *************************************************************************/

expression
: self_expression
  { $$ = $1; }
| local_expression
  { $$ = $1; }
| call_expression
  { $$ = $1; }
| void_expression
  { $$ = $1; }
| void_test_expression
  { $$ = $1; }
| new_expression
  { $$ = $1; }
| create_expression
  { $$ = $1; }
| sugar_expression
  { $$ = $1; }
| and_expression
  { $$ = $1; }
| or_expression
  { $$ = $1; }
| implication_expression
  { $$ = $1; }
| initial_expression
  { $$ = $1; }
| result_expression
  { $$ = $1; }
| while_expression
  { $$ = $1; }
| break_expression
```

```
  { $$ = $1; }
| bool_literal_expression
  { $$ = $1; }
| char_literal_expression
  { $$ = $1; }
| str_literal_expression
  { $$ = remove_quotes($1); }
| int_literal_expression
  { $$ = $1; }
;

self_expression
: SELF
  { $$ = Self_Expr(); }
;

local_expression
: used_ident
  { $$ = access_read($1, Self_Expr()); }
| expression '.' used_ident
  { $$ = Call_Expr($1, $3, Expr_List()); }
| type_specifier SCOPE used_ident
  { $$ = Call_Expr($1, $3, Expr_List()); }
;

call_expression
: used_ident act_parlist
  { $$ = Call_Expr( Self_Expr(), $1, $2 ); }
| expression '.' used_ident act_parlist
  { $$ = Call_Expr( $1, $3, $4 ); }
| type_specifier SCOPE used_ident act_parlist
  { $$ = Call_Expr( $1, $3, $4 ); }
;

act_parlist
: '(' expression_seq expression ')'
  { $$ = appback($2, $3); }
;

expression_list
: /* empty */
  { $$ = Expr_List(); }
| '(' expression_seq expression ')'
  { $$ = appback($2, $3); }
;

void_expression
: VOID
  { $$ = Void_Expr(); }
;
```

```
void_test_expression
: VOID '(' expression ')'
  { $$ = Void_Test_Expr($3); }
;

new_expression
: NEW
  { $$ = New_Expr(); }
;

create_expression
: '#' type_specifier expression_list
  { $$ = Call_Expr( $2, UsedId(stoid(atoe("create")), $1), $3 ); }
;

sugar_expression
: expression '+' expression
  { $$ = Call_Expr( $1, UsedId(stoid(atoe("plus")), $2),
           appback(Expr_List(), $3) ); }
| expression '-' expression
  { $$ = Call_Expr( $1, UsedId(stoid(atoe("minus")), $2),
         appback(Expr_List(), $3) ); }
| expression '*' expression
  { $$ = Call_Expr( $1, UsedId(stoid(atoe("times")), $2),
         appback(Expr_List(), $3) ); }
| expression '/' expression
  { $$ = Call_Expr( $1, UsedId(stoid(atoe("div")), $2),
         appback(Expr_List(), $3) ); }
| expression '^' expression
  { $$ = Call_Expr( $1, UsedId(stoid(atoe("pow")), $2),
         appback(Expr_List(), $3) ); }
| expression '%' expression
  { $$ = Call_Expr( $1, UsedId(stoid(atoe("mod")), $2),
         appback(Expr_List(), $3) ); }
| expression '<' expression
  { $$ = Call_Expr( $1, UsedId(stoid(atoe("is_lt")), $2),
         appback(Expr_List(), $3) ); }
| expression LEQ expression
  { $$ = Call_Expr( $1, UsedId(stoid(atoe("is_leq")), $2),
         appback(Expr_List(), $3) ); }
| expression '=' expression
  { $$ = Call_Expr( $1, UsedId(stoid(atoe("is_eq")), $2),
         appback(Expr_List(), $3) ); }
| expression NEQ expression
  { $$ = Call_Expr( $1, UsedId(stoid(atoe("is_neq")), $2),
         appback(Expr_List(), $3) ); }
| expression '>' expression
  { $$ = Call_Expr( $1, UsedId(stoid(atoe("is_gt")), $2),
         appback(Expr_List(), $3) ); }
| expression GEQ expression
  { $$ = Call_Expr( $1, UsedId(stoid(atoe("is_geq")), $2),
```

```
        appback(Expr_List(), $3) ); }
| '~' expression
  { $$ = Call_Expr( $2, UsedId(stoid(atoe("not")), $1),
        Expr_List()); }
| '-' expression %prec '~'
  { $$ = Call_Expr( $2, UsedId(stoid(atoe("negate")), $1), Expr_List()); }
| expression '[' expression_seq expression ']'
  { $$ = Call_Expr( $1, UsedId(stoid(atoe("aget")), $2), appback($3, $4) ); }
| '[' expression_seq expression ']'
  { $$ = Call_Expr( Self_Expr(), UsedId(stoid(atoe("aget")), $1),
        appback($2, $3) ); }
| '(' expression ')'
  { $$ = $2; }
;

and_expression
: expression AND expression
  { $$ = Logic_Expr($1, And_Op(), $3); }
;

or_expression
: expression OR expression
  { $$ = Logic_Expr($1, Or_Op(), $3); }
;

implication_expression
: expression IMPLICATION expression
  { $$ = Logic_Expr(Call_Expr($1, UsedId(stoid(atoe("not")), $2),
        Expr_List()), Or_Op(), $3); }
;

initial_expression
: INITIAL_SY '(' expression ')'
  { $$ = Initial_Expr($3); }
;

result_expression
: RESULT
  { $$ = Result_Expr(); }
;

while_expression
: WHILE '(' expression ')'
  { $$ = While_Stmt($3); }
;

break_expression
: BREAK
  { $$ = Break_Stmt(); }
;
```

```
bool_literal_expression
: BOOLCONST
  { $$ = $1; }
;

char_literal_expression
: CHARCONST
  { $$ = $1; }
;

str_literal_expression
: STRCONST
  { $$ = $1; }
| str_literal_expression STRCONST
  { $$ = concat($1, $2); }
;

int_literal_expression
: INTCONST
  { $$ = $1; }
;

%%

/***************************************************************************
 * functions required by yacc
 ***************************************************************************/

/*
 * Signature:   int yyerror( char *s )
 * Purpose:     displays an error message
 */
int yyerror( char *s )
{
fprintf( stderr, "**** Line %d: %s before %s\n",yylineno, s, yytext );
}

/*
 * Signature:   Sourcefile yyread( FILE *file )
 * Purpose:     main of parser
 */
ELEMENT yyread( FILE *file )
{
yyin = file;
if( yyparse() )
    return nil();
else
    return syntaxtree;
}
```

```
/*
 * Signature:   int yywrap(void)
 * Purpose:     necessary on HP
 */
int yywrap(void)
{
return(1);
}

/***************************************************************************
 * functions treating string literals
 ***************************************************************************/

/*
 * Signature:   String remove_quotes(String s)
 * Purpose:     removes double quotes of string literal
 */
ELEMENT remove_quotes(ELEMENT s)
{
char *a = etoa(s);
*(a + strlen(a) -1) = '\0';

return atoe(a + 1);
}

/*
 * Signature:   String concat(String s1, String s2)
 * Purpose:     appends s2 to s1 removing the inner quotes
 */
ELEMENT concat(ELEMENT s1, ELEMENT s2)
{
char *cp1 = etoa(s1), *cp2 = etoa(s2);
char *res = (char *)calloc(strlen(cp1) + strlen(cp2) -1, sizeof(char));

if (res)
    {
    strcpy(res, cp1);
    *(res+strlen(res)-1) = '\0';
    strcat (res, cp2+1);
    return atoe(res);
    }

return nil();
}
```

```
/*************************************************************************
 * functions transforming constructs of concrete syntax
 * into constructs supported by abstract syntax
 *************************************************************************/

/*
 * Signature:   Argument_List idlist_to_arglist(DeclId_List, Typespec)
 * Purpose:     resolves parameter declaration of form "x, y: TYPE"
 *              into "x: TYPE, y: TYPE"
 */
ELEMENT idlist_to_arglist(ELEMENT idlist, ELEMENT type)
{
if ( idlist == DeclId_List() )
    return Argument_List();
else
    return (appfront( Typed_Ident(insert_ident(subterm(1, idlist)), type),
            idlist_to_arglist(back(idlist), type) ));
}

/*
 * Signature:   Class_Element_List build_attr_def_list(
 *              DeclId_List idlist, Typespec type , Access_Mode access)
 * Purpose:     resolves attribute declaration of form "x, y: TYPE" into
 *              "x: TYPE, y: TYPE" and appends reader and writer routines
 */
ELEMENT build_attr_def_list(ELEMENT idlist, ELEMENT type, ELEMENT access)
{
ELEMENT res = Class_Element_List();

if ( idlist == DeclId_List() )
    return Class_Element_List();
else
    {
    /* Writer routine */
    if (access == Public())
        res = appfront( Routine_Def(FeatureId(access, subterm(1, idlist)),
        Not_Functional(), appback(Argument_List(), Typed_Ident(
        DeclId(stoid(atoe("x")), itoe(-1)), type)), Same_Type(),
        Condition_List(), Condition_List(), appback(Statement_List(),
        Return_Stmt(Set_Expr(Local_Expr(
        UsedId(stoid(atoe("x")), itoe(-1))))))), res);
    else
        res = appfront( Routine_Def(FeatureId(Private(), subterm(1, idlist)),
        Not_Functional(), appback(Argument_List(),
        Typed_Ident( DeclId(stoid(atoe("x")), itoe(-1)), type)),
        Same_Type(), Condition_List(), Condition_List(), appback(
        Statement_List(), Return_Stmt(Set_Expr(Local_Expr(
        UsedId(stoid(atoe("x")), itoe(-1))))))), res);

    /* Reader routine */
    if (access == Private())
```

```
        res = appfront( Routine_Def(FeatureId(access, subterm(1, idlist)),
        Functional(), Argument_List(), type, Condition_List(),
        Condition_List(), appback(Statement_List(),
        Return_Stmt(Attr_Expr()))), res);
    else
        res = appfront( Routine_Def(FeatureId(Public(), subterm(1, idlist)),
        Functional(), Argument_List(), type, Condition_List(),
        Condition_List(), appback(Statement_List(),
        Return_Stmt(Attr_Expr()))), res);

    res = appfront( Attr_Def(FeatureId(access, subterm(1, idlist)), type),
          res);

    return conc(res, build_attr_def_list(back(idlist), type, access));
    }
}

/*
 * Signature:   Class_Element_List change_writers(Class_Element_List)
 * Purpose:     changes function mode of writer routines in value
 *              classes to Functional()
 */
ELEMENT change_writers(ELEMENT cel)
{
ELEMENT res = Class_Element_List();
int i = 1;
ELEMENT f;

while (i<=numsubterms(cel))
    {
    f = subterm(i, cel);
    res = appback(res, f);
    i++;
    if( is(f, _Attr_Def) )
        {
        res = appback(res, subterm(i, cel));
        f = subterm(i+1, cel);  /* This is the writer */
        res = appback(res, Routine_Def(subterm(1, f), Functional(),
        subterm(3, f), subterm(4, f), subterm(5, f), subterm(6, f),
        subterm(7, f)));
        i += 2;
        }
    }
return res;
}
```

```
/*
 * Signature:   Statement_List decllist_to_stmtlist(DeclId_List, Typespec)
 * Purpose:     resolves local declaration of form "x, y: TYPE;"
 *              into separate statements "x: TYPE; y: TYPE;"
 */
ELEMENT decllist_to_stmtlist(ELEMENT idlist, ELEMENT type)
{
if ( idlist == DeclId_List() )
    return Statement_List();
else
    return (appfront( Typed_Ident(insert_ident(subterm(1, idlist)),
    type), decllist_to_stmtlist(back(idlist), type) ));
}


/*
 * Signature:   If_Stmt remove_elseif(Expr i, Statement_List t,
 *              Variante_List list, Statement_List e)
 * Purpose:     splits if-elsif-else statements into if-else statements
 */
ELEMENT remove_elseif(ELEMENT i, ELEMENT t, ELEMENT list, ELEMENT e)
{
if ( list == Variante_List() )
    return (If_Stmt(i, t, e));
else
    return (If_Stmt(i, t, appback(Statement_List(), remove_elseif(
    subterm(1, subterm(1, list)), subterm(2, subterm(1, list)),
    back(list), e))));
}

/*
 * Signature:   Variante_List build_variante_list(Expr_List, Statement_List)
 * Purpose:     resolves multiple pattern in case statements of
 *              form "when x, y then" into separate
 *              branches "when x then ... when y then ..."
 */
ELEMENT build_variante_list(ELEMENT list, ELEMENT sl)
{
if ( list == Expr_List() )
    return Variante_List();
else
    return (appfront(Variante(subterm(1, list), sl),
    build_variante_list(back(list), sl)));
}
```

```
/*
 * Signature:   Statement_List remove_case(Expr e, Variante_List vlist,
 *              Statement_List sl)
 * Purpose:     transforms case statement into if statement
 */
ELEMENT remove_case(ELEMENT e, ELEMENT vlist, ELEMENT sl)
{
if ( vlist == Variante_List() )
    return sl;
else
    return appback(Statement_List(), If_Stmt(Call_Expr(e,
    UsedId(stoid(atoe("is_eq")), -1), appback(Expr_List(),
    subterm(1, subterm(1, vlist)))), subterm(2, subterm(1, vlist)),
    remove_case(e, back(vlist), sl)));
}

/*
 * Signature:   Expr access_read(UsedId ident, Receiver receiver)
 * Purpose:     handles access to local variable or Attribute
 */
ELEMENT access_read(ELEMENT ident, ELEMENT receiver)
{
if (lookup_ident(ident))                /* locale Variable */
    return Local_Expr(ident);
else                                    /* Attribute */
    return Call_Expr(receiver, ident, Expr_List());
}

/*
 * Signature:   Statement access_write(UsedId, Receiver, Expr)
 * Purpose:     handles access to local variable or Attribute
 */
ELEMENT access_write(ELEMENT ident, ELEMENT receiver, ELEMENT par)
{
if (lookup_ident(ident))                /* locale Variable */
    return Assign_Stmt(Local_Expr(ident), par);
else                                    /* Attribute */
    return Call_Expr(receiver, ident, appback(Expr_List(), par));
}
```

```
/*************************************************************************
 * Implementation of type graph and include graph
 *************************************************************************/

/*
 * Signature:   void add_subtyping_clause_to_type_graph(Ident class,
 *              Typespec_List clause)
 * Purpose:     inserts edges into type graph
 */
void add_subtyping_clause_to_type_graph(ELEMENT class, ELEMENT clause)
{
int i;

for (i=1; i<= numsubterms(clause); i++)
    if (!is(subterm(i, clause), _Same_Type))
        insert_edge(subterm(1, (subterm(i, clause))), class, &type_graph);
}

/*
 * Signature:   void add_supertyping_clause_to_type_graph(Ident class,
 *              Typespec_List clause)
 * Purpose:     inserts edges into type graph
 */
void add_supertyping_clause_to_type_graph(ELEMENT, ELEMENT)
{
int i;

for (i=1; i<= numsubterms(clause); i++)
    insert_edge(class, subterm(1, (subterm(i, clause))), &type_graph);
}

/*
 * Signature:   int lookup_node(Ident type, graph *g)
 * Purpose:     returns id of a class
 */
int lookup_node(ELEMENT type, graph *g)
{
node *ptr = g->first_node;

while(ptr)
    {
    if (ptr->type == type)
        return ptr->id;

    ptr=ptr->next;
    }
return -1;
}
```

```
/*
 * Signature:   void insert_node(Ident type, int counter, graph *g)
 * Purpose:     adds class to graph
 */
void insert_node(ELEMENT type, int counter, graph *g)
{
static int next_number[2] = {0, 0};

node *neu;

if(lookup_node(type, g) != -1)    /* Element already in graph! */
    return ;

neu = (node *)malloc(sizeof(node));

if (!neu)
    {
    fprintf(stderr, "Not enough memory!\n");
    exit(EXIT_FAILURE);
    }
neu->type = type;
neu->next = g->first_node;
neu->id = next_number[counter]++;

g->first_node = neu;
g->nodes++;
}

/*
 * Signature:   void insert_edge(Ident super, Ident sub, graph *g)
 * Purpose:     adds edge to graph
 */
void insert_edge(ELEMENT super, ELEMENT sub, graph *g)
{
edge *neu = (edge *)malloc(sizeof(edge));

if (!neu)
    {
    fprintf(stderr, "Not enough memory!\n");
    exit(EXIT_FAILURE);
    }
neu->super = super;
neu->sub   = sub;
neu->next  = g->first_edge;

g->first_edge = neu;
}
```

```
/*
 * Signature:   void insert_including_class(Ident class)
 * Purpose:     adds the including class to all classes
 *              which have been inserted recently
 */
void insert_including_class(ELEMENT class)
{
edge *ptr = include_graph.first_edge;

while (ptr && ptr->super == NULL)
    {
    ptr->super = class;
    ptr = ptr->next;
    }
}

/*
 * Signature:   void node_set_to_array(graph *g)
 * Purpose:     transforms set of nodes into matrix
 */
void node_set_to_array(graph *g)
{
node *ptr = g->first_node;

g->matrix = (int *)calloc(g->nodes * g->nodes, sizeof(int));
if (g->matrix == NULL)
    {
    fprintf(stderr, "Not enough memory!\n");
    exit(EXIT_FAILURE);
    }

g->trans = (int *)calloc(g->nodes * g->nodes, sizeof(int));
if (g->trans == NULL)
    {
    fprintf(stderr, "Not enough memory!\n");
    exit(EXIT_FAILURE);
    }

g->class_name = (ELEMENT *)calloc(g->nodes, sizeof(ELEMENT));
if (g->class_name == NULL)
    {
    fprintf(stderr, "Not enough memory!\n");
    exit(EXIT_FAILURE);
    }

while(ptr)
    {
    *(g->class_name + ptr->id) = ptr->type;
    ptr=ptr->next;
    }
}
```

```
/*
 * Signature:   void insert_edges(graph *g, char *text)
 * Purpose:     inserts all edges into matrix
 */
void insert_edges(graph *g, char *text)
{
edge *ptr = g->first_edge;
int von, nach;

while (ptr)
    {
    von = lookup_node(ptr->super, g);
    nach= lookup_node(ptr->sub, g);

    if (von == -1)
        {
        fprintf(stderr, "### Undeclared type %s in %s_graph\n",
        etoa(idtos(ptr->super)), text);
        exit(EXIT_FAILURE);
        }
    if (nach == -1)
        {
        fprintf(stderr, "### Undeclared type %s in %s_graph\n",
        etoa(idtos(ptr->sub)), text);
        exit(EXIT_FAILURE);
        }

    *(g->matrix + von*g->nodes + nach) = 1;
    ptr=ptr->next;
    }

memcpy((char *)g->trans, (char *)g->matrix,
g->nodes*g->nodes*sizeof(int));
}

/*
 * Signature:   void transitive_closure(graph *g)
 * Purpose:     computes the transitive closure of g
 */
void transitive_closure(graph *g)
{
int c = g->nodes;
int i, j, k;

for (i=0; i<c; i++)
    for (j=0; j<c; j++)
        if (*(g->trans+j*c+i))
            for (k=0; k<c; k++)
                if(*(g->trans+i*c+k))
                    *(g->trans+j*c+k) = 1;
}
```

```
/*
 * Signature:   char find_cycle(graph *g)
 * Purpose:     detects cycles in graph
 */
char find_cycle(graph *g)
{
int i;
for (i=0; i<g->nodes; i++)
    if (*(g->trans + i*g->nodes + i))
        return 1;

return 0;
}

/*
 * Signature:   void build_graph(graph *g, char *text)
 * Purpose:     processes both graphs
 */
void build_graph(graph *g, char *text)
{
extern char graph_flag;

node_set_to_array(g);
insert_edges(g, text);
transitive_closure(g);

if (find_cycle(g))
    {
    fprintf(stderr, "Cycle in %s_graph detected\n", text);
    exit(EXIT_FAILURE);
    }
}

/****************************************************************************
 * Implementation of temporary symbol table
 ****************************************************************************/

/*
 * Signature:   void open_block(void)
 * Purpose:     opens new scope
 */
void open_block(void)
{
struct bst_Elem *new = (struct bst_Elem *)calloc(1,
                        sizeof(struct bst_Elem));

if (!new)
    {
    fprintf(stderr, "Not enough memory!\n");
    exit(EXIT_FAILURE);
    }
```

```
new->next = symtab;
symtab = new;
}

/*
 * Signature:   void close_block(void)
 * Purpose:     closes scope and removes entries
 */
void close_block(void)
{
struct bst_Elem *h = symtab;
int i;
struct List_Elem *ptr, *help;

for (i=0; i<ARRAY_SIZE; i++)
    {
    ptr = h->hash_table[i];
    while (ptr)
        {
        help = ptr->next;
        free (ptr->id);
        free (ptr);
        ptr = help;
        }
    }

symtab = h->next;
free(h);
}

/*
 * Signature:   char hash(char *ident)
 * Purpose:     returns primitive hash value
 */
char hash(char *ident)
{
int s=0;

for(s=0; *ident != '\0'; ident++)
    s += *ident;

return s % ARRAY_SIZE;
}
```

```
/*
 * Signature:   Ident insert_ident(Ident ident)
 * Purpose:     adds ident to symbol table
 */
ELEMENT insert_ident(ELEMENT ident)
{
char *id = etoa(idtos(subterm(1, ident)));
int i = hash(id);
struct List_Elem *neu;


if (lookup_ident(ident))        /* Allready declared in this scope */
    {
    fprintf(stderr, "### LINE %d: Ident %s declared twice\n",
    etoi(subterm(2, ident)), id);
    exit (EXIT_FAILURE);
    }

neu = (struct List_Elem *)malloc(sizeof(struct List_Elem));
if (!neu)
    {
    fprintf(stderr, "Not enough memory!\n");
    exit(EXIT_FAILURE);
    }

neu->id = (char *)calloc(strlen(id) + 1, sizeof(char));
if (neu->id == NULL)
    {
    fprintf(stderr, "Not enough memory!\n");
    exit(EXIT_FAILURE);
    }

strcpy(neu->id, id);
neu->next = symtab->hash_table[i];
symtab->hash_table[i] = neu;

return ident;
}

/*
 * Signature:   char lookup_ident(Ident id)
 * Purpose:     returns if id is in symbol table
 */
char lookup_ident(ELEMENT id)
{
char *ident = etoa(idtos(subterm(1, id)));
int i=hash(ident);
struct bst_Elem *bptr = symtab;
struct List_Elem *ptr;
```

```
while (bptr)
    {
    ptr = bptr->hash_table[i];
    while (ptr)
        if (strcmp(ptr->id, ident) == 0)
            return 1;
        else
            ptr = ptr->next;
    bptr = bptr->next;
    }
return 0;
}
```

B.2.3 Die MAX-Spezifikation

```
//*************************************************************************
//  Module:              SALSA_spec.m                                     *
//  Author:              Peter Müller                                     *
//  Description:         MAX specification of Sather/SALSA interpreter    *
//*************************************************************************

//*************************************************************************
// Abstract Syntax of Sather-subset
//*************************************************************************

// Classes
Program             ( Sourcefile Statement_List )
Sourcefile          * Class
Class               = AbstractClass | RefClass | ValueClass
AbstractClass       ( DeclId Typespec_List Typespec_List Condition_List
      Signature_List)
RefClass            ( DeclId Typespec_List Condition_List Class_Element_List)
ValueClass          ( DeclId Typespec_List Condition_List Class_Element_List)

// Idents
DeclId              ( Ident Int )
UsedId              ( Ident Int )
FeatureId           ( Access_Mode DeclId )

Access_Mode         = Private | Public | Readonly
Private             ()
Public              ()
Readonly            ()

// Types
Typespec_List       * Typespec
Typespec            = UsedId | Same_Type
Same_Type           ()

// Class elements
Class_Element_List  * Class_Element
Class_Element       = Attr_Def | Routine_Def | Include_Clause

Signature_List      * Signature
Signature           ( DeclId Function_Mode Argument_List Typespec
      Condition_List Condition_List )

Argument_List       * Typed_Ident
Typed_Ident         ( DeclId Typespec )

Attr_Def            ( FeatureId Typespec )
Routine_Def         ( FeatureId Function_Mode Argument_List Typespec
      Condition_List Condition_List Statement_List )
```

```
Function_Mode       = Functional | Not_Functional
Functional          ()
Not_Functional      ()

// Include clauses are removed before the syntax tree is built
// They are only used by the parser
Include_Clause      ( Access_Mode Typespec Feature_Modifier_List )
Feature_Modifier_List   * Feature_Modifier
Feature_Modifier    = Mod_Rename | UsedId
Mod_Rename          ( UsedId FeatureId )

// Conditions
Condition_List      * Condition
Condition           ( DeclId Expr )

// Statements
Statement_List      * Statement
Statement           = Typed_Ident | Assign_Stmt | If_Stmt | Call_Expr
    | Typecase_Stmt | Loop | Return_Stmt | While_Stmt
    | Break_Stmt | Assert_Stmt

Assign_Stmt         ( Local_Expr Expr )
If_Stmt             ( Expr Statement_List Statement_List )
Typecase_Stmt       ( Local_Expr Alternative_List Statement_List )
Alternative_List    * Alternative
Alternative         ( Typespec Statement_List )
Loop                ( Statement_List )
Return_Stmt         ( Expr )
While_Stmt          ( Expr )
Break_Stmt          ()
Assert_Stmt         ( Condition_List )

// Expressions
Expr                = Self_Expr | Local_Expr | Call_Expr | Void_Expr
    | New_Expr | Logic_Expr | Void_Test_Expr | Literal_Expr
    | Attr_Expr | Set_Expr | Initial_Expr | Result_Expr

Self_Expr           ()
Local_Expr          ( UsedId )
Call_Expr           ( Receiver UsedId Expr_List )
Receiver            = Expr | Typespec
Expr_List           * Expr
Void_Expr           ()
New_Expr            ()
Logic_Expr          ( Expr Operator Expr )
Operator            = And_Op | Or_Op
And_Op              ()
Or_Op               ()
Void_Test_Expr      ( Expr )
Literal_Expr        = Bool | Char | String | Int
Attr_Expr           ()
```

```
Set_Expr            ( Local_Expr )
Initial_Expr        ( Expr )
Result_Expr         ()

//***************************************************************************
// Data types for the parser
//***************************************************************************

DeclId_List     * DeclId
Variante        ( Expr Statement_List )
Variante_List   * Variante

//***************************************************************************
// Data types for static analysis
//***************************************************************************

Rout_or_Sig     = Routine_Def@ | Signature@
CElement        = Routine_Def@ | Attr_Def@ | Signature@
Interface       = Class_Element_List@ | Signature_List@

Declaration     = Class@ | CElement | Statement@
Type            = Class@ | Void_Type
Void_Type       ()

//***************************************************************************
// Data Types for dynamic functions
//***************************************************************************

// Data types of Sather
Value           = Bool | Int | Char | String | Object | Void_Value
Object          = RefObject | ValObject
RefObject       ( RefClass@ ObjectId )
ObjectId        = Int
ValObject       ( ValueClass@ Value_List )
Value_List      * Value
Void_Value      ()

// Locations
Var             = StaticVar | DynamicVar
StaticVar       = Expr@ | Typed_Ident@ | Attr_Def@ | Self_Expr | Routine_Def@

DynamicVar      = TempVar | ActVar | InstVar | SelfVar
TempVar         ( StaticVar IncarId )
ActVar          ( Typed_Ident@ IncarId )
InstVar         ( Attr_Def@ ObjectId )
SelfVar         ( IncarId )
IncarId         = Int
```

```
//*************************************************************************
// Tasks
//*************************************************************************

DynamicTask     ( Task IncarId )
Task            = NonEndTask | End
NonEndTask      = Seq_Task | CreateObj | MoveArg | Call | Return
| Branch | Typecase | Logic | Invariant | Result | Error
Seq_Task        = Point | MoveVal | Select | Void_Test | Set | Move
End             = Terminated | Failed

MoveVal         ( Var Value Task )
Select          ( Call_Expr@ Task )
Result          ( Expr@ )
Void_Test       ( Expr@ Void_Test_Expr@ Task )
Set             ( Set_Expr@ Task )
Move            ( Var Var Task )

CreateObj       ( Var Class@ Task )
MoveArg         ( Call_Expr@ Task )
Call            ( Call_Expr@ )
Return          ()
Branch          ( Expr@ Task Task )
Typecase        ( Local_Expr@ Typespec@ Task Task )
Logic           ( Expr@ Logic_Expr@ Task Task )
Invariant       ()
Error           ( String )

Terminated      ()
Failed          ()

//*************************************************************************
// Attributes and functions for static analysis
//*************************************************************************

STRUC s [Program]{

//*************************************************************************
// Functions defined in C
//*************************************************************************

// Output functions
FCT out( String )               Int
FCT linetos( Int )              String
FCT print_result( Value )       Int

// Type graph function
FCT is_subtype(Class@, Class@)  Bool
```

```
// Definitions of predefined classes
// Parameter Int is a dummy preventing the compiler from reporting an error
FCT bool_class1(Int)            Class@
FCT char_class1(Int)            Class@
FCT string_class1(Int)          Class@
FCT int_class1(Int)             Class@
FCT eval_predecl(String, Call_Expr@) Value

// Functions supporting MAX
FCT next (Point)                Point
FCT plus(Int, Int)              Int
FCT minus(Int, Int)             Int

// Check level of assertions
FCT check_level(Int)            Bool

//*****************************************************************************
// Selectors for nodes
//*****************************************************************************

FCT fst( Node N ) Node:                                 son(1, N)
FCT snd( Node N ) Node:                                 son(2, N)

FCT ident(Node N) Ident@:
    IF is[N, _DeclId@]:                                 son(1, N)
     | is[N, _UsedId@]:                                 son(1, N)
     | is[N, _FeatureId@]:                              son(2, N).ident
     | is[N, _Signature@]:                              son(1, N).ident
     | is[N, _Typed_Ident@]:                            son(1, N).ident
     | is[N, _Condition@]:                              son(1, N).ident
     | is[N, _Attr_Def@]:                               son(1, N).ident
     | is[N, _Routine_Def@]:                            son(1, N).ident
     | is[N, _Class@]:                                  son(1, N).ident
     | is[N, _Call_Expr@]:                              son(2, N).ident
     | is[N, _Local_Expr@]:                             son(1, N).ident
    ELSE nil()
FCT declid(Node N) DeclId@:
    IF is[N, _FeatureId@]:                              son(2, N)
    ELSE                                                son(1, N)
FCT usedid(Node N) UsedId@:                             son(1, N)
FCT featureid(Node N) FeatureId@:                       son(1, N)
FCT access_mode(FeatureId@ F) Access_Mode@:             son(1, F)
FCT line(Node N) Int@:                                  son(2, N)
FCT sourcefile(Program@ P) Sourcefile@:                 son(1, P)
FCT subtyping(Class@ C) Typespec_List@:                 son(2, C)
FCT supertyping(AbstractClass@ A) Typespec_List@:       son(3, A)
FCT invariant(Class@ C) Condition_List@:
    IF is[C, _AbstractClass@]:                          son(4, C)
    ELSE                                                son(3, C)
FCT signature_list(AbstractClass@ A) Signature_List@:   son(5, A)
FCT class_element_list(Class@ C) Class_Element_List@:   son(4, C)
```

```
FCT argument_list(Rout_or_Sig N) Argument_List@:
    IF is[N, _Routine_Def@]:                          son(3, N)
    ELSE                                              son(3, N)
FCT statement_list(Node N) Statement_List@:
    IF is[N, _Program@]:                              son(2, N)
     | is[N, _Routine_Def@]:                          son(7, N)
     | is[N, _Typecase_Stmt@]:                        son(3, N)
     | is[N, _Alternative@]:                          son(2, N)
     | is[N, _Loop@]:                                 son(1, N)
    ELSE                                              nil()
FCT typespec( Node N ) Typespec@:
    IF is[N, _Alternative@]:                          son(1, N)
     | is[N, _Routine_Def@]:                          son(4, N)
     | is[N, _Signature@]:                            son(4, N)
     | is[N, _Typed_Ident@]:                          son(2, N)
     | is[N, _Attr_Def@]:                             son(2, N)
    ELSE                                              nil()
FCT receiver( Call_Expr@ C ) Node:                    son(1, C)
FCT expr_list( Call_Expr@ C ) Expr_List@:             son(3, C)
FCT pre(Rout_or_Sig N) Condition_List@:
    IF is[N, _Signature@]:                            son(5, N)
    ELSE                                              son(5, N)
FCT post(Rout_or_Sig N) Condition_List@:
    IF is[N, _Signature@]:                            son(6, N)
    ELSE                                              son(6, N)
FCT function_mode(Rout_or_Sig R) Function_Mode@:      son(2, R)
FCT then_list( If_Stmt@ I ) Statement_List@:          son(2, I)
FCT else_list( Node I ) Statement_List@:              son(3, I)
FCT condition_list(Assert_Stmt@ A) Condition_List@:   son(1, A)
FCT local_expr( Node T ) Node:                        son(1, T)
FCT expr(Node N) Expr@:
    IF is[N, _Initial_Expr@]:                         son(1, N)
     | is[N, _Void_Test_Expr@]:                       son(1, N)
    ELSE                                              son(2, N)
FCT operator( Logic_Expr@ L ) Operator@:              son(2, L)
FCT alternative_list(Typecase_Stmt@ T) Alternative_List@:   son(2, T)

//*************************************************************************
// Selectors for terms
//*************************************************************************

FCT t_fst( Term N ) Term:                             subterm(1, N)
FCT t_snd( Term N ) Term:                             subterm(2, N)

FCT t_ident(Term N) Ident:
    IF is[N, _DeclId]:                                subterm(1, N)
     | is[N, _UsedId]:                                subterm(1, N)
     | is[N, _FeatureId]:                             subterm(2, N).t_ident
     | is[N, _Signature]:                             subterm(1, N).t_ident
     | is[N, _Typed_Ident]:                           subterm(1, N).t_ident
     | is[N, _Condition]:                             subterm(1, N).t_ident
```

```
     | is[N, _Attr_Def]:                                subterm(1, N).t_ident
     | is[N, _Routine_Def]:                            subterm(1, N).t_ident
     | is[N, _Class]:                                  subterm(1, N).t_ident
     | is[N, _Call_Expr]:                              subterm(2, N).t_ident
     | is[N, _Local_Expr]:                             subterm(1, N).t_ident
     | is[N, _Mod_Rename]:                             subterm(1, N).t_ident
    ELSE                                               nil()
FCT t_declid(Term N) DeclId:
    IF is[N, _FeatureId]:                              subterm(2, N)
    ELSE                                               subterm(1, N)
FCT t_featureid(Term N) FeatureId:
    IF is[N, _Mod_Rename]:                             subterm(2, N)
    ELSE                                               subterm(1, N)
FCT t_access_mode(Term N) Access_Mode:                 subterm(1, N)
FCT t_line(Term N) Int:                                subterm(2, N)
FCT t_typespec(Term N) Typespec:
    IF is[N, _Alternative]:                            subterm(1, N)
     | is[N, _Routine_Def]:                            subterm(4, N)
     | is[N, _Signature]:                              subterm(4, N)
     | is[N, _Typed_Ident]:                            subterm(2, N)
     | is[N, _Attr_Def]:                               subterm(2, N)
     | is[N, _Include_Clause]:                         subterm(2, N)
    ELSE                                               nil()
FCT t_class_element_list(Class C) Class_Element_List:  subterm(4, C)
FCT t_signature_list(AbstractClass A) Signature_List:  subterm(5, A)
FCT t_argument_list(Term N) Argument_List:
    IF is[N, _Routine_Def]:                            subterm(3, N)
    ELSE                                               subterm(3, N)
FCT t_feature_modifier_list(Include_Clause I) Feature_Modifier_List:
subterm(3, I)
//*************************************************************************
// Selectors for other data types
//*************************************************************************

FCT objclass( Object O ) Class@:                       subterm(1, O)
FCT objid( RefObject O ) ObjectId:                     subterm(2, O)
FCT t_value_list(ValObject O) Value_List:              subterm(2, O)
FCT dyn_task(DynamicTask D) Task:                      subterm(1, D)
FCT incid(DynamicTask D) IncarId:                      subterm(2, D)
FCT typed_ident(ActVar A) Typed_Ident@:                subterm(1, A)

//*************************************************************************
// Selectors for tasks
//*************************************************************************

FCT t_dest(Task T) Var:                                subterm(1, T)
FCT t_src(Task T) Var:                                 subterm(2, T)
FCT t_var(Task T) Var:                                 subterm(1, T)
FCT t_class(Task T) Class@:                            subterm(2, T)
FCT t_value(Task T) Value:                             subterm(2, T)
FCT t_call_expr(Task T) Call_Expr@:                    subterm(1, T)
```

```
FCT t_expr(Task T) Expr@:                                subterm(1, T)
FCT t_then_task(Task T) Task:
    IF is[T, _Branch]:                                   subterm(2, T)
    ELSE                                                 subterm(3, T)
FCT t_else_task(Task T) Task:
    IF is[T, _Branch]:                                   subterm(3, T)
    ELSE                                                 subterm(4, T)
FCT t_type(Typecase T) Typespec@:                        subterm(2, T)
FCT t_logic_expr(Task T) Logic_Expr@:                    subterm(2, T)
FCT t_void_test_expr(Task T) Void_Expr@:                 subterm(2, T)

//*************************************************************************
// environments
//*************************************************************************

ATT encl_class(Node N) Class@:
    IF is[N, _Class@]:                   N
    ELSE N.fath.encl_class

ATT encl_rout( Node N ) Rout_or_Sig:
    IF is[N, _Routine_Def@]:             N
     | is[N, _Signature@]:               N
    ELSE N.fath.encl_rout

ATT encl_stmt ( Node N ) Statement@:
    IF is[N.fath, _Statement_List@]:     N
    ELSE N.fath.encl_stmt

ATT encl_loop( Node N ) Loop@:
    IF is[N, _Loop@]:                    N
    ELSE N.fath.encl_loop

ATT encl_typecase( Node N ) Typecase_Stmt@:
    IF is[N, _Typecase_Stmt@]:           N
    ELSE N.fath.encl_typecase

ATT encl_condition( Node N ) Condition@:
    IF is[N, _Condition@]:               N
    ELSE N.fath.encl_condition

ATT encl_initial( Node N ) Initial_Expr@:
    IF is[N, _Initial_Expr@]:            N
    ELSE N.fath.encl_initial
```

```
//*************************************************************************
// Location of errors
//*************************************************************************

FCT loc( Node N ) String:
    LET R == N.encl_rout:
    IF R # nil():
`"ROUTINE " N.encl_class.ident.term.idtos "::" R.ident.term.idtos'
    ELSE
`"CLASS " N.encl_class.ident.term.idtos'

//*************************************************************************
// Context conditions
//*************************************************************************

// All classes must have different names
CND Sourcefile@<*, <<ID1, No1>, *>, *, <<ID2, No2>, *>, *>   :
       ID1.term # ID2.term
| `"### Class or type " ID1.term.idtos " declared twice in: "
  No1.term.linetos " and " No2.term.linetos "\n"'

// Include clauses should be removed by the parser
CND Include_Clause@ I:
    false() = true()
| `"### " I.loc ": Unresolved include clause detected\n"'

// SAME must not appear in abstract types
CND Same_Type@ S:
    !is[S.encl_class, _AbstractClass@]
| `"### " S.loc ": SAME must not appear in abstract types\n"'

// Each parameter must have a different name
CND Argument_List@<*, <<ID1, No1>, T1>, *, <<ID2, No2>, T2>, *> A:
    ID1.term # ID2.term
| `"### " No1.term.linetos ": Parameter " ID1.term.idtos " declared twice
  in " A.encl_class.ident.term.idtos "::" A.encl_rout.ident.term.idtos "\n"'

// New may appear in refercence classes only
CND New_Expr@ N:
    is[N.encl_class, _RefClass@]
| `"### " N.loc ": Illegal new expression in value class\n"'

// Void expressions must not appear as receiver or in Logic_Expr (p. 21)
CND Void_Expr@ V:
    !is[V.fath, _Logic_Expr@] && !is[V.fath, _Call_Expr@] &&
    !is[V.fath, _If_Stmt@]
| `"### " V.loc ": Illegal use of void\n"'
```

```
// while! may only be called from inside a loop
CND While_Stmt@<E> S:
    S.encl_loop # nil()
| `"### " S.loc ": while! may only be called from within loops\n"'

// break! may only be called from inside a loop
CND Break_Stmt@ S:
    S.encl_loop # nil()
| `"### " S.loc ": break! may only be called from within loops\n"'

// If no type is given in a typecase statement, there has to be an else clause
CND Typecase_Stmt@<_, <>, ES>:
    ES.fst # nil()
| `"### " ES.loc ": Missing else clause in case of empty typecase list\n"'

// result may only appear in postconditions
CND Result_Expr@ R:
    R.encl_condition.fath = R.encl_rout.post
| `"### " R.loc ": Result may only appear in postconditions\n"'

// result may not appear in initial expressions
CND Result_Expr@ R:
    R.encl_initial = nil()
| `"### " R.loc ": Result may not appear in initial expressions\n"'

// initial may only appear conditions
CND Initial_Expr@ R:
    R.encl_condition # nil() &&
    {R.encl_condition.fath = R.encl_rout.post ||
    is[R.encl_condition.fath.fath, _Assert_Stmt@]}
| `"### " R.loc ": Initial may only appear in postconditions and
  assert statements\n"'

// initial expression may not be nested
CND Initial_Expr@ I:
    I.fath.encl_initial = nil()
| `"### " I.loc ": Nested initial expression detected\n"'

// New may not appear in conditions
CND New_Expr@ N:
    N.encl_condition = nil()
| `"### " N.loc ": Illegal new in annotation detected\n"'

// Assertions must have different idents
CND Condition_List@<*, <<ID1, No1>, *>, *, <<ID2, No2>, *>, *>   :
       ID1.term # ID2.term
| `"### " No1.term.linetos " and " No2.term.linetos ": Ident "
  ID1.term.idtos " used twice condition list\n"'
```

```
//*************************************************************************
// Definitions of predefined classes
//*************************************************************************

ATT bool_class(Program@ N) Class@:      bool_class1(1)
ATT char_class(Program@ N) Class@:      char_class1(1)
ATT int_class(Program@ N) Class@:       int_class1(1)
ATT string_class(Program@ N) Class@:    string_class1(1)

//*************************************************************************
// functions for searching the declaration of an ident
//*************************************************************************

// look for any ident
FCT lookup( Ident ID, Declaration D ) Declaration:
    IF D.ident.term = ID:   D
    ELSE                    lookup(ID, D.rbroth)

// look for local variables
FCT lookup_local ( Ident@ ID, Statement@ S) Typed_Ident@:
    IF is[S, _Typed_Ident@] && ID.term = S.ident.term: S
    ELSE
IF is[S.lbroth, _Statement@]:
    lookup_local(ID, S.lbroth)
ELSE
    IF S.fath.encl_stmt # nil() :
lookup_local(ID, S.fath.encl_stmt)
    ELSE    // look at the parameter list
lookup(ID.term, S.encl_rout.argument_list.fst)

//*************************************************************************
// function for selecting the matching routine
//*************************************************************************

// Check parameter types
FCT parameter_match(Expr@ E, Typed_Ident@ TI) Bool :
    IF is[E.type, _Void_Type] || is_subtype(E.type, TI.class) = true():
IF E.rbroth # nil():    parameter_match(E.rbroth, TI.rbroth)
ELSE                    true()
    ELSE false()            // Also in case E.type = nil()

// Check if routine matches call
FCT
routine_match(Ident@ ID, Expr_List@ EL, DeclId@ DID, Argument_List@ AL) Bool:
    IF ID.term = DID.ident.term
    && EL.numsons = AL.numsons:
IF EL.fst # nil():  parameter_match(EL.fst, AL.fst)
ELSE                true()
    ELSE false()
```

```
// look for unique routine or signature
FCT lookup_rout (Ident@ ID, Expr_List@ EL, CElement C) Rout_or_Sig:
    LET H ==
    IF is[C, _Signature@]   &&
       routine_match(ID, EL, C.declid, C.argument_list) = true():   C
     | is[C, _Routine_Def@] &&
       routine_match(ID, EL, C.featureid.declid, C.argument_list) = true(): C
    ELSE nil():
    LET N == lookup_rout (ID, EL, C.rbroth):
    IF H = nil():   N
    ELSE
IF N = nil():   H
ELSE            nil()   // two matching sigs

// select routine or signature for a call
FCT select_rout (Ident@ ID, Expr_List@ EL, Class@ C) Rout_or_Sig:
    IF is[C, _AbstractClass@]:  lookup_rout (ID, EL, C.signature_list.fst)
    ELSE                        lookup_rout (ID, EL, C.class_element_list.fst)

//*************************************************************************
// checking wether an assignment is legal
// typecase variables must not be assigned to
//*************************************************************************

FCT check_typecase( Node N, Ident ID ) Bool:
    LET E == N.encl_typecase:
    IF E = nil():   true()
    ELSE
IF E.local_expr.ident.term = ID:     false()
ELSE check_typecase(E.fath, ID)

//*************************************************************************
// functions for checking wether a local expression appears within a typecase
//*************************************************************************

FCT get_alternative(Local_Expr@ E, Alternative@ A) Alternative@:
    IF desc[E, A]:  A
    ELSE get_alternative(E, A.rbroth)

FCT local_typechange(Local_Expr@ E, Node N) Typespec@:
    LET T == N.encl_typecase:
    IF T # nil()  && check_typecase(T, E.ident.term) = false():
IF E = T.local_expr || desc[E, T.statement_list]:
    // This typecase doesn't play any role
    local_typechange(E, T.fath)
ELSE
    LET TYPE == get_alternative(E, T.alternative_list.fst).typespec:
    LET L == lookup_local(E.ident, E.encl_stmt).typespec:
    IF is_subtype(L.typeclass , TYPE.typeclass) = true():   L
    ELSE TYPE
    ELSE E.usedid.decl.typespec
```

```
//*************************************************************************
// function for evaluating the type of each expression
//*************************************************************************

ATT type( Expr@ E ) Type:
    IF Self_Expr@ E:
E.encl_class
     | Local_Expr@ E:
local_typechange(E, E).typeclass
     | Call_Expr@ E:
E.target.class
     | Void_Expr@ E:
Void_Type()
     | New_Expr@ E:
E.encl_class
     | Logic_Expr@ E:
root().bool_class
     | Void_Test_Expr@ E:
root().bool_class
     | Literal_Expr@ E:
IF Bool@ E:
    root().bool_class
 | Char@ E:
    root().char_class
 | String@ E:
    root().string_class
 | Int@ E:
    root().int_class
ELSE nil()
     | Attr_Expr@ E:
E.encl_rout.class
     | Set_Expr@ E:
E.encl_class
     | Result_Expr@ E:
E.encl_rout.class
     | Initial_Expr@<EXP> E:
EXP.type
    ELSE nil()

//*************************************************************************
// binding each call to a routine or signature
//*************************************************************************

ATT target( Call_Expr@ CE ) Rout_or_Sig:
    LET T == IF is[CE.receiver, _Expr@]:    CE.receiver.type
 ELSE                            CE.receiver.typeclass:
    select_rout(CE.ident, CE.expr_list, T)
```

```
//*************************************************************************
// finding the declaration of each UsedId
//*************************************************************************

ATT decl( UsedId@ UID ) Declaration:
    IF Local_Expr@<<ID, _> UID>:
IF UID.encl_stmt # nil():   lookup_local(ID, UID.encl_stmt)
ELSE                // look at the parameter list
    lookup(ID.term, UID.encl_rout.argument_list.fst)
     | Call_Expr@<_, UID, _> CE:    CE.target
     | Typespec@<ID, _> UID:        lookup(ID.term, root().sourcefile.fst)
    ELSE nil()      // In case of unresolved include clauses. Fatal error!

//*************************************************************************
// finding the class to each Typespec
//*************************************************************************

ATT typeclass( Typespec@ T ) Class@:
    IF is[T, _Same_Type@]:  T.encl_class
    ELSE                    T.decl

FCT class( Node N ) Class@: N.typespec.typeclass

//*************************************************************************
// Context conditions
//*************************************************************************

//Only abstract types may be supertypes
CND Class@<<ID, No>, <*, SUB, *>, *>  :
    is[SUB.typeclass, _AbstractClass@]
| `"### " No.term.linetos ": Supertype " SUB.typeclass.ident.term.idtos
  " of " ID.term.idtos " is no abstract type\n"'

//Each UsedId must be declared
CND UsedId@<ID, No> UID:
    UID.decl # nil()
| `"### " No.term.linetos ": Identifier " ID.term.idtos " not declared\n"'

//Each Expression has a decidable type
CND Expr@ E:       // No Error for internal call of MAIN::main
    E.type # nil() || E.encl_class = nil()
| `"### " E.loc ": Type of expression undecidable\n"'

//Only boolean values may be used with 'and' and 'or'
CND Logic_Expr@<E1, _, E2> L:
    E1.type = root().bool_class && E2.type = root().bool_class
| `"### " L.loc ": And and Or only defined for arguments of type BOOL\n"'
```

```
//Direct calls on abstract types are not allowed
CND Call_Expr@<R, <ID, No>, _> CE:
    is[R, _Expr@] || !is[R.typeclass, _AbstractClass@]
| `"### " No.term.linetos ": Direct call on abstract type\n"'

//Variables may only hold subtypes
CND Assign_Stmt@<<<ID, No>> E1, E2>:
    E2.type = Void_Type()  ||
    is_subtype(E2.type, E1.type) = true()
| `"### " No.term.linetos ": Incompatible types in assign statement\n"'

//Conditions in if statements must be boolean
CND If_Stmt@<E, _, _>:
    E.type = root().bool_class
| `"### " E.loc ": condition in if statement not of type BOOL\n"'

//Type of returned expression must conform with declaration of Routine
CND Return_Stmt@<E> R:
    E.type = Void_Type()   ||
    is_subtype(E.type, R.encl_rout.class) = true()
| `"### " R.encl_rout.featureid.declid.line.term.linetos ": Returned type of "
  E.encl_class.ident.term.idtos "::" R.encl_rout.ident.term.idtos
  " incompatible with declared type\n"'

//Conditions in while statements must be boolean
CND While_Stmt@<E> S:
    E.type = root().bool_class
| `"### " E.loc ": parameter of while! not of type BOOL\n"'

//typecase variable must not be assigned to
CND Assign_Stmt@<<<ID, No>>, _> S:
    check_typecase(S, ID.term) = true()
| `"### " No.term.linetos ": Illegal assignment to the typecase variable\n"'

//New is not allowed in functional routines
CND New_Expr@ N:
    is[N.encl_rout.function_mode, _Not_Functional@]
| `"### " N.loc ": Illegal new expression in functional routine\n"'

//Functionals may only call functionals
CND Call_Expr@ <_, <ID, No>, _> CE:
    CE.encl_rout = nil() || is[CE.encl_rout.function_mode, _Not_Functional@]
    ||  is[CE.target.function_mode, _Functional@]
| `"### " No.term.linetos ": Illegal call of nonfunctional routine "
  ID.term.idtos " in functional routine\n"'

//Only functionals may be called in annotations
CND Call_Expr@<_, <ID, No>, _> C:
    C.encl_condition = nil() || is[C.target.function_mode, _Functional@]
| `"### " No.term.linetos ": Nonfunctional routine " ID.term.idtos
  " called in annotation\n"'
```

```
//*************************************************************************
// Check for encapsulation
//*************************************************************************

ATT check_for_privacy(Call_Expr@ CE) Bool:
    LET D == CE.target:
    IF is[D, _Signature@]
    || is[D.featureid.access_mode, _Public@]
    || CE.encl_class = CE.target.encl_class:
true()
    ELSE
false()

//Each call must be checked for privacy
CND Call_Expr@<_, <ID, No>, _> CE:
    CE.check_for_privacy = true()
| `"### " No.term.linetos ": Call of private routine " ID.term.idtos "\n"'

//*************************************************************************
// checks if two conditions are equal
//*************************************************************************

ATT par_id(Typed_Ident@ T) Int:
    IF is[T.fath, _Argument_List@]:
IF T.lbroth = nil():    1
ELSE                    plus(T.lbroth.par_id, 1)
    ELSE    -1

FCT expr_list_equal(Expr@ E1, Expr@ E2) Bool:
    IF cond_equal(E1, E2) = true():
IF E1.rbroth # nil():   expr_list_equal(E1.rbroth, E2.rbroth)
ELSE true()
    ELSE false()

FCT eq_int(Int I1, Int I2) Bool:
    IF I1 = I2:     true()
    ELSE            false()

FCT cond_equal(Expr@ E1, Expr@ E2) Bool:
    IF E1.sort # E2.sort:   false()
    ELSE
IF Local_Expr@ <UID> E1:
    eq_int(UID.decl.par_id, E2.usedid.decl.par_id)
 | Void_Test_Expr@ <E> E1:      cond_equal(E, E2.expr)
 | Initial_Expr@ <E> E1:        cond_equal(E, E2.expr)
 | Call_Expr@ <R1, UID, EL> E1:
    LET R2 == E2.receiver:
    LET HR ==
    IF is[R1, _Expr@]:
IF is[R2, _Expr@]:      cond_equal(R1, R2)
ELSE                    false()
```

```
    ELSE
IF is[R2, _Expr@]:      false()
ELSE
    IF R1.ident.term = R2.ident.term:       true()
    ELSE false():
    IF HR = true() && UID.ident.term = E2.ident.term:
LET EL2 == E2.expr_list:
IF numsons(EL) = numsons(EL2):
    IF numsons(EL) = 0: true()
    ELSE expr_list_equal(EL.fst, EL2.fst)
ELSE false()
    ELSE false()
 | Logic_Expr@ <O1, OP, O2> E1:
    IF OP.sort = E2.operator.sort
    && cond_equal(O1, E2.fst) = true()
    && cond_equal(O2, son(3, E2)) = true(): true()
    ELSE false()
 | Literal_Expr@ E1:
    IF E1.term = E2.term: true()
    ELSE false()
ELSE true()

//*************************************************************************
// check whether E1 is weaker or equal to E2
//*************************************************************************

FCT is_weaker(Expr@ E1, Expr@ E2) Bool:
    IF cond_equal(E1, E2) = true(): true()
    ELSE
LET H1 ==
IF Logic_Expr@ <O1, OP, O2> E1:
    IF is[OP, _Or_Op@]
    && {is_weaker(O1, E2)  = true()
    ||  is_weaker(O2, E2)  = true()}:       true()
    ELSE false()
ELSE false():
IF H1 = true(): true()
ELSE
    IF Logic_Expr@ <O1, OP, O2> E2:
IF is[OP, _And_Op@]
&& {is_weaker(E1, O1)  = true()
||  is_weaker(E1, O2)  = true()}:        true()
ELSE false()
    ELSE false()
```

```
//*************************************************************************
// check whether condition list surrounding C1 is weaker
// or equal condition list surrounding C2
//*************************************************************************

FCT cond_conform(Condition@ C1, Condition@ C2) Bool:
    LET R1 == C1.rbroth:
    IF is_weaker(C1.expr, C2.expr) = true():
IF R1 # nil():  cond_conform(R1, C2.fath.fst)
ELSE            true()
    ELSE
LET R2 == C2.rbroth:
IF R2 # nil():  cond_conform(C1, R2)
ELSE            false()

FCT cond_list_conform(Condition_List@ CL1, Condition_List@ CL2) Bool:
    IF CL1.numsons # 0:
IF CL2.numsons = 0:     false()
ELSE cond_conform(CL1.fst, CL2.fst)
    ELSE true()

//*************************************************************************
// functions for testing if two routine signatures conflict according to p. 4
//*************************************************************************

FCT arg_conflict(Typed_Ident@ FA, Typed_Ident@ GA) Bool:
    LET FTYPE == FA.class:
    LET GTYPE == GA.class:
    IF FTYPE = GTYPE
    || is[FTYPE, _AbstractClass@]
    || is[GTYPE, _AbstractClass@]:
IF FA.rbroth # nil():  arg_conflict(FA.rbroth, GA.rbroth)
ELSE                   true()
    ELSE    false()

FCT rout_conflict(Rout_or_Sig F, Rout_or_Sig G) Bool:
    LET FARG  == F.argument_list:
    LET GARG  == G.argument_list:
    IF F.ident.term = G.ident.term
    && FARG.numsons = GARG.numsons:
IF FARG.numsons # 0:    arg_conflict(FARG.fst, GARG.fst)
ELSE                    true()
    ELSE    false()
```

```
//*************************************************************************
// functions for testing if two routine signatures conform according to p. 4
//*************************************************************************

FCT arg_conform(Typed_Ident@ FA, Typed_Ident@ GA) Bool:
    LET FTYPE == FA.class:
    LET GTYPE == GA.class:
    IF is_subtype(GTYPE, FTYPE) = true():
IF FA.rbroth # nil():   arg_conform(FA.rbroth, GA.rbroth)
ELSE                    true()
    ELSE    false()

FCT rout_conform(Rout_or_Sig F, Rout_or_Sig G) Bool:
    LET FRET  == F.class:
    LET GRET  == G.class:
    LET FARG  == F.argument_list:
    LET GARG  == G.argument_list:
    LET FPRE  == F.pre:
    LET GPRE  == G.pre:
    LET FPOST == F.post:
    LET GPOST == G.post:
    IF F.ident.term = G.ident.term
    && FARG.numsons = GARG.numsons
    && is_subtype(FRET, GRET) = true()
    && {is[G.function_mode, _Not_Functional@] ||
is[F.function_mode, _Functional@]}:
LET H ==
IF FARG.numsons # 0:    arg_conform(FARG.fst, GARG.fst)
ELSE                    true():
IF H = true() && cond_list_conform(FPRE, GPRE) = true():
    cond_list_conform(GPOST, FPOST)
ELSE false()
    ELSE false()

//*************************************************************************
// functions for testing if two interfaces conform according to p. 5
//*************************************************************************

// The set of public routines build the interface of a class
FCT single_interface_conform(CElement C, Signature@ S) Bool:
    LET SR == S.rbroth:
    IF is[C, _Routine_Def@] && is[C.featureid.access_mode, _Public@] &&
       rout_conform(C, S) = true():
IF SR # nil():  single_interface_conform(C.fath.fst, SR)
ELSE            true()
     | is[C, _Signature@] && rout_conform(C, S) = true():
IF SR # nil():  single_interface_conform(C.fath.fst, SR)
ELSE            true()
    ELSE
IF C.rbroth # nil():    single_interface_conform(C.rbroth, S)
ELSE                    false()
```

```
FCT interface_conform(Interface I1, Signature_List@ I2) Bool:
    LET H ==
    IF I2.fst # nil():
IF I1.fst # nil() &&
   single_interface_conform(I1.fst, I2.fst) = true(): true()
ELSE    false()
    ELSE    true():
    IF H = true():  cond_list_conform(I2.fath.invariant, I1.fath.invariant)
    ELSE            false()

//Interfaces may not contain conflicting signatures
//(Private routines shouldn't conflict either)
CND Class@<<ID, _>, *,
    <*, <<_, <ID1, No1>>, *> E1, *, <<_, <ID2, No2>>, *> E2, *>>:
    is[E1, _Attr_Def@] ||
    is[E2, _Attr_Def@] ||
    !rout_conflict(E1, E2) = true()
| `"### " No1.term.linetos " and " No2.term.linetos ": Signature "
  ID1.term.idtos " declared twice in class " ID.term.idtos "\n"'

//The Interface of a class must conform to the interface of each supertype
CND Class@<<ID, No>, <*, S, *>, *, E> C:
    interface_conform(E, S.decl.signature_list) = true()
| `"### " No.term.linetos ": Interface of class " ID.term.idtos
  " doesn't conform to abstract type\n"'

//The Interface of a class must conform to the interface of each supertype
CND AbstractClass@<<ID, No>, _, <*, <I, N> S, *>, _, E> C:
    interface_conform(S.typeclass.signature_list, E) = true()
| `"### " No.term.linetos ": Interface of class " I.term.idtos
  " doesn't conform to abstract type\n"'

//*************************************************************************
// Build the void value of any class
//*************************************************************************

FCT build_void_value(Class@ C) Value:
    IF is[C, _ValueClass@]:
LET I == C.ident.term.idtos:
IF I = "BOOL":  false()
 | I = "CHAR":  0
 | I = "INT":   0
 | I = "STR":   ""
ELSE
    LET CEL == C.class_element_list.fst:
    LET VL ==  IF CEL # nil():      CEL.build_value_list
   ELSE                 Value_List():
    ValObject(C, VL)
    ELSE
Void_Value()
```

```
//*************************************************************************
// Collect values of attributes in a list
//*************************************************************************

FCT build_value_list(CElement C) Value_List:
    LET R == C.rbroth:
    LET L == IF R # nil():  R.build_value_list
     ELSE           Value_List():
    IF is[C, _Attr_Def@]:   appfront(C.class.build_void_value, L)
    ELSE                    L

FCT dvar( Var V, IncarId INC, ObjectId INS) DynamicVar:
    IF is[V, _DynamicVar]:          V
     | is[V, _Self_Expr]:           SelfVar( INC )
     | is[V, _Expr@]:               TempVar( V, INC )
     | is[V, _Typed_Ident@]:        ActVar ( V, INC )
     | is[V, _Attr_Def@]:           InstVar( V, INS )
     | is[V, _Routine_Def@]:        TempVar( V, INC )
    ELSE nil()

DYN ct ()       Task:   LET C == root().statement_list.fst.typespec.decl:
LET R == root().statement_list.snd.receiver:
IF is[C, _RefClass@]:
    CreateObj(R, C, R.after )
ELSE
    MoveVal(R, C.build_void_value, R.after )

DYN cont( DynamicVar V )        Value:          Void_Value()
DYN rt ( IncarId I )            DynamicTask:    nil()
DYN ci ()                       IncarId:        0
DYN nextci ()                   IncarId:        1
DYN no()                        ObjectId:       1
DYN curr_routine( IncarId I )   Routine_Def@:   nil()
DYN pre_state( IncarId I)       Bool:           nil()
DYN dummy()                     Value:          0

FCT succ(Task T) Task:
    IF is[T, _Point]:               psucc(T)
     | is[T, _Move]:                subterm(3, T)
     | is[T, _MoveVal]:             subterm(3, T)
     | is[T, _Select]:              subterm(2, T)
     | is[T, _MoveArg]:             subterm(2, T)
     | is[T, _Void_Test]:           subterm(3, T)
     | is[T, _CreateObj]:           subterm(3, T)
     | is[T, _Set]:                 subterm(2, T)
    ELSE    nil()
```

```
FCT psucc(Point P) Task:
    LET N== P.node:
    IF P = N.after:
IF Program@<_, <_, N>> :        Result(N)
 | Call_Expr@<N, _, <E, *>>:    E.before
 | Call_Expr@<N, _, <>> CE:     Select(CE, Call(CE))
 | Call_Expr@<*, <*, N>> CE:    Select(CE, MoveArg(CE, Call(CE)))
 | Logic_Expr@<N, O, E> L:      Logic(N, L, L.after, E.before)
 | Logic_Expr@<E, O, N> L:      Logic(N, L, L.after, L.after)
 | Set_Expr@<N> S:              Set(S, S.after)
 | Void_Test_Expr@<N> V:        Void_Test(N, V, V.after)
 | Return_Stmt@<N> R:           LET H == R.encl_rout:
Move(H, N, H.post.before)
 | Assign_Stmt@<V, N> A:        Move(V.usedid.decl, N, A.after)
 | If_Stmt@<N, TS, ES> S:       LET TT == IF TS.fst = nil(): S.after
  ELSE before(TS.fst):
LET ET == IF ES.fst = nil(): S.after
  ELSE ES.fst.before:
Branch(N, TT, ET)
 | While_Stmt@<N> W:            LET SL == W.encl_loop.statement_list:
Branch(N, SL.fst.before, SL.after)
 | Break_Stmt@<> N:             LET SL == N.encl_loop.statement_list:
SL.after
 | Typecase_Stmt@<N, <>, ES> S: ES.fst.before
 | Typecase_Stmt@<E, <*, <N, TS>>, ES> S:
    LET TT ==
IF TS.fst = nil():  S.after
ELSE TS.fst.before:
    LET ET ==
IF ES.fst = nil():  Error(`"### " S.loc ": No type matched
    and no else clause given\n"')
ELSE ES.fst.before:
    Typecase(E, N, TT, ET)
 | Typecase_Stmt@<E, <*, <N, TS>, <M, _>, *>, _> S:
    LET TT == IF TS.fst = nil():    S.after
      ELSE TS.fst.before:
    Typecase(E, N, TT, M.after)
 | Statement_List@<*, N> SL:
    LET F == SL.fath:
    IF is[F, _If_Stmt@]:        F.after
     | is[F, _Loop@]:           SL.fst.before
     | is[F, _Typecase_Stmt@]:  F.after
     | is[F, _Alternative@]:    F.fath.fath.after
    ELSE Error(`"### " SL.loc ": Internal error:
 Ran out of statement list\n"')
 | Condition_List@<*> N:    N.after_cond_list
 | Condition@<DID, N> C:
    Branch(N, C.after, Error(`"### " C.loc ": assertion "
       DID.ident.term.idtos " violated\n"'))
 | Initial_Expr@<N> I:      Move(I, N, next_initial(P, N.encl_rout))
ELSE                        P.next
```

```
     | P = N.before:
IF Self_Expr@ N:            Move(N, Self_Expr(), N.after)
 | Local_Expr@<UID> N:      Move(N, UID.decl, N.after)
 | New_Expr@ N:             CreateObj(N, N.encl_class, N.after)
 | Literal_Expr@ N:         MoveVal(N, N.term, N.after)
 | Attr_Expr@ N:            Move(N, N.encl_rout.lbroth, N.after)
 | Void_Expr@ N:            N.void_task
 | Initial_Expr@ N:         N.after
 | Result_Expr@ N:          Move(N, N.encl_rout, N.after)
 | Condition_List@ N:
    LET R == N.encl_rout:
    LET LEVEL ==
    IF N = R.pre:                   1
     | N = R.post:                  2
     | N = N.encl_class.invariant:  3
    ELSE                            4:
    IF check_level(LEVEL) = true(): P.next
    ELSE                            N.after
ELSE P.next
    ELSE nil()

//*************************************************************************
// how two continue after a condition list was checked
//*************************************************************************

FCT after_cond_list(Condition_List@ CL) Task:
    LET R == CL.encl_rout:
    IF CL = R.pre:                          next_initial(R.post.before, R)
     | CL = R.post:                         Return()
     | CL = CL.encl_class.invariant:        Invariant()
    ELSE CL.after.next

//*************************************************************************
// Searching the next appearance of an initial expr.
//*************************************************************************

FCT next_initial(Point P, Routine_Def@ R) Task:
    LET N == P.node:
    IF {check_level(4) = true() && N.encl_rout = R}
    || {check_level(2) = true() && {is[N, _Condition_List@] ||
N.encl_condition # nil()}}:
IF is[N, _Initial_Expr@] && P = N.before:       N.expr.before
ELSE next_initial(P.next, R)
    ELSE
R.statement_list.fst.before
```

```
//*************************************************************************
// Giving the attributes ids
//*************************************************************************

FCT left_attr(CElement A) Attr_Def@:
    LET L == A.lbroth:
    IF is[L, _Attr_Def@]:   L
    ELSE                    L.left_attr

ATT attr_id(Attr_Def@ A) Int:
    LET P == A.left_attr:
    IF P = nil():   1
    ELSE            plus(P.attr_id, 1)

//*************************************************************************
// initialize the attributes of a reference class
//*************************************************************************

FCT init_attr(CElement C, ObjectId O, Task T) Task:
    LET R == C.rbroth:
    LET TASK == IF R # nil():   init_attr(R, O, T)
ELSE T:
    IF is[C, _Attr_Def@]:
MoveVal(dvar(C, 0, O), C.class.build_void_value, TASK)
    ELSE TASK

//*************************************************************************
// modify one value in a value list
//*************************************************************************

FCT set_value(Value_List VL, Int I, Value V) Value_List:
    IF I = 1:       appfront(V, VL.back)
    ELSE            appfront(VL.t_fst, set_value(VL.back, minus(I, 1), V))

//*************************************************************************
// determine the value of Void_Expr
//*************************************************************************

FCT void_task(Void_Expr@ V) Task:
    LET VA == V.after:
    IF Assign_Stmt@<<U>, V> A:
LET C == U.decl.class:
MoveVal(V, C.build_void_value, VA)
     | Return_Stmt@<V> R:
LET C == R.encl_rout.class:
MoveVal(V, C.build_void_value, VA)
     | While_Stmt@<V> W:
MoveVal( V, false(), VA )
     | Expr_List@<*, V, *> EL:
// Type must be determined after selection of routine
MoveVal( V, Void_Value(), VA )
```

```
     | Void_Test_Expr@<V>:
MoveVal( V, Void_Value(), VA )
     | Condition@<_, V>:
MoveVal( V, false(), VA )
     | Initial_Expr@<V>:
MoveVal( V, Void_Value(), VA )
    ELSE    // Never reached (CND detects error)
Error(`"### " V.loc ": Internal error: Illegal use of void\n"')

//*************************************************************************
// function for initializing the parameters of a routine
//*************************************************************************

FCT move_arg(Expr@ E, Typed_Ident@ P, IncarId TO, IncarId FROM, Task T) Task:
    IF E.rbroth # nil():
Move( dvar(P, TO, 0), dvar(E, FROM, 0),
      move_arg(E.rbroth, P.rbroth, TO, FROM, T) )
    ELSE
Move( dvar(P, TO, 0), dvar(E, FROM, 0), T)

//*************************************************************************
// function for initializing the local variables of a routine
// parameters are initialized by move_arg
//*************************************************************************

FCT init_alternative_list(Alternative_List@ al, Task T) Task:
    IF al.fst = nil():      T
    ELSE                    init_alternative(al.fst, T)

FCT init_alternative(Alternative@ A, Task T) Task:
    IF A.rbroth # nil():
 init_statement_list(A.statement_list, init_alternative(A.rbroth, T))
    ELSE init_statement_list(A.statement_list, T)

FCT init_statement_list(Statement_List@ sl, Task T) Task:
    IF sl.fst = nil():      T
    ELSE                    init_locals(sl.fst, T)

FCT init_locals(Statement@ S, Task T) Task:
    LET SR == S.rbroth:
    LET TASK == IF SR = nil():      T
ELSE                init_locals(SR, T):
    IF is[S, _Typed_Ident@]:
MoveVal(S, S.class.build_void_value, TASK)
     | is[S, _If_Stmt@]:
init_statement_list(S.then_list,
init_statement_list(S.else_list, TASK))
     | is[S, _Typecase_Stmt@]:
init_statement_list(S.statement_list,
init_alternative_list(S.alternative_list, TASK))
    ELSE TASK
```

```
//*************************************************************************
// function returns the type of a value
//*************************************************************************

FCT type_of_value(Value V) Class@:
    IF is[V, _Object]:      V.objclass
     | is[V, _Bool]:        root().bool_class
     | is[V, _Char]:        root().char_class
     | is[V, _Int]:         root().int_class
     | is[V, _String]:      root().string_class
    ELSE nil()

//*************************************************************************
// Check for built-in classes
//*************************************************************************

FCT predecl(Ident ID) Bool:
    LET S == ID.idtos:
    IF S = "BOOL"
    || S = "CHAR"
    || S = "INT"
    || S = "STR"
    || S = "SYS"
    || S = "INOUT": true()
    ELSE            false()
}

//*************************************************************************
// Rules for evolving algebra
//*************************************************************************

RULES

CASE ct().sort
IS _Seq_Task THEN
    ct() := ct().succ
IS _Select THEN
    IF is[ct().t_call_expr.target, _Routine_Def@] THEN      // static binding
curr_routine(nextci()) := ct().t_call_expr.target
    ELSE                                                    // dynamic binding
curr_routine(nextci()) := select_rout(ct().t_call_expr.ident,
    ct().t_call_expr.expr_list,
    dvar(ct().t_call_expr.receiver, ci(), 0).cont.type_of_value )
    FI
IS _MoveArg THEN
    ct() := move_arg(ct().t_call_expr.expr_list.fst, curr_routine(
    nextci()).argument_list.fst, nextci(), ci(), ct().succ)
IS _Call THEN
    IF is[ct().t_call_expr.receiver, _Typespec@] THEN
SelfVar(nextci()).cont :=
ct().t_call_expr.receiver.typeclass.build_void_value
```

```
    ELSE
SelfVar(nextci()).cont :=
dvar(ct().t_call_expr.receiver, ci(), 0).cont
    FI

    IF curr_routine(nextci()).encl_class.ident.term.predecl = true() THEN
dvar(ct().t_call_expr, ci(), 0).cont := eval_predecl(curr_routine(
nextci()).encl_class.ident.term.idtos, ct().t_call_expr)
ct() := ct().t_call_expr.after
    ELSE
IF is[ct().t_call_expr.receiver, _Self_Expr@] THEN
    ct() := init_locals(curr_routine(nextci()).statement_list.fst,
    curr_routine(nextci()).pre.before)
ELSE
    pre_state(nextci()) := true()
    ct() := init_locals(curr_routine(nextci()).statement_list.fst,
    curr_routine(nextci()).encl_class.invariant.before)
FI

rt(nextci())    := DynamicTask(ct().t_call_expr.after, ci())
ci()            := nextci()
nextci()        := plus(nextci(), 1)
    FI
IS _Invariant THEN
    IF pre_state(ci()) = true() THEN
ct() := curr_routine(ci()).pre.before
    ELSE
ct() := ci().rt.dyn_task
ci() := ci().rt.incid
dvar(rt(ci()).dyn_task.node, rt(ci()).incid, 0).cont :=
dvar(curr_routine(ci()), ci(), 0).cont
    FI
IS _Return THEN
    IF is[rt(ci()).dyn_task.node.receiver, _Self_Expr@] THEN
ct()    := Invariant()
    ELSE
ct()    := ci().curr_routine.encl_class.invariant.before
    FI
    pre_state(ci())   := false()
IS _Logic THEN
    dvar(ct().t_logic_expr, ci(), 0).cont := dvar(ct().t_expr, ci(), 0).cont
    IF is[ct().t_logic_expr.operator, _And_Op@] THEN
IF dvar(ct().t_expr, ci(), 0).cont = true() THEN
    ct() := ct().t_else_task
ELSE
    ct() := ct().t_then_task
FI
    ELSE
IF dvar(ct().t_expr, ci(), 0).cont = true() THEN
    ct() := ct().t_then_task
ELSE
```

```
    ct() := ct().t_else_task
FI
    FI
IS _Void_Test THEN
    IF dvar(ct().t_expr, ci(), 0).cont = Void_Value() THEN
dvar(ct().t_void_test_expr, ci(), 0).cont := true()
    ELSE
IF eq[dvar(ct().t_expr, ci(), 0).cont,
dvar(ct().t_expr, ci(), 0).cont.type_of_value.build_void_value] THEN
    dvar(ct().t_void_test_expr, ci(), 0).cont := true()
ELSE
    dvar(ct().t_void_test_expr, ci(), 0).cont := false()
FI
    FI
IS _Move THEN
    IF is[ct().t_src, _Attr_Def@] THEN
CASE SelfVar(ci()).cont.sort
IS _Void_Value THEN
    dvar(ct().t_dest, ci(), 0).cont := Void_Value()
IS _RefObject THEN
    dvar(ct().t_dest, ci(), 0).cont :=
    dvar(ct().t_src, ci(), SelfVar(ci()).cont.objid).cont
IS _ValObject THEN
    dvar(ct().t_dest, ci(), 0).cont := subterm(ct().t_src.attr_id,
    SelfVar(ci()).cont.t_value_list)
END
    ELSE
IF is[dvar(ct().t_src, ci(), 0).cont, _Void_Value] THEN
    IF is[dvar(ct().t_dest, ci(), 0), _ActVar] THEN
dvar(ct().t_dest, ci(), 0).cont := build_void_value(
dvar(ct().t_dest, ci(), 0).typed_ident.class)
    ELSE
dvar(ct().t_dest, ci(), 0).cont :=
dvar(ct().t_src, ci(), 0).cont
    FI
ELSE
    dvar(ct().t_dest, ci(), 0).cont := dvar(ct().t_src, ci(), 0).cont
FI
    FI
IS _Branch THEN
    IF dvar(ct().t_expr, ci(), 0).cont = true() THEN
ct() := ct().t_then_task
    ELSE
ct() := ct().t_else_task
    FI
IS _Typecase THEN
    IF is[dvar(ct().t_expr, ci(), 0).cont, _Void_Type] THEN
IF is_subtype(ct().t_expr.usedid.decl.class, ct().t_type.typeclass)
   = true() THEN
    ct() := ct().t_then_task
ELSE
```

```
    ct() := ct().t_else_task
FI
    ELSE
IF is_subtype(dvar(ct().t_expr, ci(), 0).cont.type_of_value,
   ct().t_type.typeclass) = true() THEN
    ct() := ct().t_then_task
ELSE
    ct() := ct().t_else_task
FI
    FI
IS _Set THEN
    IF is[ct().t_expr.encl_class, _RefClass@] THEN
IF SelfVar(ci()).cont # Void_Value() THEN
    dvar(ct().t_expr.encl_rout.lbroth.lbroth, 0,
    SelfVar(ci()).cont.objid).cont :=
    dvar(ct().t_expr.local_expr, ci(), 0).cont
FI
dvar(ct().t_expr, ci(), 0).cont := Void_Value()
    ELSE
dvar(ct().t_expr, ci(), 0).cont := ValObject(ct().t_expr.encl_class,
set_value(SelfVar(ci()).cont.t_value_list,
ct().t_expr.encl_rout.lbroth.lbroth.attr_id,
dvar(ct().t_expr.local_expr, ci(), 0).cont))
    FI
IS _CreateObj THEN
    no()     := plus(no(), 1)
    dvar(ct().t_var, ci(), 0).cont := RefObject(ct().t_class, no())
    IF ct().t_class.class_element_list.fst # nil() THEN
ct() := init_attr(ct().t_class.class_element_list.fst,
no(), ct().succ)
    ELSE
ct() := ct().succ
    FI
IS _MoveVal THEN
    dvar(ct().t_var, ci(), 0).cont := ct().t_value
IS _Result THEN
    dummy() := dvar(ct().t_expr, ci(), 0).cont.print_result
    ct()    := Terminated()
IS _Error THEN
    IF curr_routine(ci()) # nil() THEN
dummy() := out(`"Evaluation interrupted near " curr_routine(ci()).loc
"\n" ct().t_fst')
    ELSE
dummy() := out(ct().t_fst)
    FI
    ct() := Failed()
IS _Terminated THEN
IS _Failed THEN
END
```

B.2.4 Die C-Funktionen

```
/*************************************************************************
 * Module:              SALSA.c                                          *
 * Author:              Peter Müller                                     *
 * Description:         C functions of Sather/SALSA interpreter           *
 *************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "SALSA_spec.h"
#include "common.h"

/*************************************************************************
 * global variables
 *************************************************************************/

char check_mode = 1;

/*************************************************************************
 * main and related functions
 *************************************************************************/

/*
 * Signature:   Program gen_main_call(int, int, char *, Sourcefile, char *)
 * Purpose:     adds call of m::main(parlist) to source file
 */
ELEMENT gen_main_call(int act_arg, int argc, char *argv[], ELEMENT s, char *m)
{
ELEMENT exprlist = Expr_List();

while (act_arg < argc)
    exprlist = appback(exprlist, atoe(argv[act_arg++]));

return Program(s, appback( appback(Statement_List(), Typed_Ident(DeclId(
        stoid(atoe("r")), itoe(-1)), UsedId(stoid(atoe(m)), itoe(-1))) ),
        Call_Expr(Local_Expr(UsedId(stoid(atoe("r")), itoe(-1))),
        UsedId(stoid(atoe("main")), itoe(-1)), exprlist)));
}

/*
 * Signature:   void usage(void)
 * Purpose:     prints command line options and exits
 */
void usage(void)
{
printf("Usage: SALSA [-m] [-cn] source[.sa]\n");
printf("Options  -m MAIN  start with routine main in specified class\n");
printf("         -c0      no checks\n");
printf("         -c1      check preconditions (default)\n");
printf("         -c2      check pre- and postconditions\n");
```

```
printf("         -c3      check invariants\n");
printf("         -c4      check all \n");

exit(EXIT_SUCCESS);
}

/*
 * Signature:   int main(int argc, char *argv[])
 * Purpose:     reads, analyses and evaluates source file
 */
int main(int argc, char *argv[])
{
ELEMENT elem;           /* Sourcefile, Program */
int act_arg = 1;
char *m = "MAIN";

while(act_arg < argc && argv[act_arg][0] == '-')
    {
    switch(argv[act_arg][1])
        {
        case 'c':
            check_mode = argv[act_arg][2] - '0';
            if (check_mode < 0)
                check_mode = 0;
            if (check_mode > 4)
                check_mode = 4;
            break;
        case 'm':
            act_arg++;
            if (act_arg < argc)
                m = argv[act_arg];
            break;
        default:
            fprintf(stderr,"Illegal command line option %s\n",argv[act_arg]);
            usage();
        }
    act_arg++;
    }

if (act_arg >= argc)
    {
    fprintf(stderr, "Missing filename\n");
    usage();
    }

elem = parse(argv[act_arg++]);

printf("\nThis is the Sather Interpreter\n");
printf("+++ analysing\n");

elem = resolve_subtyping(elem);
```

```
elem = resolve_includes(elem);
elem = gen_main_call(act_arg, argc, argv, elem, m);

if ( init_s(elem) )
    {
    printf("+++ evaluating\n");
    init_dyninput();
    eval_dynsem();
    printf("+++ finished\n");
    }

return EXIT_SUCCESS;
}

/*
 * Signature:   Int print_result(Value e)
 * Purpose:     prints the return value of MAIN::main
 */
ELEMENT print_result(ELEMENT e)
{
printf("+++ main-routine evaluates to ");
switch (sort(e))
    {
    case _Bool:
        if (etob(e))
            printf("True\n");
        else
            printf("False\n");
        break;
    case _Char:
        printf("%c\n", etoc(e));
        break;
    case _Int:
        printf("%d\n", etoi(e));
        break;
    case _String:
        printf("%s\n", etoa(e));
        break;
    case _Void_Value:
        printf("Void\n");
        break;
    default:
        printf("an illegal object\n");
        fprintf(stderr, "Illegal return type of routine main\n");
        exit( EXIT_FAILURE);
    }

return (itoe(0));
}
```

```
/*************************************************************************
 * Implementation of predefined classes
 *************************************************************************/

/*
 * Purpose:     prints message for violated conditions
 */
void violation(char *rout, char *ass)
{
printf("Evaluation interrupted near ROUTINE %s\n", rout);
printf("ROUTINE %s: assertion %s violated\n", rout, ass);
exit(EXIT_SUCCESS);
}


/*
 * Signature:   Value eval_inout(Call_Expr@ CE)
 * Purpose:     evaluates calls on INOUT
 */
ELEMENT eval_inout(ELEMENT CE)
{
char *a = etoa(idtos(term(ident(CE))));
ELEMENT self;           /* Value */
long i;
char c;
char buffer[500];

if (is(term(receiver(CE)), _UsedId))
    self = Void_Value();
else
    self = cont ( dvar(receiver(CE), ci, itoe(0)) );

switch(numsons(expr_list(CE)))
    {
    case 0:
        if (strcmp(a, "read_int") == 0)
            {
            scanf("%d", &i);
            return itoe(i);
            }
        if (strcmp(a, "read_char") == 0)
            {
            scanf("%c", &c);
            return ctoe(c);
            }
        if (strcmp(a, "read_string") == 0)
            {
            scanf("%s", buffer);
            return atoe(buffer);
            }
        if (strcmp(a, "nl") == 0)
```

```
            {
            printf("\n");
            return self;
            }
        break;
    case 1:
        if (strcmp(a, "write_int") == 0)
            {
            printf("%d", etoi(cont(dvar(fst(expr_list(CE)), ci, itoe(0)))));
            return self;
            }
        if (strcmp(a, "write_char") == 0)
            {
            printf("%c", etoc(cont(dvar(fst(expr_list(CE)), ci, itoe(0)))));
            return self;
            }
        if (strcmp(a, "write_string") == 0)
            {
            printf("%s", etoa(cont(dvar(fst(expr_list(CE)), ci, itoe(0)))));
            return self;
            }
    }

fprintf(stderr, "Illegal routine call INOUT::%s\n", a);
exit (EXIT_FAILURE);
}

/*
 * Signature:   Value eval_bool(Call_Expr@ CE)
 * Purpose:     evaluates calls on BOOL
 */
ELEMENT eval_bool(ELEMENT CE)
{
char *a = etoa(idtos(term(ident(CE))));
long self;
long par1;

if (is(term(receiver(CE)), _UsedId) ||
    sort(cont(dvar(receiver(CE), ci, itoe(0)))) == _Void_Value)
    self = 0;
else
    self = etob(cont ( dvar(receiver(CE), ci, itoe(0)) ));

switch(numsons(expr_list(CE)))
    {
    case 0:
        if (strcmp(a, "not") == 0)
            return btoe(!self);
        break;
    case 1:
        par1 = etob(cont ( dvar(fst(expr_list(CE)), ci, itoe(0)) ));
```

```
        if (strcmp(a, "is_eq") == 0)
            return btoe(self == par1);
        if (strcmp(a, "is_neq") == 0)
            return btoe(self != par1);
    }

fprintf(stderr, "Illegal routine call BOOL::%s\n", a);
exit (EXIT_FAILURE);
}


/*
 * Signature:   Value eval_int(Call_Expr@ CE)
 * Purpose:     evaluates calls on INT
 */
ELEMENT eval_int(ELEMENT CE)
{
char *a = etoa(idtos(term(ident(CE))));
int self;
int par1;
int i, res, help;

if (is(term(receiver(CE)), _UsedId) ||
    is(cont(dvar(receiver(CE), ci, itoe(0))), _Void_Value))
    self = 0;
else
    self = etoi(cont ( dvar(receiver(CE), ci, itoe(0)) ));

switch(numsons(expr_list(CE)))
    {
    case 0:
        if (strcmp(a, "negate") == 0)
            return (itoe(-self));
        if (strcmp(a, "abs") == 0)
            return (self < 0) ? (itoe(-self)) : (itoe(self));
        break;
    case 1:
        par1 = etoi(cont ( dvar(fst(expr_list(CE)), ci, itoe(0)) ));
        if (strcmp(a, "plus") == 0)
            return itoe(self + par1);
        if (strcmp(a, "minus") == 0)
            return itoe(self - par1);
        if (strcmp(a, "times") == 0)
            return itoe(self * par1);
        if (strcmp(a, "div") == 0)
            if (par1)
                return itoe(self / par1);
            else
                violation("INT::div", "div_by_zero");
        if (strcmp(a, "pow") == 0)
            if (par1 <0)
                violation("INT::pow", "pos_exponent");
```

```
            else
                {
                res = 1;
                for (i= par1; i>0; i--)
                    res *= self;

                return itoe(res);
                }
        if (strcmp(a, "mod") == 0)
            if (par1)
                return itoe(self % par1);
            else
                violation("INT::mod", "mod_by_zero");
        if (strcmp(a, "is_lt") == 0)
            return  btoe(self < par1);
        if (strcmp(a, "is_leq") == 0)
            return  btoe(self <= par1);
        if (strcmp(a, "is_eq") == 0)
            return btoe(self == par1);
        if (strcmp(a, "is_neq") == 0)
            return  btoe(self != par1);
        if (strcmp(a, "is_gt") == 0)
            return  btoe(self > par1);
        if (strcmp(a, "is_geq") == 0)
            return  btoe(self >= par1);
        if (strcmp(a, "min") == 0)
            return (self < par1) ? self : par1;
        if (strcmp(a, "max") == 0)
            return (self > par1) ? self : par1;
    }

fprintf(stderr, "Illegal routine call INT::%s\n", a);
exit (EXIT_FAILURE);
}

/*
 * Signature:   Value eval_char(Call_Expr@ CE)
 * Purpose:     evaluates calls on CHAR
 */
ELEMENT eval_char(ELEMENT CE)
{
char *a = etoa(idtos(term(ident(CE))));
char self;
char *res;
char par1;

if (is(term(receiver(CE)), _UsedId) ||
    is(cont(dvar(receiver(CE), ci, itoe(0))), _Void_Value))
    self = '\0';
else
    self = etoc(cont ( dvar(receiver(CE), ci, itoe(0)) ));
```

```
switch(numsons(expr_list(CE)))
    {
    case 0:
        if (strcmp(a, "ctoa") == 0)
            {
            res = calloc(2, sizeof(char));
            if (res == NULL)
                {
                fprintf(stderr, "Not enough memory\n");
                exit(EXIT_FAILURE);
                }
            *res = self;
            return atoe(res);
            }
        break;
    case 1:
        par1 = etoc(cont ( dvar(fst(expr_list(CE)), ci, itoe(0)) ));
        if (strcmp(a, "is_lt") == 0)
            return  btoe(self < par1);
        if (strcmp(a, "is_leq") == 0)
            return  btoe(self <= par1);
        if (strcmp(a, "is_eq") == 0)
            return  btoe(self == par1);
        if (strcmp(a, "is_neq") == 0)
            return  btoe(self != par1);
        if (strcmp(a, "is_gt") == 0)
            return  btoe(self > par1);
        if (strcmp(a, "is_geq") == 0)
            return  btoe(self >= par1);
    }

fprintf(stderr, "Illegal routine call CHAR::%s\n", a);
exit (EXIT_FAILURE);
}

/*
 * Signature:   Value eval_str(Call_Expr@ CE)
 * Purpose:     evaluates calls on STR
 */
ELEMENT eval_str(ELEMENT CE)
{
char *a = etoa(idtos(term(ident(CE))));
char *self;
char *s;
char c;
ELEMENT par;            /* Value */

if (is(term(receiver(CE)), _UsedId) ||
    is(cont(dvar(receiver(CE), ci, itoe(0))), _Void_Value))
    self = "";
else
```

```
    self = etoa(cont ( dvar(receiver(CE), ci, itoe(0)) ));

switch(numsons(expr_list(CE)))
    {
    case 0:
        if (strcmp(a, "atoi") == 0)
            return itoe(atoi(self));
        if (strcmp(a, "first") == 0)
            if (strlen(self))
                return ctoe(*self);
            else
                violation("STR::first", "not_empty");
        if (strcmp(a, "rest") == 0)
            if (strlen(self))
                return atoe(self+1);
            else
                violation("STR::rest", "not_empty");
        if (strcmp(a, "is_empty") == 0)
            return btoe(!strlen(self));
        if (strcmp(a, "strlen") == 0)
            return itoe(strlen(self));
        break;
    case 1:
        if (strcmp(a, "is_eq") == 0)
            {
            s = etoa(cont ( dvar(fst(expr_list(CE)), ci, itoe(0)) ));
            return btoe (strcmp(self, s) == 0);
            }
        if (strcmp(a, "is_neq") == 0)
            {
            s = etoa(cont ( dvar(fst(expr_list(CE)), ci, itoe(0)) ));
            return btoe (strcmp(self, s) != 0);
            }
        if (strcmp(a, "is_lt") == 0)
            {
            s = etoa(cont ( dvar(fst(expr_list(CE)), ci, itoe(0)) ));
            return btoe (strcmp(self, s) < 0);
            }
        if (strcmp(a, "is_gt") == 0)
            {
            s = etoa(cont ( dvar(fst(expr_list(CE)), ci, itoe(0)) ));
            return btoe (strcmp(self, s) > 0);
            }
        if (strcmp(a, "is_leq") == 0)
            {
            s = etoa(cont ( dvar(fst(expr_list(CE)), ci, itoe(0)) ));
            return btoe (strcmp(self, s) <= 0);
            }
        if (strcmp(a, "is_geq") == 0)
            {
            s = etoa(cont ( dvar(fst(expr_list(CE)), ci, itoe(0)) ));
```

```
            return btoe (strcmp(self, s) >= 0);
            }
        if (strcmp(a, "plus") == 0)
            {
            par = cont ( dvar(fst(expr_list(CE)), ci, itoe(0)) );
            if (is(par, _String))
                {
                s = calloc(strlen(self)+strlen(etoa(par))+1, sizeof(char));
                if (s == NULL)
                    {
                    fprintf(stderr,"Not enough memory\n");
                    return EXIT_FAILURE;
                    }
                strcpy(s, self);
                strcat(s, etoa(par));
                return atoe(s);
                }
            else
                {
                s = calloc(strlen(self) + 2, sizeof(char));
                if (s == NULL)
                    {
                    fprintf(stderr,"Not enough memory\n");
                    return EXIT_FAILURE;
                    }
                sprintf(s, "%s%c", self, etoc(par));
                return atoe(s);
                }
            }
    }
fprintf(stderr, "Illegal routine call STR::%s\n", a);
exit (EXIT_FAILURE);
}


/*
 * Signature:   Value eval_sys(Call_Expr@ CE)
 * Purpose:     evaluates calls on SYS
 */
ELEMENT eval_sys(ELEMENT CE)
{
char *a = etoa(idtos(term(ident(CE))));
ELEMENT t;              /* Class@ */
ELEMENT cl;             /* Class_List@ */
int i;
ELEMENT par;            /* Value */

switch(numsons(expr_list(CE)))
    {
    case 1:
        par = cont(dvar(fst(expr_list(CE)), ci, itoe(0)));
        cl = sourcefile(root());
```

```
        if (strcmp(a, "tp") == 0)
            {
            t = type_of_value(par);
            for (i=1; i<= numsons(cl) && son(i, cl) != t; i++) ;
            return itoe(i);
            }
        if (strcmp(a, "str_for_tp") == 0)
            {
            i = etoi(par);
            if (i<1 || i>numsons(cl))
                return atoe("Illegal tp");

            return idtos(term(ident(son(i, cl))));
            }
        if (strcmp(a, "exit") == 0)
            exit(etoi(par));
        break;
    case 2:
        if (strcmp(a, "ob_eq") == 0)
            return btoe(eq(cont( dvar(fst(expr_list(CE)), ci, itoe(0))),
                        cont( dvar(snd(expr_list(CE)), ci, itoe(0))) ));
    }

fprintf(stderr, "Illegal routine call SYS::%s\n", a);
exit (EXIT_FAILURE);
}

/*
 * Signature:   Value eval_sys(String class, Call_Expr@ CE)
 * Purpose:     evaluates calls on predefined classes
 */
ELEMENT eval_predecl(ELEMENT class, ELEMENT CE)
{
char *c = etoa(class);

if (strcmp(c, "INT") == 0)
    return eval_int(CE);
if (strcmp(c, "CHAR") == 0)
    return eval_char(CE);
if (strcmp(c, "BOOL") == 0)
    return eval_bool(CE);
if (strcmp(c, "STR") == 0)
    return eval_str(CE);
if (strcmp(c, "INOUT") == 0)
    return eval_inout(CE);
if (strcmp(c, "SYS") == 0)
    return eval_sys(CE);

fprintf(stderr, "Predeclared class %s not implemented\n", c);
exit(EXIT_FAILURE);
}
```

```
/*************************************************************************
 * functions supplementing s_spec.m
 *************************************************************************/

/*
 * Signature:   Point next(Point P)
 * Purpose:     returns next point in syntax tree
 */
ELEMENT next( ELEMENT P )
{
if( node(P) == root() )
    return nil();
else
    return P+1;
}

/*
 * Signature:   Int plus(Int a, Int b)
 * Purpose:     implements +
 */
ELEMENT plus(ELEMENT a, ELEMENT b)
{
return itoe(etoi(a) + etoi(b));
}

/*
 * Signature:   Int minus(Int a, Int b)
 * Purpose:     implements -
 */
ELEMENT minus(ELEMENT a, ELEMENT b)
{
return itoe(etoi(a) - etoi(b));
}

/*
 * Signature:   Class@ bool_class1(Int a)
 * Purpose:     returns Class@
 */
ELEMENT bool_class1(ELEMENT a)
{
return lookup(stoid(atoe("BOOL")), fst(sourcefile(root())));    /* Class@ */
}

/*
 * Signature:   Class@ char_class1(Int a)
 * Purpose:     returns Class@
 */
ELEMENT char_class1(ELEMENT a)
{
return lookup(stoid(atoe("CHAR")), fst(sourcefile(root())));      /* Class@ */
}
```

```
/*
 * Signature:   Class@ string_class1(Int a)
 * Purpose:     returns Class@
 */
ELEMENT string_class1(ELEMENT a)
{
return lookup(stoid(atoe("STR")), fst(sourcefile(root())));       /* Class@ */
}

/*
 * Signature:   Class@ int_class1(Int a)
 * Purpose:     returns Class@
 */
ELEMENT int_class1(ELEMENT a)
{
return lookup(stoid(atoe("INT")), fst(sourcefile(root())));       /* Class@ */
}

/*
 * Signature:   Bool check_pre(Int a)
 * Purpose:     returns true, if condition level a is turned on
 */
ELEMENT check_level(ELEMENT a)
{
return btoe(check_mode >= etoi(a));
}

/*
 * Signature:   Int out(String s)
 * Purpose:     prints any string
 */
ELEMENT out(ELEMENT s)
{
printf("%s", etoa(s));
return itoe(0);
}
```

```
/*************************************************************************
 * Module:              common.h                                         *
 * Author:              Peter Müller                                     *
 * Description:         common header for interpreter and AutoDoc        *
 *************************************************************************/

/*************************************************************************
 * data types for type graph and include graph
 *************************************************************************/

typedef struct edge
    {
    ELEMENT super;          /* Ident */
    ELEMENT sub;            /* Ident */
    struct edge *next;
    } edge;

typedef struct node
    {
    ELEMENT type;           /* Ident */
    int id;
    struct node *next;
    } node;

typedef int *graph_matrix;

typedef struct
    {
    node *first_node;
    int nodes;
    edge *first_edge;
    graph_matrix matrix;
    graph_matrix trans;
    ELEMENT *class_name;    /* Ident */
    } graph;

extern graph type_graph;
extern graph include_graph;

/*************************************************************************
 * external functions and variables
 *************************************************************************/

extern int lookup_node(ELEMENT type, graph *g);
extern ELEMENT parse(char *file);
extern ELEMENT resolve_includes(ELEMENT);
extern ELEMENT resolve_subtyping(ELEMENT);
```

```
/*************************************************************************
 * Module:              common.c                                         *
 * Author:              Peter Müller                                     *
 * Description:         common C functions of Interpreter and AutoDoc    *
 *************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#ifdef AUTO
    #include "AutoDoc_spec.h"
#else
    #include "SALSA_spec.h"
#endif
#include "common.h"

/*************************************************************************
 * functions supporting main
 *************************************************************************/

/*
 * Signature:   Sourcefile parse(char *file)
 * Purpose:     build filename and start parser
 */
ELEMENT parse(char *file)
{
extern  ELEMENT yyread( FILE * );

ELEMENT elem;   /* Sourcefile */
FILE *fileptr;
char *filename = (char *)calloc(strlen(file) + 3, sizeof(char));

if (filename == NULL)
    {
    fprintf(stderr,"Not enough memory\n");
    exit(EXIT_FAILURE);
    }

strcpy(filename, file);

if(strcmp(filename+strlen(filename)-3, ".sa"))
    strcat(filename, ".sa");

if( !( fileptr = fopen(filename,"r") ) )
    {
    fprintf(stderr,"Can't open file %s for input\n",filename);
    exit(EXIT_FAILURE);
    }

printf("+++ reading\n");
```

```
if( (elem = yyread( fileptr )) == nil() )
    {
    fprintf(stderr,"Fatal read error: exit\n",filename);
    exit(EXIT_FAILURE);
    }

return elem;
}

/*************************************************************************
 * functions supplementing s_spec.m
 *************************************************************************/

/*
 * Signature:   String linetos(Int i)
 * Purpose:     returns line number as string
 */
ELEMENT linetos(ELEMENT i)
{
char buffer[20];

if (etoi(i) == -1)
    return atoe("internal declaration");

sprintf(buffer, "LINE %d", etoi(i));
return atoe(buffer);
}

/*************************************************************************
 * Declaration of predeclared classes
 *************************************************************************/

/*
 * Signature:   Sourcefile add_predecl_classes(Sourcefile s)
 * Purpose:     add definition of predefined classes to syntax tree
 */
ELEMENT add_predecl_classes(ELEMENT s)
{
ELEMENT ce;             /* Class_Element_List */

/* $OB */
s = appback(s, AbstractClass(DeclId(stoid(atoe("$OB")), itoe(-1)),
    Typespec_List(), Typespec_List(), Condition_List(), Signature_List()));
/* BOOL */
ce = appback(Class_Element_List(), Routine_Def(FeatureId(Public(), DeclId(
     stoid(atoe("not")), itoe(-1))), Functional(), Argument_List(),
     Same_Type(), Condition_List(), Condition_List(), Statement_List()));
ce = appback(ce, Routine_Def(FeatureId(Public(), DeclId(stoid(atoe(
     "is_eq")), itoe(-1))), Functional(), appback(Argument_List(),
     Typed_Ident(DeclId(stoid(atoe("x")), itoe(-1)), Same_Type())),
     Same_Type(), Condition_List(), Condition_List(), Statement_List()));
```

```
ce = appback(ce, Routine_Def(FeatureId(Public(), DeclId(stoid(atoe(
     "is_neq")), itoe(-1))), Functional(), appback(Argument_List(),
     Typed_Ident(DeclId(stoid(atoe("x")), itoe(-1)), Same_Type())),
     Same_Type(), Condition_List(), Condition_List(), Statement_List()));
s = appback(s, ValueClass(DeclId(stoid(atoe("BOOL")), itoe(-1)),
    Typespec_List(), Condition_List(), ce));
/* CHAR */
ce = appback(Class_Element_List(), Routine_Def(FeatureId(Public(), DeclId(
     stoid(atoe("ctoa")), itoe(-1))), Functional(), Argument_List(),
     UsedId(stoid(atoe("STR")), itoe(-1)), Condition_List(),
     Condition_List(), Statement_List()));
ce = appback(ce, Routine_Def(FeatureId(Public(), DeclId(stoid(atoe(
     "is_lt")), itoe(-1))), Functional(), appback(Argument_List(),
     Typed_Ident(DeclId(stoid(atoe("x")), itoe(-1)), Same_Type())),
     UsedId(stoid(atoe("BOOL")), itoe(-1)), Condition_List(),
     Condition_List(), Statement_List()));
ce = appback(ce, Routine_Def(FeatureId(Public(), DeclId(stoid(atoe(
     "is_leq")), itoe(-1))), Functional(), appback(Argument_List(),
     Typed_Ident(DeclId(stoid(atoe("x")), itoe(-1)), Same_Type())),
     UsedId(stoid(atoe("BOOL")), itoe(-1)), Condition_List(),
     Condition_List(), Statement_List()));
ce = appback(ce, Routine_Def(FeatureId(Public(), DeclId(stoid(atoe(
     "is_eq")), itoe(-1))), Functional(), appback(Argument_List(),
     Typed_Ident(DeclId(stoid(atoe("x")), itoe(-1)), Same_Type())),
     UsedId(stoid(atoe("BOOL")), itoe(-1)), Condition_List(),
     Condition_List(), Statement_List()));
ce = appback(ce, Routine_Def(FeatureId(Public(), DeclId(stoid(atoe(
     "is_neq")), itoe(-1))), Functional(), appback(Argument_List(),
     Typed_Ident(DeclId(stoid(atoe("x")), itoe(-1)), Same_Type())),
     UsedId(stoid(atoe("BOOL")), itoe(-1)), Condition_List(),
     Condition_List(), Statement_List()));
ce = appback(ce, Routine_Def(FeatureId(Public(), DeclId(stoid(atoe(
     "is_gt")), itoe(-1))), Functional(), appback(Argument_List(),
     Typed_Ident(DeclId(stoid(atoe("x")), itoe(-1)), Same_Type())),
     UsedId(stoid(atoe("BOOL")), itoe(-1)), Condition_List(),
     Condition_List(), Statement_List()));
ce = appback(ce, Routine_Def(FeatureId(Public(), DeclId(stoid(atoe(
     "is_geq")), itoe(-1))), Functional(), appback(Argument_List(),
     Typed_Ident(DeclId(stoid(atoe("x")), itoe(-1)), Same_Type())),
     UsedId(stoid(atoe("BOOL")), itoe(-1)), Condition_List(),
     Condition_List(), Statement_List()));
s = appback(s, ValueClass(DeclId(stoid(atoe("CHAR")), itoe(-1)),
    Typespec_List(), Condition_List(), ce));
/* INT */
ce = appback(Class_Element_List(), Routine_Def(FeatureId(Public(), DeclId(
     stoid(atoe("plus")), itoe(-1))), Functional(), appback(Argument_List(),
     Typed_Ident(DeclId(stoid(atoe("x")), itoe(-1)), Same_Type())),
     Same_Type(), Condition_List(), Condition_List(), Statement_List()));
```

```
ce = appback(ce, Routine_Def(FeatureId(Public(), DeclId(stoid(atoe(
     "minus")), itoe(-1))), Functional(), appback(Argument_List(),
     Typed_Ident(DeclId(stoid(atoe("x")), itoe(-1)), Same_Type())),
     Same_Type(), Condition_List(), Condition_List(), Statement_List()));
ce = appback(ce, Routine_Def(FeatureId(Public(), DeclId(stoid(atoe(
     "times")), itoe(-1))), Functional(), appback(Argument_List(),
     Typed_Ident(DeclId(stoid(atoe("x")), itoe(-1)), Same_Type())),
     Same_Type(), Condition_List(), Condition_List(), Statement_List()));
ce = appback(ce, Routine_Def(FeatureId(Public(), DeclId(stoid(atoe(
     "div")), itoe(-1))), Functional(), appback(Argument_List(),
     Typed_Ident(DeclId(stoid(atoe("x")), itoe(-1)), Same_Type())),
     Same_Type(), appback(Condition_List(), Condition(DeclId(stoid(atoe(
     "div_by_zero")), itoe(-1)), Call_Expr(Local_Expr(UsedId(stoid(atoe(
     "x")), itoe(-1))), UsedId(stoid(atoe("is_neq")), itoe(-1)), appback(
     Expr_List(), itoe(0))))), Condition_List(), Statement_List()));
ce = appback(ce, Routine_Def(FeatureId(Public(), DeclId(stoid(atoe("pow")),
     itoe(-1))), Functional(), appback(Argument_List(), Typed_Ident(
     DeclId(stoid(atoe("x")), itoe(-1)), Same_Type())), Same_Type(),
     appback(Condition_List(), Condition(DeclId(stoid(atoe("pos_exponent")),
     itoe(-1)), Call_Expr(Local_Expr(UsedId(stoid(atoe("x")), itoe(-1))),
     UsedId(stoid(atoe("is_geq")), itoe(-1)), appback(Expr_List(),
     itoe(0))))), Condition_List(), Statement_List()));
ce = appback(ce, Routine_Def(FeatureId(Public(), DeclId(stoid(atoe("mod")),
     itoe(-1))), Functional(), appback(Argument_List(), Typed_Ident(
     DeclId(stoid(atoe("x")), itoe(-1)), Same_Type())), Same_Type(),
     appback(Condition_List(), Condition(DeclId(stoid(atoe("mod_by_zero")),
     itoe(-1)), Call_Expr(Local_Expr(UsedId(stoid(atoe("x")), itoe(-1))),
     UsedId(stoid(atoe("is_neq")), itoe(-1)), appback(Expr_List(),
     itoe(0))))), Condition_List(), Statement_List()));
ce = appback(ce, Routine_Def(FeatureId(Public(), DeclId(stoid(atoe(
     "is_lt")), itoe(-1))), Functional(), appback(Argument_List(),
     Typed_Ident(DeclId(stoid(atoe("x")), itoe(-1)), Same_Type())),
     UsedId(stoid(atoe("BOOL")), itoe(-1)), Condition_List(),
     Condition_List(), Statement_List()));
ce = appback(ce, Routine_Def(FeatureId(Public(), DeclId(stoid(atoe(
     "is_leq")), itoe(-1))), Functional(), appback(Argument_List(),
     Typed_Ident(DeclId(stoid(atoe("x")), itoe(-1)), Same_Type())),
     UsedId(stoid(atoe("BOOL")), itoe(-1)), Condition_List(),
     Condition_List(), Statement_List()));
ce = appback(ce, Routine_Def(FeatureId(Public(), DeclId(stoid(atoe(
     "is_eq")), itoe(-1))), Functional(), appback(Argument_List(),
     Typed_Ident(DeclId(stoid(atoe("x")), itoe(-1)), Same_Type())),
     UsedId(stoid(atoe("BOOL")), itoe(-1)), Condition_List(),
     Condition_List(), Statement_List()));
ce = appback(ce, Routine_Def(FeatureId(Public(), DeclId(stoid(atoe(
     "is_neq")), itoe(-1))), Functional(), appback(Argument_List(),
     Typed_Ident(DeclId(stoid(atoe("x")), itoe(-1)), Same_Type())),
     UsedId(stoid(atoe("BOOL")), itoe(-1)), Condition_List(),
     Condition_List(), Statement_List()));
```

```
ce = appback(ce, Routine_Def(FeatureId(Public(), DeclId(stoid(atoe(
     "is_gt")), itoe(-1))), Functional(), appback(Argument_List(),
     Typed_Ident(DeclId(stoid(atoe("x")), itoe(-1)), Same_Type())),
     UsedId(stoid(atoe("BOOL")), itoe(-1)), Condition_List(),
     Condition_List(), Statement_List()));
ce = appback(ce, Routine_Def(FeatureId(Public(), DeclId(stoid(atoe(
     "is_geq")), itoe(-1))), Functional(), appback(Argument_List(),
     Typed_Ident(DeclId(stoid(atoe("x")), itoe(-1)), Same_Type())),
     UsedId(stoid(atoe("BOOL")), itoe(-1)), Condition_List(),
     Condition_List(), Statement_List()));
ce = appback(ce, Routine_Def(FeatureId(Public(), DeclId(stoid(atoe(
     "negate")), itoe(-1))), Functional(), Argument_List(), Same_Type(),
     Condition_List(), Condition_List(), Statement_List()));
ce = appback(ce, Routine_Def(FeatureId(Public(), DeclId(stoid(atoe("abs")),
     itoe(-1))), Functional(), Argument_List(), Same_Type(),
     Condition_List(), Condition_List(), Statement_List()));
ce = appback(ce, Routine_Def(FeatureId(Public(), DeclId(stoid(atoe("min")),
     itoe(-1))), Functional(), appback(Argument_List(), Typed_Ident(
     DeclId(stoid(atoe("x")), itoe(-1)), Same_Type())), Same_Type(),
     Condition_List(), Condition_List(), Statement_List()));
ce = appback(ce, Routine_Def(FeatureId(Public(), DeclId(stoid(atoe("max")),
     itoe(-1))), Functional(), appback(Argument_List(), Typed_Ident(
     DeclId(stoid(atoe("x")), itoe(-1)), Same_Type())), Same_Type(),
     Condition_List(), Condition_List(), Statement_List()));
s = appback(s, ValueClass(DeclId(stoid(atoe("INT")), itoe(-1)),
    Typespec_List(), Condition_List(), ce));
/* STR */
ce = appback(Class_Element_List(), Routine_Def(FeatureId(Public(),
     DeclId(stoid(atoe("plus")), itoe(-1))), Functional(), appback(
     Argument_List(), Typed_Ident(DeclId(stoid(atoe("x")), itoe(-1)),
     Same_Type())), Same_Type(), Condition_List(), Condition_List(),
     Statement_List()));
ce = appback(ce, Routine_Def(FeatureId(Public(), DeclId(stoid(atoe("plus")),
     itoe(-1))), Functional(), appback(Argument_List(), Typed_Ident(DeclId(
     stoid(atoe("x")), itoe(-1)), UsedId(stoid(atoe("CHAR")), itoe(-1)))),
     Same_Type(), Condition_List(), Condition_List(), Statement_List()));
ce = appback(ce, Routine_Def(FeatureId(Public(), DeclId(stoid(atoe(
     "is_lt")), itoe(-1))), Functional(), appback(Argument_List(),
     Typed_Ident(DeclId(stoid(atoe("x")), itoe(-1)), Same_Type())),
     UsedId(stoid(atoe("BOOL")), itoe(-1)), Condition_List(),
     Condition_List(), Statement_List()));
ce = appback(ce, Routine_Def(FeatureId(Public(), DeclId(stoid(atoe(
     "is_leq")), itoe(-1))), Functional(), appback(Argument_List(),
     Typed_Ident(DeclId(stoid(atoe("x")), itoe(-1)), Same_Type())),
     UsedId(stoid(atoe("BOOL")), itoe(-1)), Condition_List(),
     Condition_List(), Statement_List()));
ce = appback(ce, Routine_Def(FeatureId(Public(), DeclId(stoid(atoe(
     "is_eq")), itoe(-1))), Functional(), appback(Argument_List(),
     Typed_Ident(DeclId(stoid(atoe("x")), itoe(-1)), Same_Type())),
     UsedId(stoid(atoe("BOOL")), itoe(-1)), Condition_List(),
     Condition_List(), Statement_List()));
```

```
ce = appback(ce, Routine_Def(FeatureId(Public(), DeclId(stoid(atoe(
     "is_neq")), itoe(-1))), Functional(), appback(Argument_List(),
     Typed_Ident(DeclId(stoid(atoe("x")), itoe(-1)), Same_Type())),
     UsedId(stoid(atoe("BOOL")), itoe(-1)), Condition_List(),
     Condition_List(), Statement_List()));
ce = appback(ce, Routine_Def(FeatureId(Public(), DeclId(stoid(atoe(
     "is_gt")), itoe(-1))), Functional(), appback(Argument_List(),
     Typed_Ident(DeclId(stoid(atoe("x")), itoe(-1)), Same_Type())),
     UsedId(stoid(atoe("BOOL")), itoe(-1)), Condition_List(),
     Condition_List(), Statement_List()));
ce = appback(ce, Routine_Def(FeatureId(Public(), DeclId(stoid(atoe(
     "is_geq")), itoe(-1))), Functional(), appback(Argument_List(),
     Typed_Ident(DeclId(stoid(atoe("x")), itoe(-1)), Same_Type())),
     UsedId(stoid(atoe("BOOL")), itoe(-1)), Condition_List(),
     Condition_List(), Statement_List()));
ce = appback(ce, Routine_Def(FeatureId(Public(), DeclId(stoid(atoe(
     "atoi")), itoe(-1))), Functional(), Argument_List(),
     UsedId(stoid(atoe("INT")), itoe(-1)), Condition_List(),
     Condition_List(), Statement_List()));
ce = appback(ce, Routine_Def(FeatureId(Public(), DeclId(stoid(atoe(
     "first")), itoe(-1))), Functional(), Argument_List(),
     UsedId(stoid(atoe("CHAR")), itoe(-1)), appback(Condition_List(),
     Condition(DeclId(stoid(atoe("not_empty")), itoe(-1)),
     Call_Expr(Call_Expr(Self_Expr(), UsedId(stoid(atoe("is_empty")),
     itoe(-1)), Expr_List()), UsedId(stoid(atoe("not")), itoe(-1)),
     Expr_List()))), Condition_List(), Statement_List()));
ce = appback(ce, Routine_Def(FeatureId(Public(), DeclId(stoid(atoe("rest")),
     itoe(-1))), Functional(), Argument_List(), Same_Type(), appback(
     Condition_List(), Condition(DeclId(stoid(atoe("not_empty")), itoe(-1)),
     Call_Expr(Call_Expr(Self_Expr(), UsedId(stoid(atoe("is_empty")),
     itoe(-1)), Expr_List()), UsedId(stoid(atoe("not")), itoe(-1)),
     Expr_List()))), Condition_List(), Statement_List()));
ce = appback(ce, Routine_Def(FeatureId(Public(), DeclId(stoid(atoe(
     "strlen")), itoe(-1))), Functional(), Argument_List(),
     UsedId(stoid(atoe("INT")), itoe(-1)), Condition_List(),
     Condition_List(), Statement_List()));
ce = appback(ce, Routine_Def(FeatureId(Public(), DeclId(stoid(atoe(
     "is_empty")), itoe(-1))), Functional(), Argument_List(),
     UsedId(stoid(atoe("BOOL")), itoe(-1)), Condition_List(),
     Condition_List(), Statement_List()));
s = appback(s, ValueClass(DeclId(stoid(atoe("STR")), itoe(-1)),
    Typespec_List(), Condition_List(), ce));
/* INOUT */
ce = appback(Class_Element_List(), Routine_Def(FeatureId(Public(),
     DeclId(stoid(atoe("write_int")), itoe(-1))), Not_Functional(),
     appback(Argument_List(), Typed_Ident(DeclId(stoid(atoe("x")),
     itoe(-1)), UsedId(stoid(atoe("INT")), itoe(-1)))), Same_Type(),
     Condition_List(), Condition_List(), Statement_List()));
ce = appback(ce, Routine_Def(FeatureId(Public(), DeclId(stoid(atoe(
     "write_char")), itoe(-1))), Not_Functional(), appback(Argument_List(),
     Typed_Ident(DeclId(stoid(atoe("x")), itoe(-1)), UsedId(stoid(atoe(
```

```
     "CHAR")), itoe(-1)))), Same_Type(), Condition_List(), Condition_List(),
     Statement_List()));
ce = appback(ce, Routine_Def(FeatureId(Public(), DeclId(stoid(atoe(
     "write_string")), itoe(-1))), Not_Functional(), appback(Argument_List(),
     Typed_Ident(DeclId(stoid(atoe("x")), itoe(-1)), UsedId(stoid(atoe(
     "STR")), itoe(-1)))), Same_Type(), Condition_List(), Condition_List(),
     Statement_List()));
ce = appback(ce, Routine_Def(FeatureId(Public(), DeclId(stoid(atoe(
     "read_int")), itoe(-1))), Not_Functional(), Argument_List(),
     UsedId(stoid(atoe("INT")), itoe(-1)), Condition_List(),
     Condition_List(), Statement_List()));
ce = appback(ce, Routine_Def(FeatureId(Public(), DeclId(stoid(atoe(
     "read_char")), itoe(-1))), Not_Functional(), Argument_List(),
     UsedId(stoid(atoe("CHAR")), itoe(-1)), Condition_List(),
     Condition_List(), Statement_List()));
ce = appback(ce, Routine_Def(FeatureId(Public(), DeclId(stoid(atoe(
     "read_string")), itoe(-1))), Not_Functional(), Argument_List(),
     UsedId(stoid(atoe("STR")), itoe(-1)), Condition_List(),
     Condition_List(), Statement_List()));
ce = appback(ce, Routine_Def(FeatureId(Public(), DeclId(stoid(atoe("nl")),
     itoe(-1))), Not_Functional(), Argument_List(), Same_Type(),
     Condition_List(), Condition_List(), Statement_List()));
s = appback(s, RefClass(DeclId(stoid(atoe("INOUT")), itoe(-1)),
    Typespec_List(), Condition_List(), ce));
/* SYS */
ce = appback(Class_Element_List(), Routine_Def(FeatureId(Public(),
     DeclId(stoid(atoe("ob_eq")), itoe(-1))), Functional(), appback(
     appback(Argument_List(), Typed_Ident(DeclId(stoid(atoe("x")),
     itoe(-1)), UsedId(stoid(atoe("$OB")), itoe(-1)))), Typed_Ident(DeclId(
     stoid(atoe("y")), itoe(-1)), UsedId(stoid(atoe("$OB")), itoe(-1)))),
     UsedId(stoid(atoe("BOOL")), itoe(-1)), Condition_List(),
     Condition_List(), Statement_List()));
ce = appback(ce, Routine_Def(FeatureId(Public(), DeclId(stoid(atoe("tp")),
     itoe(-1))), Functional(), appback(Argument_List(), Typed_Ident(DeclId(
     stoid(atoe("x")), itoe(-1)), UsedId(stoid(atoe("$OB")), itoe(-1)))),
     UsedId(stoid(atoe("INT")), itoe(-1)), Condition_List(),
     Condition_List(), Statement_List()));
ce = appback(ce, Routine_Def(FeatureId(Public(), DeclId(stoid(atoe(
     "str_for_tp")), itoe(-1))), Functional(), appback(Argument_List(),
     Typed_Ident(DeclId(stoid(atoe("x")), itoe(-1)), UsedId(stoid(atoe(
     "INT")), itoe(-1)))), UsedId(stoid(atoe("STR")), itoe(-1)),
     Condition_List(), Condition_List(), Statement_List()));
ce = appback(ce, Routine_Def(FeatureId(Public(), DeclId(stoid(atoe("exit")),
     itoe(-1))), Not_Functional(), appback(Argument_List(), Typed_Ident(
     DeclId(stoid(atoe("x")), itoe(-1)), UsedId(stoid(atoe("INT")),
     itoe(-1)))), UsedId(stoid(atoe("INT")), itoe(-1)), Condition_List(),
     Condition_List(), Statement_List()));
s = appback(s, RefClass(DeclId(stoid(atoe("SYS")), itoe(-1)),
    Typespec_List(), Condition_List(), ce));
return s;
}
```

```
/*************************************************************************
 * commom implementation type graph and include graph
 *************************************************************************/

/*
 * Signature:   int is_path(int from, int to, graph *g)
 * Purpose:     tests whether there is a certain path in g
 */
int is_path(int from, int to, graph *g)
{
return *(g->trans + from*g->nodes + to);
}

/*************************************************************************
 * Implementation of type graph
 *************************************************************************/

/*
 * Signature:   Bool is_subtype(Class@ sub, Class@ super)
 * Purpose:     tests whether sub is a subtype of super
 *              According to p. 4 a class is a subtype of itself
 */
ELEMENT is_subtype(ELEMENT sub, ELEMENT super)
{
int von = lookup_node(term(ident(super)), &type_graph);
int nach= lookup_node(term(ident(sub  )), &type_graph);

if (sub != super)
    return btoe(*(type_graph.trans + von*type_graph.nodes + nach));
else
    return btoe(1);
}

/*
 * Signature:   int find_next_abstract(Signature_List array[])
 * Purpose:     returns next abstract class to be processed
 */
int find_next_abstract(ELEMENT array[])
{
int i;

for (i=0; i<type_graph.nodes; i++)
    if (array[i] == nil())
        if (*(etoa(idtos(*(type_graph.class_name + i)))) == '$')
            return i;   /* We found an abstract type */

return -1;
}
```

```
/*
 * Signature:   char signature_conflict(Signature s1, Signature s2)
 * Purpose:     tests whether two signatures conflict
 */
char signature_conflict(ELEMENT s1, ELEMENT s2)
{
int i;
ELEMENT al1 = t_argument_list(s1);      /* Argument_List */
ELEMENT al2 = t_argument_list(s2);      /* Argument_List */
ELEMENT t1, t2;                         /* Typespec      */

if (t_ident(s1) != t_ident(s2)  ||
    numsubterms(al1) != numsubterms(al2) )
    return 0;

for(i=1; i<= numsubterms(al1); i++)
    {
    t1 = t_typespec(subterm(i, al1));
    t2 = t_typespec(subterm(i, al2));
    if (sort(t1) == _Same_Type || sort(t2) == _Same_Type)
        return 1;   /* SAME is not allowed in abstract types */
    if (*(etoa(idtos(t_ident(t1)))) != '$' &&   /* according to p. 4 */
        *(etoa(idtos(t_ident(t2)))) != '$' &&
        t_ident(t1) != t_ident(t2)          )
        return 0;
    }
return 1;
}

/*
 * Signature:   char signature_eq(Signature s1, Signature s2)
 * Purpose:     tests whether two signatures are equal
 */
char signature_eq(ELEMENT s1, ELEMENT s2)
{
int i;
ELEMENT al1 = t_argument_list(s1);      /* Argument_List */
ELEMENT al2 = t_argument_list(s2);      /* Argument_List */
ELEMENT t1, t2;                         /* Typespec      */
ELEMENT r1  = t_typespec(s1);           /* Return Type   */
ELEMENT r2  = t_typespec(s2);           /* Return Type   */

/* Equality of name and number of parameters    */
if (t_ident(s1) != t_ident(s2)  ||
    numsubterms(al1) != numsubterms(al2))
    return 0;

/* Equality of return type                      */
if (sort(r1) == _Same_Type || sort(r2) == _Same_Type)
    return 0;   /* SAME is not allowed in abstract types */
```

```
if (t_ident(r1) != t_ident(r2) )
    return 0;

/* Equality of parameter lists                  */
for(i=1; i<= numsubterms(al1); i++)
    {
    t1 = t_typespec(subterm(i, al1));
    t2 = t_typespec(subterm(i, al2));
    if (sort(t1) == _Same_Type || sort(t2) == _Same_Type)
        return 0;   /* SAME is not allowed in abstract types */
    if (t_ident(t1) != t_ident(t2) )
        return 0;
    }
return 1;
}

/*
 * Signature:   Signature_List remove_overridden(
                Signature_List subdef, Signature_List n)
 * Purpose:     removes inherited signatures that are overridden
 */
ELEMENT remove_overridden(ELEMENT subdef, ELEMENT n)
{
int j;

if (n == Signature_List() || subdef == Signature_List())
    return n;

for (j=1; j<= numsubterms(subdef); j++)
    if (signature_conflict(t_fst(n), subterm(j, subdef)))
        return remove_overridden(subdef, back(n));

return appfront(t_fst(n), remove_overridden(subdef, back(n)));
}

/*
 * Signature:   Signature_List remove_equal_sigs(Signature_List s)
 * Purpose:     removes equal signatures (in case of multiple inheritance)
 */
ELEMENT remove_equal_sigs(ELEMENT s)
{
int i;

if (s == Signature_List())
    return s;

if(s == nil())
    exit(EXIT_FAILURE);

for (i=2; i<=numsubterms(s); i++)
    if (signature_conflict(t_fst(s), subterm(i, s)))
```

```
        if (signature_eq(t_fst(s), subterm(i, s)))
            return remove_equal_sigs(back(s));

return appfront(t_fst(s), remove_equal_sigs(back(s)));
}

/*
 * Signature:   void build_sig_list(int, Signature_List *, Sourcefile)
 * Purpose:     builds the signature list of an abstract type by unifying the
 *              signature lists of all supertypes
 */
void build_sig_list(int sub, ELEMENT array[], ELEMENT t)
{
int super;
int i;
ELEMENT subdef; /* Signature_List */

array[sub] = Signature_List();

/* We build the unification of the signature lists of all supertypes of sub */

for (super=0; super<type_graph.nodes; super++)
    if (*(type_graph.matrix + type_graph.nodes * super + sub))
        if (*etoa(idtos(*(type_graph.class_name + super))) == '$')
            {
            if (array[super] == nil())
                build_sig_list(super, array, t);

            array[sub] = conc(array[sub], array[super]);
            }

/* Now we check wether any explicit declaration overrides an inherited */

for (i= 1; i <= numsubterms(t); i++)
    if (sort(subterm(i, t)) == _AbstractClass &&
        t_ident(subterm(i, t)) == *(type_graph.class_name + sub))
        {
        subdef = t_signature_list(subterm(i, t));
        break;
        }

array[sub] = remove_overridden(subdef, array[sub]);

/* Now we have to test wether the inherited Signature_List
   contains any conflicting signatures                      */

array[sub] = remove_equal_sigs(array[sub]);
array[sub] = conc(subdef, array[sub]);
}
```

```
/*
 * Signature:   Sourcefile build_term(Sourcefile t, Signature_List array[])
 * Purpose:     replaces the literal signature lists of an abstract class
 *              by the constructed list (including inherited signatures)
 */
ELEMENT build_term(ELEMENT t, ELEMENT array[])
{
ELEMENT e;      /* Class@ */

if (t == Sourcefile())
    return t;

e = t_fst(t);

if (sort(e) == _AbstractClass)
    return appfront(AbstractClass(subterm(1, e), subterm(2, e),
           subterm(3, e), subterm(4, e), array[lookup_node(t_ident(e),
           &type_graph)]), build_term(back(t), array));
else
    return appfront(e, build_term(back(t), array));
}

/*
 * Signature:   Sourcefile resolve_subtyping(Sourcefile t)
 * Purpose:     supplement the signature lists of abstract classes
 *              by the signatures of its supertypes
 */
ELEMENT resolve_subtyping(ELEMENT t)
{
int i;
int next;       /* Index of the next abstract type to be processed */

ELEMENT *subtype_array;         /* Ident */

subtype_array = (ELEMENT *)malloc(type_graph.nodes * sizeof(ELEMENT));
if (subtype_array == NULL)
    {
    fprintf(stderr, "Not enough memory\n");
    exit(EXIT_FAILURE);
    }

for (i = 0; i<type_graph.nodes; i++)
    *(subtype_array+i) = nil();

while ((next = find_next_abstract(subtype_array)) != -1)
    build_sig_list(next, subtype_array, t);

return build_term(t, subtype_array);
}
```

```
/*************************************************************************
 * Implementation of include graph
 *************************************************************************/

/*
 * Signature:   int find_next_include(int array[])
 * Purpose:     returns id of next included class to be processed
 */
int find_next_include(int *array)
{
int i;

for (i=0; i<include_graph.nodes; i++)
    if (*(array + i) == 0)
        return i;

/* This statement shoult never be reached! */
fprintf(stderr, "### Include clauses not resolvable\n");
exit(EXIT_FAILURE);
}

/*
 * Signature:   FeatureId new_name(
 *              Feature_modifier_list, Access_Mode, FeatureId)
 * Purpose:     returns name of a feature with regard to feature modifiers
 */
ELEMENT new_name(ELEMENT fm_list, ELEMENT access, ELEMENT id)
{
if (fm_list == Feature_Modifier_List())         /* Unchanged */
    if (access == Public())
        return id;
    else
        return FeatureId(Private(), t_snd(id));

if (sort(t_fst(fm_list)) == _UsedId)
    {
    if (t_ident(t_fst(fm_list)) == t_ident(id))
        return nil();   /* Feature removed */
    }
else                            /* Mod_Rename */
    {
    if (t_ident(t_fst(fm_list)) == t_ident(id))
        return t_featureid(t_fst(fm_list));
    }

return new_name(back(fm_list), access, id);
}
```

```
/*
 * Signature:   Class insert_class_elements(Class, Class)
 * Purpose:     copies class elements from included class to including class
 */
ELEMENT insert_class_elements(ELEMENT from, ELEMENT to)
{
ELEMENT new_list = Class_Element_List();
ELEMENT old_list = t_class_element_list(to);    /* Class_Element_List   */
ELEMENT from_list = t_class_element_list(from); /* Class_Element_List   */
ELEMENT feature;                                /* FeatureId            */
ELEMENT cur_elem;                               /* Class_Element        */
ELEMENT fun;                                    /* Function_Mode        */

while (sort(t_fst(old_list)) != _Include_Clause ||
       t_ident(t_typespec(t_fst(old_list))) != t_ident(from))
    {
    new_list = appback(new_list, t_fst(old_list));
    old_list = back(old_list);
    }

/* t_fst(old_list) is now the include clause we wanted. The Rest of the
   Class_Element_List is appended to new_list,
   the include clause is removed!                                       */
new_list = conc(new_list, back(old_list));

while (from_list != Class_Element_List())
    {
    cur_elem = t_fst(from_list);
    feature = new_name(t_feature_modifier_list(t_fst(old_list)),
              t_access_mode(t_fst(old_list)), t_featureid(cur_elem));

    if (feature != nil())
        {
        if (feature == t_featureid(cur_elem))
            new_list = appback(new_list, cur_elem);
        else
            {
            switch (sort(cur_elem))
                {
                case _Attr_Def:
                    new_list = appback(new_list, Attr_Def(feature,
                               subterm(2, cur_elem)));
                    /* An Attribute is allways followed by its
                       reader and writer                                */
                    /* Process reader */
                    from_list = back(from_list);
                    cur_elem = t_fst(from_list);
                    if (t_access_mode(feature) == Private())
                        new_list = appback(new_list, Routine_Def(FeatureId(
                        Private(), subterm(2, feature)), subterm(2,
                        cur_elem), subterm(3, cur_elem), subterm(4,
```

```
                    cur_elem), subterm(5, cur_elem), subterm(6,
                    cur_elem), subterm(7, cur_elem)));
                else
                    new_list = appback(new_list, Routine_Def(FeatureId(
                    Public(), subterm(2, feature)), subterm(2,
                    cur_elem), subterm(3, cur_elem), subterm(4,
                    cur_elem), subterm(5, cur_elem), subterm(6,
                    cur_elem), subterm(7, cur_elem)));

                /* Process writer */
                from_list = back(from_list);
                cur_elem = t_fst(from_list);
                if (is(to, _ValueClass))
                    fun = Functional();
                else
                    fun = Not_Functional();

                if (t_access_mode(feature) == Public())
                    new_list = appback(new_list, Routine_Def(FeatureId(
                    Public(), subterm(2, feature)), fun, subterm(3,
                    cur_elem), subterm(4, cur_elem), subterm(5,
                    cur_elem), subterm(6, cur_elem), subterm(7,
                    cur_elem)));
                else
                    new_list = appback(new_list, Routine_Def(FeatureId(
                    Private(), subterm(2, feature)), fun, subterm(3,
                    cur_elem), subterm(4, cur_elem), subterm(5,
                    cur_elem), subterm(6, cur_elem), subterm(7,
                    cur_elem)));
                break;
            case _Routine_Def:
                new_list = appback(new_list, Routine_Def(feature,
                subterm(2, cur_elem), subterm(3, cur_elem), subterm(4,
                cur_elem), subterm(5, cur_elem), subterm(6, cur_elem),
                subterm(7, cur_elem)));
                break;
            default:
                fprintf(stderr, "### LINE %d: Illegal class element in
                    class %s\n", etoi(t_line(t_declid(cur_elem))),
                    etoa(idtos(t_ident(from))));
                exit (EXIT_FAILURE);
            }
        }
    }
    from_list = back(from_list);
    }
if (sort(to) == _RefClass)
  return RefClass(subterm(1, to), subterm(2, to), subterm(3, to), new_list);
else
  return ValueClass(subterm(1, to), subterm(2, to), subterm(3, to), new_list);
}
```

```
/*
 * Signature:   Source_file copy_included_code(Source_file, Ident, Ident)
 * Purpose:     searches relevant classes and starts copy
 */
ELEMENT copy_included_code(ELEMENT t, ELEMENT from, ELEMENT to)
{
int i;

if (t_ident(t_fst(t)) == to)
    {
    for (i = numsubterms(t); t_ident(subterm(i, t)) != from; i--)
        ;

    return appfront(insert_class_elements(subterm(i, t), t_fst(t)), back(t));
    }
else
    if (t_ident(t_fst(t)) == from)  /* Source has to be behind target */
        return copy_included_code(appback(back(t), t_fst(t)), from, to);
    else
        return appfront(t_fst(t), copy_included_code(back(t), from, to));
}

/*
 * Signature:   Source_file resolve_includes(Source_file t)
 * Purpose:     supplement the feature lists of classes
 *              by the fearures of included classes
 */
ELEMENT resolve_includes(ELEMENT t)
{
int i, j;
int summe = 0;
int next;

int *include_array = (int *)calloc(include_graph.nodes, sizeof(int));
if (include_array == NULL)
    {
    fprintf(stderr, "Not enough memory!\n");
    exit (EXIT_FAILURE);
    }

for (i = 0; i<include_graph.nodes; i++)
    for (j = 0; j<include_graph.nodes; j++)
        if (is_path(i,j, &include_graph))
            {
            (*(include_array + i))++;
            summe++;
            }

while (summe != -include_graph.nodes)
    {
    next = find_next_include(include_array);
```

```
    for (i = 0; i<include_graph.nodes; i++)
        if (is_path(i, next, &include_graph))
            {
            if (etob(predecl(*(include_graph.class_name + next))))
                {
                fprintf(stderr, "### CLASS %s: Predefined classes can't be
                    included by any other classes\n",
                    etoa(idtos(*(include_graph.class_name + i))));
                exit (EXIT_FAILURE);
                }

            t = copy_included_code(t, *(include_graph.class_name + next),
                *(include_graph.class_name + i));
            *(include_graph.matrix + i*include_graph.nodes + next) = 0;
            (*(include_array + i))--;
            summe--;
            }
    *(include_array + next) = -1;
    summe--;
    }

return t;
}
```

B.2.5 Das Makefile

```
#*************************************************************************
#* Module:              Makefile                                         *
#* Author:              Peter Müller                                     *
#* Description:         Makefile for Sather/SALSA interpreter            *
#*************************************************************************

include MACHINE.flg

CC              = c89
INCLUDES        =
DEBUG           =
LOC_DEFINES     = -DNO_BROWSER
CFLAGS          = $(INCLUDES) $(DEFINES) $(LOC_DEFINES) $(DEBUG)
YACC            = yacc -d
LEX             = flex
LANG            = SALSA
MAXOPTS         = -nowarn

all:    gen $(LANG)

gen:
    mkdir gen

$(LANG) : gen/y.tab.o gen/lex.yy.o gen/$(LANG).o gen/$(LANG)_spec.o
gen/common.o
    $(CC) $(CFLAGS) -o $(LANG) gen/y.tab.o gen/lex.yy.o $(MAXLIB)
          gen/$(LANG).o gen/$(LANG)_spec.o gen/common.o

gen/y.tab.o : gen/y.tab.c gen/lex.yy.c
    (cd gen; $(CC) $(INCLUDES) $(CFLAGS) -c y.tab.c )

gen/y.tab.c gen/y.tab.h: $(LANG)_pars.y gen/$(LANG)_spec.h
    cp $(LANG)_pars.y gen/
    (cd gen; $(YACC) $(LANG)_pars.y)

gen/lex.yy.o: gen/lex.yy.c
    (cd gen; $(CC) $(INCLUDES) $(CFLAGS) -c lex.yy.c )

gen/lex.yy.c: $(LANG)_scan.l gen/y.tab.h
    cp $(LANG)_scan.l gen/
    (cd gen; $(LEX) -L $(LANG)_scan.l )

gen/$(LANG)_spec.o :  gen/$(LANG)_spec.c
    (cd gen; $(CC) $(CFLAGS) -c $(LANG)_spec.c )

gen/$(LANG)_spec.h gen/$(LANG)_spec.c gen/$(LANG)_spec.g : $(LANG)_spec.m
    cp $(LANG)_spec.m gen/
    (cd gen; $(MAX) $(MAXOPTS) $(LANG)_spec.m )
```

```
gen/$(LANG).o :  $(LANG).c  gen/$(LANG)_spec.h common.h
    cp $(LANG).c gen/
    cp common.h gen/
    (cd gen; $(CC) $(CFLAGS) -c $(LANG).c )

gen/common.o: common.c common.h gen/$(LANG)_spec.h
    cp common.c gen/
    cp common.h gen/
    (cd gen; $(CC) $(CFLAGS) -c common.c )

clean :
    rm -rf gen/ core $(LANG)

realclean :
    make clean

install :
    make all; cp $(LANG) ../run
```

Anhang C

Die Implementierung von AutoDoc

C.1 Die MAX-Spezifikation

```
//*************************************************************************
//  Module:              AutoDoc_spec.m                                   *
//  Author:              Peter Müller                                     *
//  Description:         MAX specification of AutoDoc                     *
//*************************************************************************

//*************************************************************************
// Abstract Syntax of Sather-subset
//*************************************************************************

// Classes
Program             ( Sourcefile Statement_List )
Sourcefile          * Class
Class               = AbstractClass | RefClass | ValueClass
AbstractClass       ( DeclId Typespec_List Typespec_List Condition_List
      Signature_List)
RefClass            ( DeclId Typespec_List Condition_List Class_Element_List)
ValueClass          ( DeclId Typespec_List Condition_List Class_Element_List)

// Idents
DeclId              ( Ident Int )
UsedId              ( Ident Int )
FeatureId           ( Access_Mode DeclId )

Access_Mode         = Private | Public | Readonly
Private             ()
Public              ()
Readonly            ()
```

```
// Types
Typespec_List       * Typespec
Typespec            = UsedId | Same_Type
Same_Type           ()

// Class elements
Class_Element_List  * Class_Element
Class_Element       = Attr_Def | Routine_Def | Include_Clause

Signature_List      * Signature
Signature           ( DeclId Function_Mode Argument_List Typespec
      Condition_List Condition_List )

Argument_List       * Typed_Ident
Typed_Ident         ( DeclId Typespec )

Attr_Def            ( FeatureId Typespec )
Routine_Def         ( FeatureId Function_Mode Argument_List Typespec
      Condition_List Condition_List Statement_List )

Function_Mode       = Functional | Not_Functional
Functional          ()
Not_Functional      ()

// Include clauses are removed before the syntax tree is built
// They are only used by the parser
Include_Clause      ( Access_Mode Typespec Feature_Modifier_List )
Feature_Modifier_List   * Feature_Modifier
Feature_Modifier    = Mod_Rename | UsedId
Mod_Rename          ( UsedId FeatureId )

// Conditions
Condition_List      * Condition
Condition           ( DeclId Expr )

// Statements
Statement_List      * Statement
Statement           = Typed_Ident | Assign_Stmt | If_Stmt | Call_Expr
    | Typecase_Stmt | Loop | Return_Stmt | While_Stmt
    | Break_Stmt | Assert_Stmt

Assign_Stmt         ( Local_Expr Expr )
If_Stmt             ( Expr Statement_List Statement_List )
Typecase_Stmt       ( Local_Expr Alternative_List Statement_List )
Alternative_List    * Alternative
Alternative         ( Typespec Statement_List )
Loop                ( Statement_List )
Return_Stmt         ( Expr )
While_Stmt          ( Expr )
Break_Stmt          ()
Assert_Stmt         ( Condition_List )
```

```
// Expressions
Expr                = Self_Expr | Local_Expr | Call_Expr | Void_Expr
    | New_Expr | Logic_Expr | Void_Test_Expr | Literal_Expr
    | Attr_Expr | Set_Expr | Initial_Expr | Result_Expr

Self_Expr           ()
Local_Expr          ( UsedId )
Call_Expr           ( Receiver UsedId Expr_List )
Receiver            = Expr | Typespec
Expr_List           * Expr
Void_Expr           ()
New_Expr            ()
Logic_Expr          ( Expr Operator Expr )
Operator            = And_Op | Or_Op
And_Op              ()
Or_Op               ()
Void_Test_Expr      ( Expr )
Literal_Expr        = Bool | Char | String | Int
Attr_Expr           ()
Set_Expr            ( Local_Expr )
Initial_Expr        ( Expr )
Result_Expr         ()

//*************************************************************************
// Data types for the parser
//*************************************************************************

DeclId_List         * DeclId
Variante            ( Expr Statement_List )
Variante_List       * Variante

//*************************************************************************
// Data types for static analysis
//*************************************************************************

Rout_or_Sig         = Routine_Def@ | Signature@

//*************************************************************************
// Attributes and functions for static analysis
//*************************************************************************

STRUC auto [Sourcefile]{

//*************************************************************************
// Functions defined in C
//*************************************************************************
FCT linetos( Int ) String
FCT print_class( Class@ ) Int
```

```
//*************************************************************************
// Selectors for nodes
//*************************************************************************

FCT ident(Node N) Ident@:
    IF is[N, _DeclId@]:                                 son(1, N)
     | is[N, _UsedId@]:                                 son(1, N)
     | is[N, _FeatureId@]:                              son(2, N).ident
     | is[N, _Signature@]:                              son(1, N).ident
     | is[N, _Typed_Ident@]:                            son(1, N).ident
     | is[N, _Condition@]:                              son(1, N).ident
     | is[N, _Attr_Def@]:                               son(1, N).ident
     | is[N, _Routine_Def@]:                            son(1, N).ident
     | is[N, _Class@]:                                  son(1, N).ident
     | is[N, _Call_Expr@]:                              son(2, N).ident
     | is[N, _Local_Expr@]:                             son(1, N).ident
    ELSE                                                nil()
FCT declid(Node N) DeclId@:
    IF is[N, _FeatureId@]:                              son(2, N)
    ELSE                                                son(1, N)
FCT usedid(Node N) UsedId@:                             son(1, N)
FCT featureid(Node N) FeatureId@:                       son(1, N)
FCT access_mode(FeatureId@ F) Access_Mode@:             son(1, F)
FCT line(Node N) Int@:                                  son(2, N)
FCT subtyping(Class@ C) Typespec_List@:                 son(2, C)
FCT supertyping(AbstractClass@ A) Typespec_List@:       son(3, A)
FCT invariant(Class@ C) Condition_List@:
    IF is[C, _Signature@]:                              son(4, C)
    ELSE                                                son(3, C)
FCT signature_list(AbstractClass@ A) Signature_List@:   son(5, A)
FCT class_element_list(Class@ C) Class_Element_List@:   son(4, C)
FCT argument_list(Rout_or_Sig N) Argument_List@:
    IF is[N, _Routine_Def@]:                            son(3, N)
    ELSE                                                son(3, N)
FCT statement_list(Node N) Statement_List@:
    IF is[N, _Routine_Def@]:                            son(7, N)
     | is[N, _Typecase_Stmt@]:                          son(3, N)
     | is[N, _Alternative@]:                            son(2, N)
     | is[N, _Loop@]:                                   son(1, N)
    ELSE                                                nil()
FCT typespec( Node N ) Typespec@:
    IF is[N, _Alternative@]:                            son(1, N)
     | is[N, _Routine_Def@]:                            son(4, N)
     | is[N, _Signature@]:                              son(4, N)
     | is[N, _Typed_Ident@]:                            son(2, N)
     | is[N, _Attr_Def@]:                               son(2, N)
    ELSE                                                nil()
FCT receiver( Call_Expr@ C ) Node:                      son(1, C)
FCT expr_list( Call_Expr@ C ) Expr_List@:               son(3, C)
```

```
FCT pre(Rout_or_Sig N) Condition_List@:
    IF is[N, _Signature@]:                              son(5, N)
    ELSE                                                son(5, N)
FCT post(Rout_or_Sig N) Condition_List@:
    IF is[N, _Signature@]:                              son(6, N)
    ELSE                                                son(6, N)
FCT function_mode(Rout_or_Sig R) Function_Mode@:        son(2, R)
FCT then_list( If_Stmt@ I ) Statement_List@:            son(2, I)
FCT else_list( Node I ) Statement_List@:                son(3, I)
FCT condition_list(Assert_Stmt@ A) Condition_List@:     son(1, A)
FCT local_expr( Node T ) Node:                          son(1, T)
FCT expr(Node N) Expr@:
    IF is[N, _Initial_Expr@]:                           son(1, N)
    ELSE                                                son(2, N)
FCT operator( Logic_Expr@ L ) Operator@:                son(2, L)
FCT alternative_list(Typecase_Stmt@ T) Alternative_List@:
son(2, T)

FCT fst( Node N ) Node:                                 son(1, N)
FCT snd( Node N ) Node:                                 son(2, N)

//*************************************************************************
// Selectors for terms
//*************************************************************************

FCT t_fst( Term N ) Term:                               subterm(1, N)
FCT t_snd( Term N ) Term:                               subterm(2, N)

FCT t_ident(Term N) Ident:
    IF is[N, _DeclId]:                                  subterm(1, N)
     | is[N, _UsedId]:                                  subterm(1, N)
     | is[N, _FeatureId]:                               subterm(2, N).t_ident
     | is[N, _Signature]:                               subterm(1, N).t_ident
     | is[N, _Typed_Ident]:                             subterm(1, N).t_ident
     | is[N, _Condition]:                               subterm(1, N).t_ident
     | is[N, _Attr_Def]:                                subterm(1, N).t_ident
     | is[N, _Routine_Def]:                             subterm(1, N).t_ident
     | is[N, _Class]:                                   subterm(1, N).t_ident
     | is[N, _Call_Expr]:                               subterm(2, N).t_ident
     | is[N, _Local_Expr]:                              subterm(1, N).t_ident
     | is[N, _Mod_Rename]:                              subterm(1, N).t_ident
    ELSE                                                nil()
FCT t_declid(Term N) DeclId:
    IF is[N, _FeatureId]:                               subterm(2, N)
    ELSE                                                subterm(1, N)
FCT t_featureid(Term N) FeatureId:
    IF is[N, _Mod_Rename]:                              subterm(2, N)
    ELSE                                                subterm(1, N)
FCT t_access_mode(Term N) Access_Mode:                  subterm(1, N)
FCT t_line(Term N) Int:                                 subterm(2, N)
```

```
FCT t_typespec(Term N) Typespec:
    IF is[N, _Alternative]:                          subterm(1, N)
     | is[N, _Routine_Def]:                          subterm(4, N)
     | is[N, _Signature]:                            subterm(4, N)
     | is[N, _Typed_Ident]:                          subterm(2, N)
     | is[N, _Attr_Def]:                             subterm(2, N)
     | is[N, _Include_Clause]:                       subterm(2, N)
    ELSE                                             nil()
FCT t_class_element_list(Class C) Class_Element_List:   subterm(4, C)
FCT t_signature_list(AbstractClass A) Signature_List:   subterm(5, A)
FCT t_argument_list(Term N) Argument_List:
    IF is[N, _Routine_Def]:                          subterm(3, N)
    ELSE                                             subterm(3, N)
FCT t_feature_modifier_list(Include_Clause I) Feature_Modifier_List:
subterm(3, I)

//*************************************************************************
// environments
//*************************************************************************

ATT encl_class(Node N) Class@:
    IF is[N, _Class@]:              N
    ELSE encl_class(fath(N))

//*************************************************************************
// Context conditions
//*************************************************************************

//All classes must have different names
CND Sourcefile@<*, <<ID1, No1>, *>, *, <<ID2, No2>, *>, *>   :
       term(ID1) # term(ID2)
| `"### Class or type " idtos(term(ID1)) " declared twice in: "
    linetos(term(No1)) " and " linetos(term(No2)) "\n"'

//Include clauses should be removed by the parser
CND Include_Clause@ I:
    false() = true()
| `"### CLASS " idtos(term(ident(encl_class(I))))
    ": Unresolved include clause detected\n"'

//SAME must not appear in subtyping lists
CND Class@<<ID, No>, <*, S, *>, *> C:
    is[S, _UsedId@]
| `"### " linetos(term(No)) ": SAME must not appear in subtyping lists\n"'

//SAME must not appear in supertyping lists
CND AbstractClass@<<ID, No>, _, <*, S, *>, *> C:
    is[S, _UsedId@]
| `"### " linetos(term(No)) ": SAME must not appear in supertyping lists\n"'
```

```
//*************************************************************************
// functions for searching the declaration of an ident
//*************************************************************************

FCT lookup( String ID, Class@ C ) Class@:
    IF C.ident.term.idtos = ID:     C
    ELSE                            lookup(ID, rbroth(C))

//*************************************************************************
// Context conditions
//*************************************************************************

//Superclass must be declared
CND Class@<<ID, No>, <*, <I, N>, *>, *> C:
    lookup(I.term.idtos, root().fst) # nil()
| `"### " linetos(term(No)) ": Class " I.term.idtos " not declared\n"'

//Subclass must be declared
CND AbstractClass@<<ID, No>, _, <*, <I, N>, *>, *> C:
    lookup(I.term.idtos, root().fst) # nil()
| `"### " linetos(term(No)) ": Class " I.term.idtos " not declared\n"'

//*************************************************************************
// Output functions
//*************************************************************************

FCT print_all(Class@ C, Int pre) Int:
    LET D == IF pre = 1 || predecl(C.ident.term) = false(): print_class(C)
ELSE 0:
    IF C.rbroth # nil():    print_all(C.rbroth, pre)
    ELSE                    0

FCT print_source( Int pre ) Int:
    print_all(root().fst, pre)

FCT print_sub(Typespec@ T) Int:
    LET C == lookup(T.fst.term.idtos, root().fst):
    LET D == print_class(C):
    IF T.rbroth # nil():    print_sub(T.rbroth)
    ELSE                    0

FCT print_single( String N, Int super ) Int:
    LET C == lookup(N, root().fst):
    IF C # nil():
LET D == print_class(C):
LET SUB == C.subtyping.fst:
IF super = 1 && SUB # nil():
    print_sub(SUB)
ELSE 0
    ELSE 0
```

```
//*************************************************************************
// Check for built-in classes
//*************************************************************************

FCT predecl(Ident ID) Bool:
    LET S == idtos(ID):
    IF S = "BOOL"
    || S = "CHAR"
    || S = "$OB"
    || S = "INT"
    || S = "STR"
    || S = "SYS"
    || S = "INOUT": true()
    ELSE            false()
}
```

C.2 Die C-Funktionen

```
/*************************************************************************
 * Module:              AutoDoc.c                                        *
 * Author:              Peter Müller                                     *
 * Description:         C functions of Autodoc                           *
 *************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "AutoDoc_spec.h"
#include "common.h"

/*************************************************************************
 * global variables
 *************************************************************************/

char all_flag = 'N';
int supertypes_flag = 0;
int class_flag = 0;
int pre_flag = 0;
int output_flag = 0;
FILE *out;

/*************************************************************************
 * function proto types
 *************************************************************************/

void print_statement_list(ELEMENT list, int tab);
void print_expr(ELEMENT e);

/*************************************************************************
 * main and related functions
 *************************************************************************/

/*
 * Signature:   void usage(void)
 * Purpose:     prints command line options and exits
 */
void usage(void)
{
printf("Usage: s [-a] [-c class] [-s] [-p] [-o file] source[.sa]\n");
printf("Options: -a        pretty printer\n");
printf("         -c class  print class \n");
printf("         -s        print sypertypes (only with -c)\n");
printf("         -p        print predefined classes\n");
printf("         -o file   print to file \n");

exit(EXIT_FAILURE);
}
```

```
/*
 * Signature:   int main(int argc, char *argv[])
 * Purpose:     reads, analyses and prints source file
 */
int main(int argc, char *argv[])
{
ELEMENT elem;           /* Sourcefile */
int act_arg = 1;

while(act_arg < argc && argv[act_arg][0] == '-')
    {
    switch(argv[act_arg][1])
        {
        case 'a':
            all_flag = 'Y';
            break;
        case 'p':
            pre_flag = 1;
            break;
        case 'c':
            act_arg++;
            if (act_arg >= argc)
                {
                fprintf(stderr, "Illegal use of option -c\n");
                usage();
                }

            class_flag = act_arg;
            break;
        case 's':
            supertypes_flag = 1;
            break;
        case 'o':
            act_arg++;
            if (act_arg >= argc)
                {
                fprintf(stderr, "Illegal use of option -o\n");
                usage();
                }

            output_flag = act_arg;
            break;
        default:
            fprintf(stderr,"Illegal command line option %s\n",argv[act_arg]);
            usage();
        }
    act_arg++;
    }
```

```
if (act_arg >= argc)
    {
    fprintf(stderr, "Missing filename\n");
    usage();
    }

if (output_flag)
    {
    if( !( out = fopen(argv[output_flag],"w") ) )
        {
        fprintf(stderr,"Can't open file %s for output\n",argv[output_flag]);
        return EXIT_FAILURE;
        }
    }
else
    out = stdout;

elem = parse(argv[act_arg++]);

elem = resolve_subtyping(elem);
elem = resolve_includes(elem);

if ( init_auto(elem) )
    {
    if (class_flag)
        print_single( atoe(argv[class_flag]), itoe(supertypes_flag) );
    else
        print_source( itoe(pre_flag) );
    }

printf("+++ finished\n");

return EXIT_SUCCESS;
}

/*************************************************************************
 * output functions
 *************************************************************************/

/*
 * Signature:   void print_tab(int n)
 * Purpose:     prints certain number of tabs as indent
 */
void print_tab(int n)
{
for (; n>0; n--)
    fprintf(out, "\t");
}
```

```
/*
 * Signature:   void print_ident(Ident@ id)
 * Purpose:     prints an ident
 */
void print_ident(ELEMENT id)
{
fprintf(out, "%s", etoa(idtos(term(id))));
}

/*
 * Signature:   void print_subtyping(Typespec_List@ list, char c)
 * Purpose:     prints a subtyping or supertyping clause
 */
void print_subtyping(ELEMENT list, char c)
{
int i;

if (list == Typespec_List())
    return;

for (i=1; i<= numsons(list); i++)
    {
    if (i == 1)
        fprintf(out, " %c ", c);
    else
        fprintf(out, ", ");
    print_ident(fst(son(i, list)));
    }
}

/*
 * Signature:   void print_type(Typespec@ t)
 * Purpose:     prints type specifier
 */
void print_type(ELEMENT t)
{
if (sort(term(t)) == _Same_Type)
    fprintf(out, "SAME");
else
    print_ident(fst(t));
}

/*
 * Signature:   void print_expr_list(Expr_List@ list)
 * Purpose:     prints list of actual parameters
 */
void print_expr_list(ELEMENT list)
{
int i;
if (numsons(list) == 0)
    return;
```

```
fprintf(out, "(");
for (i = 1; i<= numsons(list); i++)
    {
    print_expr(son(i, list));
    if (i < numsons(list))
        fprintf(out, ", ");
    }
fprintf(out, ")");
}

/*
 * Signature:   void print_expr(Expr@ e)
 * Purpose:     prints expression
 */
void print_expr(ELEMENT e)
{
switch (sort(term(e)))
    {
    case _Self_Expr:
        fprintf(out, "self");
        break;
    case _Local_Expr:
        print_ident(ident(e));
        break;
    case _Call_Expr:
        if (is(term(receiver(e)), _Expr))
            {
            print_expr(receiver(e));
            fprintf(out, ".");
            }
        else
            {
            print_type(receiver(e));
            fprintf(out, "::");
            }
        print_ident(fst(snd(e)));
        print_expr_list(expr_list(e));
        break;
    case _Void_Expr:
        fprintf(out, "void");
        break;
    case _New_Expr:
        fprintf(out, "new");
        break;
    case _Logic_Expr:
        fprintf(out, "(");
        print_expr(fst(e));
        if (sort(term(operator(e))) == _And_Op)
            fprintf(out, " and ");
        else
            fprintf(out, " or ");
```

```
        print_expr(son(3, e));
        fprintf(out, ")");
        break;
    case _Void_Test_Expr:
        fprintf(out, "void(");
        print_expr(fst(e));
        fprintf(out, ")");
        break;
    case _Bool:
        if (etob(term(e)))
            fprintf(out, "true");
        else
            fprintf(out, "false");
        break;
    case _Char:
        fprintf(out, "\'%c\'", etoc(term(e)));
        break;
    case _String:
        fprintf(out, "\"%s\"", etoa(term(e)));
        break;
    case _Int:
        fprintf(out, "%d", etoi(term(e)));
        break;
    case _Attr_Expr:
        fprintf(out, "GET");
        break;
    case _Set_Expr:
        fprintf(out, "SET");
        break;
    case _Result_Expr:
        fprintf(out, "result");
        break;
    case _Initial_Expr:
        fprintf(out, "initial(");
        print_expr(fst(e));
        fprintf(out, ")");
        break;
    default:
        fprintf(stderr, "Illegal expression detected\n");
        exit(EXIT_FAILURE);
    }
}
```

```
/*
 * Signature:   void print_alternative_list(Alternative_List@ list, int tab)
 * Purpose:     prints alternatives of a typecase statement
 */
void print_alternative_list(ELEMENT list, int tab)
{
int i;

for (i= 1; i <= numsons(list); i++)
    {
    print_tab(tab);
    fprintf(out, "when ");
    print_type(typespec(son(i, list)));
    fprintf(out, " then\n");
    print_statement_list(snd(son(i, list)), tab + 1);
    }
}

/*
 * Signature:   void print_condition(Condition@ c, int tab)
 * Purpose:     prints condition list
 */
void print_condition(ELEMENT c, int tab)
{
print_tab(tab);
print_ident(ident(c));
fprintf(out, ": ");
print_expr(expr(c));
}

/*
 * Signature:   void print_condition_list(Condition_List@ c, char mode)
 * Purpose:     prints condition list
 */
void print_condition_list(ELEMENT c, char mode)
{
int i;
int tabs;
int n = numsons(c);

if(n == 0)
    return;

switch(mode)
    {
    case 'I':
        fprintf(out, "\ninvariant\n");
        tabs = 1;
        break;
    case 'E':
        fprintf(out, "\n\tensures\n");
```

```
        tabs = 2;
        break;
    case 'R':
        fprintf(out, "\n\trequires\n");
        tabs = 2;
        break;
    case 'A':
        tabs = 0;
        break;
    default:
        fprintf(stderr, "Illegal mode in print_condition_list: %c\n", mode);
    }

for(i=1; i<=n; i++)
    {
    print_condition(son(i, c), tabs);
    if (i < n)
        {
        fprintf(out, ",");
        fprintf(out, "\n");
        }
    }
}

/*
 * Signature:   void print_statement_list(Statement_List@ list, int tab)
 * Purpose:     prints list of statements
 */
void print_statement_list(ELEMENT list, int tab)
{
int i;
ELEMENT a;

for (i= 1; i <= numsons(list); i++)
    {
    print_tab(tab);
    a = son(i, list);
    switch (sort(term(a)))
        {
        case _Typed_Ident:
            print_ident(ident(a));
            fprintf(out, ": ");
            print_type(snd(a));
            break;
        case _Assign_Stmt:
            print_expr(local_expr(a));
            fprintf(out, " := ");
            print_expr(snd(a));
            break;
        case _If_Stmt:
            fprintf(out, "if ");
```

```
        print_expr(fst(a));
        fprintf(out, " then\n");
        print_statement_list(then_list(a), tab + 1);
        print_tab(tab);
        if (numsons(else_list(a)) > 0)
            {
            fprintf(out, "else\n");
            print_statement_list(else_list(a), tab + 1);
            print_tab(tab);
            }
        fprintf(out, "end");
        break;
    case _Call_Expr:
        print_expr(a);
        break;
    case _Typecase_Stmt:
        fprintf(out, "typecase ");
        print_expr(local_expr(a));
        fprintf(out, "\n");
        print_alternative_list(alternative_list(a), tab);
        print_tab(tab);
        fprintf(out, "else\n");
        print_statement_list(else_list(a), tab + 1);
        print_tab(tab);
        fprintf(out, "end");
        break;
    case _Return_Stmt:
        fprintf(out, "return ");
        print_expr(fst(a));
        break;
    case _Loop:
        fprintf(out, "loop\n");
        print_statement_list(fst(a), tab + 1);
        print_tab(tab);
        fprintf(out, "end");
        break;
    case _While_Stmt:
        fprintf(out, "while!(");
        print_expr(fst(a));
        fprintf(out, ")");
        break;
    case _Break_Stmt:
        fprintf(out, "break!");
        break;
    case _Assert_Stmt:
        fprintf(out, "assert ");
        print_condition_list(fst(a), 'A');
        break;
    default:
        fprintf(stderr, "Illegal statement detected\n");
        exit(EXIT_FAILURE);
```

```
        }
    fprintf(out, ";\n");
    }
}

/*
 * Signature:   void print_parlist(Argument_List@ list)
 * Purpose:     prints list of formal parameters
 */
void print_parlist(ELEMENT list)
{
int i;

if (numsons(list) == 0)
    return;

fprintf(out, "(");
for (i = 1; i<= numsons(list); i++)
    {
    print_ident(ident(son(i, list)));
    fprintf(out, ": ");
    print_type(snd(son(i, list)));
    if (i < numsons(list))
        fprintf(out, ", ");
    }
fprintf(out, ")");
}

/*
 * Signature:   void print_signature(Signature@ s)
 * Purpose:     prints signature
 */
void print_signature(ELEMENT s)
{
fprintf(out, "\t");
if(sort(term(function_mode(s))) == _Functional)
    fprintf(out, "functional ");

print_ident(ident(s));
print_parlist(argument_list(s));
fprintf(out, ": ");
print_type(typespec(s));
print_condition_list(pre(s), 'R');
print_condition_list(post(s), 'E');
if (numsons(pre(s)) + numsons(post(s)) == 0)
    fprintf(out, ";\n");
}
```

```
/*
 * Signature:   void print_routine(Routine_Def@ r)
 * Purpose:     prints routine
 */
void print_routine(ELEMENT r)
{
fprintf(out, "\t");
switch (sort(term(fst(featureid(r)))))
    {
    case _Private:
        fprintf(out, "private ");
        break;
    case _Public:
        break;
    case _Readonly:
        break;
    }

if(sort(term(function_mode(r))) == _Functional)
    fprintf(out, "functional ");

print_ident(ident(r));
print_parlist(argument_list(r));
fprintf(out, ": ");
print_type(typespec(r));
print_condition_list(pre(r), 'R');
print_condition_list(post(r), 'E');
if (all_flag == 'Y')
    {
    if (numsons(pre(r)) + numsons(post(r)) == 0)
        fprintf(out, " is\n");
    else
        fprintf(out, "\n\tis\n");
    print_statement_list(statement_list(r), 2);
    fprintf(out, "\tend;\n\n");
    }
else
    fprintf(out, ";\n");
}

/*
 * Signature:   void print_attr(Attr_Def@ a)
 * Purpose:     prints attribute definition
 */
void print_attr(ELEMENT a)
{
fprintf(out, "\t");
switch (sort(term(fst(featureid(a)))))
    {
    case _Private:
        fprintf(out, "private ");
```

```
        break;
    case _Public:
        break;
    case _Readonly:
        fprintf(out, "readonly ");
        break;
    }
fprintf(out, "attr ");
print_ident(ident(a));
fprintf(out, ": ");
print_type(snd(a));
fprintf(out, ";\n");
}

/*
 * Signature:   void print_class_element_list(Class_Element_List@ list)
 * Purpose:     prints list of class elements
 */
void print_class_element_list(ELEMENT list)
{
int i;

for(i = 1; i<= numsons(list); i++)
    {
    switch (sort(term(son(i, list))))
        {
        case _Routine_Def:
            print_routine(son(i, list));
            break;
        case _Attr_Def:
            print_attr(son(i, list));
            i += 2; /* Skip reader and writer */
            break;
        }
    }
}

/*
 * Signature:   void print_signature_list(Signature_List@ list)
 * Purpose:     prints list of signatures
 */
void print_signature_list(ELEMENT list)
{
int i;

for(i = 1; i<= numsons(list); i++)
    print_signature(son(i, list));
}
```

```
/*
 * Signature:   Int print_class(Class@ c)
 * Purpose:     print class node
 */
ELEMENT print_class(ELEMENT c)
{
int i;

switch (sort(term(c)))
    {
    case _AbstractClass:
        fprintf(out, "type ");
        print_ident(ident(c));
        print_subtyping(subtyping(c), '<');
        print_subtyping(supertyping(c), '>');
        if (numsons(invariant(c)) != 0)
            print_condition_list(invariant(c), 'I');
        fprintf(out, " is\n");
        print_signature_list(signature_list(c));
        fprintf(out, "end;\n\n");
        return;
        break;
    case _RefClass:
        fprintf(out, "class ");
        break;
    case _ValueClass:
        fprintf(out, "value class ");
        break;
    }

print_ident(ident(c));
print_subtyping(subtyping(c), '<');
if (numsons(invariant(c)) != 0)
    print_condition_list(invariant(c), 'I');
fprintf(out, " is\n");
print_class_element_list(class_element_list(c));
fprintf(out, "end;\n\n");

return itoe(0);
}
```

C.3 Das Makefile

```
#*************************************************************************
#* Module:              Makefile                                         *
#* Author:              Peter Müller                                     *
#* Description:         Makefile for AutoDoc                             *
#*************************************************************************

include MACHINE.flg

CC              = gcc
INCLUDES        =
DEBUG           =
LOC_DEFINES     = -DNO_BROWSER -DAUTO
CFLAGS          = $(INCLUDES) $(DEFINES) $(LOC_DEFINES) $(DEBUG)
YACC            = yacc -d
LEX             = flex
LANG            = AutoDoc
SALSA           = SALSA
MAXOPTS         = -nowarn
SALSADIR        = ../../SALSA/src

all:    gen $(LANG)

gen:
    mkdir gen

$(LANG) : gen/y.tab.o gen/lex.yy.o gen/$(LANG).o gen/$(LANG)_spec.o
gen/common.o
    $(CC) $(CFLAGS) -o $(LANG) gen/y.tab.o gen/lex.yy.o $(MAXLIB)
        gen/$(LANG).o gen/$(LANG)_spec.o gen/common.o

gen/y.tab.o : gen/y.tab.c gen/lex.yy.c
    (cd gen; $(CC) $(INCLUDES) $(CFLAGS) -c y.tab.c )

gen/y.tab.c gen/y.tab.h: $(SALSADIR)/$(SALSA)_pars.y gen/$(LANG)_spec.h
    cp $(SALSADIR)/$(SALSA)_pars.y gen/$(LANG)_pars.y
    (cd gen; $(YACC) $(LANG)_pars.y)

gen/lex.yy.o: gen/lex.yy.c
    (cd gen; $(CC) $(INCLUDES) $(CFLAGS) -c lex.yy.c )

gen/lex.yy.c: $(SALSADIR)/$(SALSA)_scan.l gen/y.tab.h
    cp $(SALSADIR)/$(SALSA)_scan.l gen/$(LANG)_scan.l
    (cd gen; $(LEX) -L $(LANG)_scan.l )

gen/$(LANG)_spec.o :  gen/$(LANG)_spec.c
    (cd gen; $(CC) $(CFLAGS) -c $(LANG)_spec.c )
```

```
gen/$(LANG)_spec.h gen/$(LANG)_spec.c gen/$(LANG)_spec.g : $(LANG)_spec.m
    cp $(LANG)_spec.m gen/
    (cd gen; $(MAX) $(MAXOPTS) $(LANG)_spec.m )

gen/$(LANG).o :  $(LANG).c  gen/$(LANG)_spec.h $(SALSADIR)/common.h
    cp $(LANG).c gen/
    cp $(SALSADIR)/common.h gen/
    (cd gen; $(CC) $(CFLAGS) -c $(LANG).c )

gen/common.o: $(SALSADIR)/common.c $(SALSADIR)/common.h gen/$(LANG)_spec.h
    cp $(SALSADIR)/common.c gen/
    cp $(SALSADIR)/common.h gen/
    (cd gen; $(CC) $(CFLAGS) -c common.c )

clean :
    rm -rf gen/ core $(LANG)

realclean :
    make clean

install :
    make all; cp $(LANG) ../run
```

Glossar

A

Abschwächung: Der boolesche Ausdruck A ist eine Abschwächung von B, wenn A aus B folgt.

Algebra: (engl. *algebra*) Eine Algebra ist gegeben durch eine Trägermenge A und eine Menge von Funktionen, die Argumente aus A^n erwarten und ein Ergebnis aus A liefern.

Algebra, dynamische: (engl. *evolving algebra*) Verfahren zur Beschreibung der →operationellen Semantik einer Programmiersprache. Die Systemzustände werden durch →Algebren dargestellt. Änderungen des Zustands werden durch Transformationen der Algebren dargestellt.

Annotation: (engl. *assertion*) Logischer Term, der zu einem bestimmten Zeitpunkt der Programmausführung erfüllt sein muß. Überbegriff für →Invariante, →Vorbedingung, →Nachbedingung, →Zusicherung.

Annotationssprache: (engl. *assertion language*) Sprache, die die Anreicherung eines Programms der →Basissprache um →Annotationen ermöglicht.

Anweisungsblock: Folge von Anweisungen, deren Beginn und Ende nicht durch Schlüsselwörter oder Klammern, sondern durch →Zusicherungen festgelegt ist.

Ausnahme: (engl. *exception*) Verletzung einer → Annotation, auf die mit einer speziellen Bearbeitung reagiert werden muß, da ansonsten die → Zuverlässigkeit des Programms beeinträchtigt ist.

Ausnahmebehandlung: (engl. *exception handling*) Reaktion auf eine →Ausnahme zur kontrollierten Weiterführung oder Beendigung des Progamms.

B

Basissprache: Programmiersprache, auf der eine →Annotationssprache aufsetzt.

Binden, dynamisches: (engl. *dynamic binding*). Zuordnung einer →Prozedur zum Aufruf zur Laufzeit.

Binden, statisches: (engl. *static binding*). Zuordnung einer →Prozedur zum Aufruf zur Übersetzungszeit.

C

code-sharing: Gemeinsame Nutzung von Programmcode durch verschiedene Klassen.

D

dynamisch: (engl. *dynamic*) Erst während des Programmablaufs feststellbar.

E

Eiffel: Objektorientierte Programmiersprache mit Annotationsmechanismen (siehe [Mey92b]).

F

Fehler, logischer: (engl. *error*) Inkonsistenz eines Programmstücks mit seiner →Spezifikation.

Funktion, dynamische: (engl. *dynamic function*) Funktion einer →Algebra, deren Werte während der Programmausführung verändert werden können (vgl. →dynamische Algebra).

I

Initialausdruck: (engl. *initial expression*) Ausdruck, der im →Initialzustand ausgewertet, aber erst später verwendet wird.

Initialzustand: (engl. *prestate*) Zustand der Programmausführung beim Betreten einer →Routine. Der Initialzustand liegt nach dem Übertragen der aktuellen Parameter und vor der Ausführung der ersten Anweisung des Rumpfes.

Inkludierungsbestimmung: (engl. *include clause*) →Klassenkomponente zur Einbindung von Programmtext fremder Klassen (→code-sharing).

Invariante: (engl. *invariant*) Logischer Term, der vor und nach der Ausführung eines Programmstücks erfüllt sein muß.

Invariante, Klassen-: (engl. *class invariant*) →Objektinvariante.

Invariante, Objekt-: Logischer Term, der beim Betreten und Verlassen jedes Objekts einer Klasse erfüllt sein muß.

Invariante, Schleifen-: (engl. *loop invariant*) Logischer Term, der vor und nach jeder Ausführung des Schleifenkörpers erfüllt sein muß.

K

Kapselung: (engl. *encapsulation*) Verbergen von Implementierungsdetails einer Klasse. Nur die Spezifikation einer gekapselten → Routine ist nach außen sichtbar.

Klasse, Referenz-: (engl. *reference class*) Klasse, deren Objekte stets über Referenzen angesprochen werden.

Klasse, Value-: (engl. *value class*) Klasse, deren Objekte als Werte angesprochen werden. Bei Zuweisungen oder Verwendung als Parameter wird eine Kopie des Objekts erzeugt.

Klassenkomponente: (engl. *class element*) Bestandteil einer Klassendefinition, z. B. Attribut, →Methode oder →Inkludierungsbestimmung.

Kontrollfluß: (engl. *control flow*) Abfolge der Anweisungen. Der Kontrollfluß regelt die Ausführung der Rechenschritte und den Zugriff aus Daten.

Kontrollflußgraph: Datenstruktur, die den →Kontrollfluß eines Programms wiedergibt.

Korrektheit, logische: (engl. *correctness*) Ein Programm ist logisch korrekt, falls es seine Aufgaben gemäß den gestellten Anforderungen und seiner Spezifikation erfüllt.

L

Larch: Projekt zur Unterstützung der →Verifikation von Programmen durch →Annotationen (siehe [GH93]).

M

Methode: (engl. *method*) Hier synonym gebraucht für →Prozedur und →Routine.

N

Nachbedingung: (engl. *postcondition*) Logischer Term, der nach Ausführung einer → Routine erfüllt sein muß.

P

Polymorphismus: (engl. *polymorphism*) Syntaktisch gleiche Nachrichten an Objekte verschiedener Klassen rufen verschiedene, oft semantisch ähnliche →Methoden auf.

Programmierung, funktionale: Programmierung durch Funktionen, deren Ergebnis lediglich von Parametern, nicht aber vom Systemzustand abhängt (Seiteneffektfreie Programmierung).

Programmierung, imperative: Programmierung durch Formulierung von Algorithmen. Imperative Programme bestehen aus Folgen von Anweisungen.

Programmierung, objektorientierte: (engl. *object-oriented programming*). Objektorientierte Programme bestehen aus Objekten, die durch den Austausch von Nachrichten miteinander kommunizieren. Objekte enthalten sowohl Daten als auch die Operationen, die auf diesen Daten ausgeführt werden können.

Prozedur: (engl. *procedure*) Abgeschlossenes, benanntes Programmstück, evtl. parametrisiert.

R

Robustheit: (engl. *robustness*) Fähigkeit eines Systems, →Ausnahmen zu erkennen und mit geeigneter →Ausnahmebehandlung zu reagieren.

Routine: (engl. *routine*) Hier synonym gebraucht für →Prozedur und →Methode.

S

SALSA: (Abk. für Sather Annotation Language for Specification) Die in dieser Arbeit entwickelte →Annotationssprache für die →Basissprache →Sather.

Sather:
Objektorientierte Programmiersprache, die in dieser Arbeit als →Basissprache dient (siehe [Omo94]).

Semantik: (engl. *semantics*) Inhaltliche Bedeutung einer Sprache.

Semantik, operationelle: (engl. *operational semantics*) Beschreibung der →Semantik durch Angabe eines Verfahrens, wie ein Programm durch eine Folge von Rechenschritten (Operationen) ausgeführt werden kann. Die Interpretation eines Programms geschieht dabei durch eine Folge von Zuständen.

Semantik, dynamische: (engl. *dynamic semantics*) Semantisches Verhalten des Programms zur Laufzeit.

Semantik, Interpreter-: →operationelle Semantik.

Semantik, statische: (engl. *static semantics*) Semantische Eigenschaften des Programms, die zur Übersetzungszeit feststellbar sind.

Speicherbereinigung: (engl. *garbage collection*) Verfahren zur Freigabe belegter, jedoch ungenutzter Speicherbereiche.

Spezifikation: (engl. *specification*) Detaillierte Beschreibung der Teile eines Ganzen und ihrer Eigenschaften.

Spezifikation, algebraische: (engl. *algebraic specification*) →Spezifikation eines Datentyps durch eine →Algebra mit Trägermenge und Operationen. Die Eigenschaften der Operationen werden durch Gleichungen (sog. Gesetze) formuliert.

statisch: (engl. *static*) Zur Übersetzungszeit feststellbar.

Syntax, abstrakte: (engl. *abstract syntax*) Aufbau von Programmen ohne konkrete Notation.

Syntax, konkrete: (engl. *concrete syntax*) Aufbau von Programmen mit konkreter Notation.

T

Terminalzustand: (engl. *poststate*). Zustand der Programmausführung beim Verlassen einer →Routine. Der Terminalzustand liegt nach der Berechnung des Rückgabewerts und vor der Rückkehr zum →Aufrufer.

V

Vererbung: (engl. *inheritance*) Weitergabe von Merkmalen von einem Datenobjekt an ein anderes Objekt.

Verifikation: (engl. *verification*)
Formaler Nachweis von Eigenschaften von Programmen oder Programmteilen. Verifikation dient zum Beweis der →logischen Korrektheit von Programmen.

Verschärfung: Der boolesche Ausdruck A ist eine Verschärfung von B, wenn B aus A folgt.

Vertrag: (engl. *contract*) Formulierung der Bedingungen zur Nutzung eines Programmstücks. Ein Vertrag besteht für beide Parteien aus Vorzügen und Verpflichtungen (vgl. [Mey92a]).

Vorbedingung: (engl. *precondition*) Logischer Term, der vor Ausführung einer →Routine erfüllt sein muß.

Z

Zusicherung: (engl. *assertion*) Logischer Term, der zu einem bestimmten Zeitpunkt während der Ausführung einer → Routine erfüllt sein muß.

Zuverlässigkeit: (engl. *reliability*) Kombination aus → Korrektheit und → Robustheit.

Literaturverzeichnis

[BW81] F. L. Bauer, H. Wössner. *Algorithmische Sprache und Programmentwicklung*. Springer-Verlag, 1981.

[BJ89] J.-C. Bolot, P. Jalote. Formal Verification of Programs with Exceptions. In *The Nineteeth International Symposium on Fault-Tolerant Computing*, S. 283–290, IEEE Computer Society Press, 1989.

[GHW85] J. V. Guttag, J. J. Horning, J. M. Wing. *Larch in Five Easy Pieces*. DEC Systems Research Center Report 5, Palo Alto, 1985.

[GHM90] J. V. Guttag, J. J. Horning, A. Modet. *Report on the Larch Shared Language: Version 2.3*. DEC Systems Research Center Report 58, Palo Alto, 1990.

[GH91] J. V. Guttag, J. J. Horning. A tutorial on Larch and LCL, a Larch/C interface language. In S. Prehn, W. J. Toetenel, Hrsg., *Formal Software Development Methods*, S. 1–78, LNCS 552, Springer Verlag, 1991.

[GH93] J. V. Guttag, J. J. Horning, Hrsg. *Larch: Language and Tools for Formal Specification*. Texts and Monographs in Computer Science. Springer-Verlag, 1993.

[Jon91] K. D. Jones. *LM3: A Larch Interface Language for Modula-3, A Definition and Introduction, Version 1.0*. DEC Systems Research Center Report 72, Palo Alto, 1991.

[Knu69] D. E. Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Addison-Wesley Publishing Company, 1969.

[Mar94] V. Markl. *Ausnahmebehandlung in imperativen, objektorientierten und deduktiven Systemen*. Diplomarbeit, Technische Universität München, 1994.

[Mey92a] B. Meyer. Design by Contract. In D. Mandrioli, B. Meyer, Hrsg., *Advances in object-oriented software engineering*, S. 1–50, Prentice Hall, 1992.

[Mey92b] B. Meyer. *Eiffel: The Language*. Prentice Hall, 1992.

[MOS93] S. Murer, S. Omohundro, C. Szypersky. *Engineering a Programming Language: The Type and Class System of Sather*. Report TR-93-064, International Computer Science Institute, Berkeley, 1993.

[Omo94] S. M. Omohundro. *The Sather 1.0 Specification*. International Computer Science Institute, Berkeley, 1994.

[PH93] A. Poetzsch-Heffter. Programming Language Specification and Prototyping Using the MAX System. In M. Bruynooghe, J. Penjam, Hrsg., *Programming Language Implementation and Logic Programming*, S. 137–150, LNCS 714, Springer Verlag, 1993.

[PHE93] A. Poetzsch-Heffter, T. Eisenbarth. *The MAX System: A Tutorial Introduction*. Technical Report TUM-I9307, Institut für Informatik, Technische Universität München, 1993.

[PH94] A. Poetzsch-Heffter. Developing Efficient Interpreters Based on Formal Language Specifications. In P. A. Fritzson, Hrsg., *Compiler Construction*, S. 233–247, LNCS 786, Springer Verlag, 1994.

Abbildungsverzeichnis

2.1 Gliederung eines Tasks in Teilaufgaben 5
2.2 Implementierung der Fakultätsfunktion 7

3.1 Gliederung der Anweisungen in Blöcke, Prozeduren und Objekte . 13

4.1 Sather Programmbeispiel 33
4.2 Auszug aus der abstrakten Sather-Syntax 35
4.3 Syntaxbaum mit Kontrollflußgraph für `n := n - 1` 36
4.4 Die Tasks des Interpreters 39
4.5 Datenobjekte eines Sather-Programms 40
4.6 Die Speicherplätze von Sather-Variablen 41
4.7 Die dynamischen Funktionen der Algebra 42
4.8 Die Ausdrücke von Sather 43
4.9 Kontrollfluß für Zugriff auf Speicher 44
4.10 Regeln für den Speicherzugriff 44
4.11 Kontrollfluß für Objekterzeugung und -manipulation 46
4.12 Regeln für Attributveränderung 46
4.13 Regeln für Objekterzeugung 47
4.14 Kontrollfluß für logisches *und* und *oder* 48
4.15 Regeln für logisches *und* und *oder* 48
4.16 Kontrollfluß für Methodenaufruf und `return` 49
4.17 Regeln für Methodenauswahl und Parameterübergabe 50
4.18 Regeln für Sather-Methodenaufruf 51
4.19 Regeln für Methodenrückkehr 52

5.1 Die Produktionen für Bedingungen 57
5.2 Kontrollfluß für `result`-Ausdruck 58
5.3 Erkennung von abgeschwächten und verschärften Bedingungen . . 59
5.4 Kontrollfluß für die Überprüfung von Bedingungen 60
5.5 Regeln für `Branch`-Task 61
5.6 Programmbeispiel mit Objektinvariante 63
5.7 Behandlung eines Methodenaufrufs in SALSA 65
5.8 Regeln für SALSA-Methodenaufruf 66
5.9 Programmfortsetzung nach Überprüfung der Annotationen 67

5.10 Regeln für Invariantenbehandlung 68
5.11 Steuerung der Auswertung der Initialausdrücke 68
5.12 Kontrollfluß für Initialausdrücke 69
5.13 Kontrollfluß für `return`-Anweisung 69
5.14 Regeln für SALSA-Methodenrückkehr 70
5.15 Annotierte `create`-Routine 70
5.16 Annotierte `fac`-Routine 71
5.17 Produktionen für Zusicherungen 72
5.18 Verwendung von Zusicherungen 73

6.1 Vererbungs- und Inkludierungsbeziehungen 75
6.2 Algebraische Spezifikation des Datentyps $stack\chi$ 76
6.3 AutoDoc-Dokumentation der `STACK`-Klasse 86

Tabellenverzeichnis

2.1 Vorzüge und Verpflichtungen des Beispielvertrages 7
2.2 Vertrag für den Aufruf der Funktion `fac` 8

4.1 Definition und Verwendung der Tasks 39
4.2 Verwendung der Ausdrücke 43

6.1 Die Klassen des Programmbeispiels 75