$\mathsf{Oberon}_{\mathsf{T}}$ — eine Programmiersprache für sicherheitskritische Systeme

Daniel Schweizer

Institut für Technische Informatik und Kommunikationsnetze
ETH Zürich, Gloriastrasse 35
CH-8092 Zürich, Switzerland

TIK-Report No. 21
Version 1.0

September 1996

Zusammenfassung

Im Rahmen des Forschungprojektes PESCA wurde ein Verfahren zur Verifikation von Programmen entwickelt, die in einer algebraischen Spezifikationssprache beschrieben und mit einer imperativen Programmiersprache implementiert werden. Bevorzugtes Anwendungsgebiet für diesen Ansatz sind abstrakte Datentypen, die in Steuerungen von sicherheitskritischen Systemen eingesetzt werden und deren Laufzeitverhalten mit Sicherheit vorhersehbar sein muss. Als Implementationssprache wird eine Teilmenge der Programmiersprache Oberon verwendet, die wir im folgenden mit $\mathsf{Oberon}_\mathsf{T}$ bezeichen. Der vorliegende Bericht beschreibt die Einschränkungen von $\mathsf{Oberon}_\mathsf{T}$ gegenüber Oberon und definiert die dynamische Semantik der Sprache unter Verwendung algebraischer Spezifikationstechniken.

Die Auswahl der zugelassenen Sprachkonstrukte erfolgt primär unter dem Gesichtspunkt der Sicherheit und in zweiter Linie unter jenem der einfachen Verifizierbarkeit. Ausgeschlossen werden insbesondere Konstrukte, welche aufgrund von Ressourcenbeschränkungen zu Laufzeitfehlern führen können (z.B. dynamische Datenstrukturen, Rekursion) und solche, deren Zeitverhalten nicht im voraus bestimmbar ist (z.B. `WHILE`-Schleife).

Der Bericht dient als Referenz für weitere Arbeiten, in denen $\mathsf{Oberon}_\mathsf{T}$ als Implementationssprache für abstrakte Datentypen verwendet wird.

Inhaltsverzeichnis

Kapitel 1

Einführung

Im Rahmen des Forschungprojektes PESCA [DS96] wurde ein Verfahren zur Verifikation von Programmen entwickelt, die in einer algebraischen Spezifikationssprache beschrieben und mit einer imperativen Programmiersprache implementiert werden [Sch]. Als Beispiele für solche Sprachen dienen die algebraische Spezifikationssprache LSL [GH93] und die im folgenden beschriebene Teilmenge von Oberon [Rei92], die wir mit $\mathsf{Oberon}_\mathsf{T}$ bezeichnen. Die Auswahl der zugelassenen Konstrukte erfolgt primär unter dem Gesichtspunkt der Sicherheit und in zweiter Linie unter jenem der einfachen Verifizierbarkeit.

Die Sprache Oberon ist eine sehr kompakte, wohldefinierte und sorgfältig entworfene imperative Programmiersprache. Sie eignet sich aufgrund ihres Modulkonzepts, ihrer Einfachheit und der Effizienz ihrer Implementation ausgezeichnet für die Programmierung abstrakter Datentypen. Es geht also im folgenden keinesfalls darum, die Sprache Oberon in irgendeiner Weise zu "verbessern"; viel eher möchten wir die universelle Programmiersprache durch Auszeichnung einer Teilmenge so beschränken, dass wir ohne die Verwendung potentiell fehleranfälliger Sprachkonstrukte[1] auch sicherheitskritische technische Systeme realisieren können. Selbstverständlich geben wir damit einen Teil der Mächtigkeit der Sprache zugunsten erhöhter Sicherheit preis.

Es wird vorausgesetzt, dass der Leser zumindest eine algebraische Spezifikationssprache (z.B. LSL [GH93], OBJ [GWMJ92], CIP-L [Par90]) und eine Pascal-ähnliche Programmiersprache (z.B. Pascal, Modula-2, Oberon [Rei92]) kennt. Wir werden im Rahmen dieses Berichtes nur einige ausgewählte Aspekte von Syntax und Semantik der beiden verwendeten Sprachen LSL und Oberon betrachten können.

[1] Wir verstehen darunter z.B. potentiell nicht terminierende Schleifen und Rekursionen sowie dynamische Datenstrukturen, die auf endlichen Speichern implementiert sind.

1.1 Aufbau des Berichts

Kapitel 2 erläutert kurz einige Eigenschaften sicherheitskritischer Systeme und motiviert anhand dieser die Verwendung einer Teilmenge der Sprache Oberon.

Kapitel 3 beschreibt die Einschränkungen von $\mathsf{Oberon}_\mathsf{T}$ gegenüber Oberon.

Kapitel 4 definiert die dynamische Semantik von $\mathsf{Oberon}_\mathsf{T}$.

Kapitel 2

Sicherheitskritische Systeme

Die Charakterisierung sicherheitskritischer Systeme ist ein äusserst schwieriges und im Rahmen dieses Berichtes unmöglich vollständig abzuhandelndes Thema. Trotzdem versuchen wir, einige Eigenschaften solcher Systeme, die im Hinblick auf eine formale Verifikation von Bedeutung sind, zu identifizieren und für unsere Zwecke sinnvoll auszunützen. Für weitergehende Betrachtungen wird [Rus95] empfohlen.

In [How95] finden wir für den Begriff *safety-critical system* folgende Definition:

> A computer, electronic or electromechanical system whose failure may cause injury or death to human beings. Eg. an aircraft or nuclear power station control system. Common tools used in the design of safety-critical systems are redundancy and formal methods.

2.1 Interessierendes Verhalten

Grundsätzlich können wir bei sicherheitskritischen Systemen davon ausgehen, dass in der Regel das Verhalten im schlechtestmöglichen Fall interessiert. Dies gilt sowohl für den Verbrauch endlicher Ressourcen als auch für das zeitliche Verhalten. Aus diesem Grunde ist es z.B. nicht sinnvoll, in der Architektur solcher Systeme stochastische Elemente wie Cache-Speicher oder Prozessoren mit Pipelines zu verwenden. Tatsächlich manifestiert sich diese Erkenntnis bei der Architektur von Rechnern für eingebettete Systeme (z.B. *Transputer* [Inm88]).

Analoge Überlegungen bei der Entwicklung von Programmiersprachen führen dazu, dass wir auf einige nicht inhärent sichere Konstrukte der Sprache Oberon verzichten möchten.

Es erscheint uns z.B. als sinnlos, in einem sicherheitskritischen System — meist unter dem Titel der *Abstraktion*[1] oder einer effizienten Berechnungsmethode — eine primitiv rekursive Funktion (und andere terminieren nicht in jedem

[1] Tatsächlich handelt es sich meist um eine unsorgfältige Analyse des Problems.

Fall) mit einer `WHILE`- oder `REPEAT`-Schleife zu berechnen und anschliessend mit erheblichem Aufwand zu analysieren, ob sie immer terminiert.

In Bezug auf die Endlichkeit der Ressourcen *Speicher* und *Wortlänge* muss darauf hingewiesen werden, dass es nicht sinnvoll ist, diese inhärenten Begrenzungen beim Systementwurf zu vernachlässigen. Die Folgen solcher Vereinfachungen sind komplexe Fehlerbehandlungen und Laufzeitfehler.

Der Verhinderung von Laufzeitfehlern muss in jedem Fall höchste Priorität beigemessen werden. Insbesondere bei Systemen, die keinen *fail-safe* Zustand[2] haben, können sowohl erkannte als auch unerkannte Laufzeitfehler fatale Folgen haben.

Sicherheit und Lebendigkeit Je nach verwendetem Spezifikationsformalismus definiert man verschiedene Formen der Begriffe Sicherheit und Lebendigkeit (siehe z.B. [Sta90, Bau90, Rei86]). Wir beschränken uns hier darauf, die Programmiersprache so einzuschränken, dass folgende Eigenschaften einer Implementation verifizierbar sind:

Unter der Voraussetzung, dass ihre Vorbedingung erfüllt ist, gelten für jede Operation des implementierten ADT folgende Aussagen:

- Für jede Operation kann — abhängig von der ausführenden Maschine — eine maximale und eine minimale Zeitspanne für deren Abarbeitung berechnet werden[3].
- Jede Operation genügt den Axiomen und Theoremen der algebraischen Spezifikation des ADT.
- Für jede Operation kann — abhängig von der ausführenden Maschine — der maximale Speicherbedarf berechnet werden.

2.2 Konsequenzen für die Programmiersprache

Bevor wir festlegen, auf welche Oberon-Sprachkonstrukte verzichtet werden soll, zitieren wir eine Definition des Begriffs *safety-critical software language* aus [TM92]:

> ... in software engineering (programming languages), a high-level language or subset of the language that (1) has a formal syntax with well-defined semantics, (2) is block structured and strongly typed, and

[2] *fail-safe state*: A state of the system that cannot result in an accident, although some other important goals of the system, such as availability, may be compromised [MoD91, TM92].

[3] Wir gehen in einer ersten Betrachtung von einer defensiven Berechnung aus, d.h. bei Verzweigungen wir immer der kürzeste (längste) Pfad gewählt, um die minimale (maximale) Ausführungszeit zu berechnen. Durch eine Pfadanalyse wäre es möglich, genauere Angaben zu machen. Unsere Berechnungen sind aber in jedem Fall sicher.

(3) is analyzable using static code analysis tools. Safety-critical software languages prohibit the use of practices that are unsafe or difficult to analyze, such as floating-point arithmetic, recursion (whether simple or mutual), and object code patching. Also excluded are anomalies of flow control such as unintended loops, unreachable code, loops with multiple entries, and variables being used before being set or set and not used.

Kapitel 3

Einschränkungen der Sprache Oberon

3.1 Kontrollfluss

Schleifen und Rekursion Um auf einfache Art und Weise garantieren zu können, dass alle Operationen des ADT in jedem Fall terminieren, schliessen wir das `WHILE ... DO ... END`-, das `REPEAT ... UNTIL ...`- und das `LOOP ... EXIT ... END`-Konstrukt sowie *Rekursionen* aus. Ausserdem muss die Laufvariable der `FOR`-Schleife bei jedem Schleifendurchlauf inkrementiert werden, womit auch das Konstrukt `... BY ...` entfällt.

Direkte Rekursion wird dadurch ausgeschlossen, dass Prozeduren erst am Ende ihres *procedure-body* als deklariert gelten. Um indirekte Rekursion (*mutual recursion*) auszuschliessen, verzichten wir auf die *forward declaration*. Da zyklische Importe ohnehin ausgeschlossen sind, entstehen auch dadurch keine Probleme.

Dieser statischen Einschränkung stehen flexiblere, aber sehr aufwendige Methoden gegenüber, die durch Analyse eines Programms oder einer Spezifikation (z.B. [Gri81, Sta90]) versuchen, Aussagen über verschiedene Formen von Lebendigkeit zu machen. Diesen komplexeren Ansätzen ist in der Praxis bisher wenig Erfolg beschieden. Zudem garantiert eine Analyse der Spezifikation ohne anschliessende Verifikation der Implementation keineswegs, dass das implementierte System den Anforderungen genügt. Eine sichere Lösung wäre die automatische Codegenerierung. In [Hoa87] wird aber gezeigt, dass aus einer hinreichend abstrakten Spezifikation im allgemeinen nur ineffizienter Code erzeugt werden kann. Eine weitere mögliche Lösung dieses Problems bietet die schrittweise Implementation durch semantikerhaltende Transformationen [Par90] an. Leider ist auch dieses Verfahren sehr aufwendig und wird hier nicht weiter untersucht.

`CASE`-Anweisungen gibt es in Oberon$_T$ nicht, weil das Konstrukt nicht unbedingt benötigt wird und zudem als Operator mit beliebig vielen Argumenten algebraisch nur schwer beschreibbar ist[1].

RETURN-Anweisungen müssen als *letzte Anweisung* eines Prozedurrumpfs auftreten.

3.2 Speicherverwaltung

In den zur Diskussion stehenden Systemen sind Ausnahmesituationen, die allein durch Unzulänglichkeiten der Steuerung entstehen, wie etwa *stack overflow* oder *memory allocation failed* sowie stochastisches Zeitverhalten bei der Speicherallokation und -freigabe unerwünscht. Die Behandlung solcher Ausnahmefälle — sofern überhaupt möglich — kompliziert ein System in unnötiger Art und Weise. Aus diesem Grunde unterziehen wir uns folgenden Beschränkungen:

Allokation von Datenstrukturen Sämtliche Datenstrukturen werden statisch alloziert. Wir verzichten also auf jegliche Art von dynamischen Datenstrukturen. In $\mathsf{Oberon}_\mathsf{T}$ gibt es keinen Datentypen `POINTER`. Ein *heap overflow (memory allocation failed)* sowie Probleme mit dem Zeitverhalten durch Speicherallokation und -freigabe sind somit von vornherein ausgeschlossen. Damit ist auch die Konstante `NIL` unnötig.

Typerweiterung Auf die Typerweiterung (`RECORD`-Erweiterung) wird verzichtet, da diese ohne die Verwendung dynamischer Datenstrukturen und dynamischer Typen (`POINTER`, `VAR`-Parameter) wenig Sinn macht, aber die Spezifikation der Sprach-Semantik erheblich komplizierter würde[2]. Damit sind auch die Sprachkonstrukte `... IS ...` und `WITH ... DO ... END` unnötig.

Stack Trotz Ausschluss von Rekursionen bleibt die Gefahr eines *stack overflows*. Dieses Problem kann aber einfach gelöst werden, indem der maximale Speicherbedarf für den *Stack* statisch berechnet wird. In erster Näherung genügt eine defensive Berechnung des maximalen *Stack*-Speicherbedarfs, bei der man auf eine Analyse von sich gegenseitig ausschliessenden Prozeduraufrufen verzichtet. Eine solche Pfadanalyse könnte sich allenfalls bei Massenprodukten lohnen, wo sich der erhöhte Entwicklungsaufwand durch Einsparungen beim Speicherbedarf rechtfertigen lässt. Die Sicherheit ist aber auch bei einer defensiven Berechnung ohne Pfadanalyse gewährleistet.

[1] Es müsste für jede mögliche Anzahl Argumente ein separater Operator definiert werden.

[2] Dies gilt insbesondere für die Spezifikationssprache LSL, welche keine Subsorten-Relation [GM89, GWMJ92] kennt.

3.3 Seiteneffekte

Globale Variablen In $\mathsf{Oberon}_{\mathsf{T}}$ schliessen wir den Gebrauch von globalen Variablen aus. Diese Einschränkung ist unbedingt nötig, da sonst selbst grundlegendste Gesetze der Algebra, wie z.B. das Kommutativgesetz der *booleschen* Algebra oder der Algebra der ganzen Zahlen nicht gelten würden. Sämtliche Ausdrücke (*expressions*) bleiben damit seiteneffektfrei.

Beispiel 3.1 (Seiteneffekte) Wir betrachten das Modul `Problem` aus Abbildung 3.1. Zur Illustration des Problems bedienen wir uns bereits hier der Technik der algebraischen Semantik, die in Abschnitt 4 vertieft behandelt wird[3].

```
MODULE Problem;

VAR a: BOOLEAN; (* Modulglobale Variable *)

PROCEDURE f():BOOLEAN;
BEGIN
  a := FALSE;
  RETURN TRUE;
END f;

END Problem.
```

Abbildung 3.1: Funktion mit Seiteneffekten

Wenn wir wie in Gleichung 3.1 davon ausgingen, dass der &-Operator von Oberon trotz Zulassung von Funktionen mit Seiteneffekten kommutativ ist, wie dies für den $\wedge$-Operator der *booleschen* Algebra gilt, so müssten auch die beiden Anweisungssequenzen in den Gleichungen 3.2 und 3.3 den gleichen Wert ergeben.

$$\begin{aligned}&\forall \mathsf{b1}, \mathsf{b2} : \mathtt{BOOLEAN}, \mathsf{s} : \mathtt{Store}.\\ &\quad [\![\mathsf{b1} \mathbin{\&} \mathsf{b2}]\!][\mathsf{s}, \mathtt{true}] == [\![\mathsf{b1}]\!][\mathsf{s}, \mathtt{true}] \wedge [\![\mathsf{b2}]\!][\mathsf{s}, \mathtt{true}] \end{aligned} \tag{3.1}$$

$$\forall \mathsf{a} : \mathtt{BOOLEAN}, \mathsf{s} : \mathtt{Store}.$$

$$([\![\mathsf{a} := \mathtt{TRUE}; \mathtt{RETURN}(\mathsf{a} \mathbin{\&} \mathtt{f}())]\!][\mathsf{s}, \mathtt{true}]) : \mathtt{Bool} == \mathtt{true} \tag{3.2}$$

$$([\![\mathsf{a} := \mathtt{TRUE}; \mathtt{RETURN}(\mathtt{f}() \mathbin{\&} \mathsf{a})]\!][\mathsf{s}, \mathtt{true}]) : \mathtt{Bool} == \mathtt{true} \tag{3.3}$$

Offensichtlich widersprechen diese Gleichungen aber der Semantik des realen Oberon-Programms. Im ersten Fall (Gleichung 3.2) ergibt das Programm den

[3]Die Zeichen $[\![\_\_]\!]\_\_$ sind Bezeichner für Semantikfunktionen und die Terme $[\mathsf{s}, \mathtt{true}]$ stehen für den momentanen Speicherzustand.

Wert `TRUE`, im zweiten (Gleichung 3.3) den Wert `FALSE`. Sofern wir Prozeduren mit Seiteneffekten ausschliessen, tritt das Problem in dieser Form nicht mehr auf. Ansonsten müssten wir die Bedeutung des &-Operators statt durch Gleichung 3.1 durch Gleichung 3.4 festlegen, welche besagt, dass der zweite Operand auf dem durch den ersten Operanden (evtl.) veränderten Speicherzustand evaluiert wird.

$$\begin{aligned} &\forall \mathtt{b1}, \mathtt{b2} : \mathtt{BOOLEAN}, \mathtt{s} : \mathtt{Store}. \\ &\quad [\![\mathtt{b1} \mathbin{\&} \mathtt{b2}]\!][\mathtt{s}, \mathtt{true}] == [\![\mathtt{b1}]\!][\mathtt{s}, \mathtt{true}] \wedge [\![\mathtt{b2}]\!]([\![\mathtt{b1}]\!][\mathtt{s}, \mathtt{true}]) \end{aligned} \tag{3.4}$$

□

Selbstverständlich wäre es mit algebraischer denotationeller Semantik möglich, eine Programmiersprache mit globalen Variablen zu beschreiben. Wie im obigen Beispiel gezeigt wird, hätte dies aber zur Folge, dass die üblichen algebraischen Gesetze nicht mehr gelten. Unserer Meinung nach ist dies auch ein häufiger Grund für die Fehlinterpretation von Programmen mit Seiteneffekten.

Prozeduren sind in $\mathsf{Oberon}_\mathsf{T}$ immer Funktionsprozeduren und frei von Seiteneffekten. Wir verzichten deshalb auch auf die Verwendung von Referenzparametern (`VAR`-Parameter) und lassen dafür strukturierte Resultattypen zu. Andere Ansätze [Hoa72, Gan83] vermeiden Probleme mit Seiteneffekten, indem der Zustand des ADT nur als impliziter Parameter (modul-globale Variable) vorhanden ist. Dies hat aber den Nachteil, dass es jeweils nur eine Instanz des ADT geben kann.

Da das Verhalten von $\mathsf{Oberon}_\mathsf{T}$-Prozeduren grundsätzlich kontextunabhängig sein soll, verzichten wir auch auf verschachtelte Prozedurdeklarationen.

Jeder Prozedurrumpf muss als *letzte Anweisung* explizit einen `RETURN`-Wert zurückgeben.

3.4 Module

Das Modul `SYSTEM` darf nicht importiert werden.

3.5 Typen

Wir verzichten auf folgende Oberon-Typen:

- `POINTER`
- `PROCEDURE`
- `LONGREAL, REAL`

- LONGINT, SHORTINT
- SET

3.6 Standardprozeduren

Wir verzichten auf folgende Standardprozeduren:

- ENTIER, LONG, SHORT (Typkonversionen)
- NEW (Speicherallokation)
- COPY (Kopieren von Zeichenketten mit VAR-Parameter)
- EXCL, INCL (SET - Operationen)
- INC, DEC (INTEGER - Operationen mit VAR-Parametern)

3.7 Implementation

Die Syntax-Analyse für unser experimentelles System wurde mit GIPSY [Hel95, Mar94] implementiert. Basierend auf erweiterbaren attributierten Grammatiken [Mar94] erlaubt dieses Werkzeug die Erweiterung des *front-end* zu Verifikationszwecken [Sch]. Für die Codegenerierung benützen wir vorderhand einen Standard-Oberon-2-Compiler, wobei wir

- bei der Verwendung von FOR-Schleifen auf die Einhaltung der dynamischen Semantik gemäss Abschnitt 4.8.4 achten,
- strukturierte Rückgabewerte mit Hilfe von Zeigervariablen simulieren, indem wir innerhalb des Prozedurrumpfs Speicherplatz für den Rückgabewert allozieren und den tatsächlich zurückgebenen Zeiger sofort nach Abarbeitung der Funktionsprozedur dereferenzieren.

Der Entscheid, für die Codegenerierung vorderhand einen Standard-Oberon-2-Compiler zu benützen, ist rein pragmatischer Natur und liegt darin begründet, dass dessen Neu-Implementation einen erheblichen Aufwand mit sich bringt.

Kapitel 4

Semantik der Sprache $\mathsf{Oberon_T}$

4.1 Algebraische denotationelle Semantik

Die *algebraische denotationelle Semantik*[1] [GPG81, SK95, BHK89] ist eine formale Methode zur Beschreibung der Semantik von Programmiersprachen. Dabei ordnen *Semantikfunktionen*, die durch *semantische Gleichungen* definiert sind, jedem gültigen Satz der Programmiersprache eine Bedeutung aus den *semantischen Bereichen* zu. *Syntaktische Kategorien* bezeichnen Mengen von syntaktischen Objekten der Sprache (z.B. Ausdrücke, Anweisungen etc.) und werden aus der *abstrakten Syntax*[2] abgeleitet. Semantische Bereiche sind Σ-Algebren, deren Eigenschaften durch entsprechende algebraische Spezifikationen festgelegt sind. Semantikfunktionen sind somit Abbildungen von syntaktischen Kategorien in semantische Bereiche.

Im Rahmen der formalen Sprachdefinition betrachten wir ausschliesslich die *dynamische* Semantik von $\mathsf{Oberon_T}$. Diese spezifiziert das Laufzeitverhalten eines *gültigen* Programms, während die *statische* Semantik festlegt, wann ein Programm gültig ist. Die statische Semantik befasst sich demnach mit Aspekten wie Deklarationsanalyse und Typkorrektheit und ist nicht Thema dieses Berichts. Wir gehen bei der Spezifikation der dynamischen Semantik stets davon aus, dass ein *gültiges* $\mathsf{Oberon_T}$-Programm vorliegt. Die statische Semantik entspricht bis auf erwähnte Ausnahmen jener von Oberon.

Die Einfachheit der Implementationssprache und ihrer Semantik ist ein wichtiges Kriterium, auf das wir in der Folge grössten Wert legen. Es zeigt sich, dass die Definition einer formalen Semantik dazu beiträgt, die Programmiersprache so kompakt wie möglich zu halten, ohne dass dies für den Anwender bedeutende Unannehmlichkeiten mit sich bringt. So können wir z.B. durch die Einschränkung, dass `RETURN`-Anweisungen nur am Ende eines Prozedurrumpfs

[1] Statt von *algebraischer denotationeller Semantik* spricht man oft auch einfach von *algebraischer Semantik* [SK95].

[2] siehe Abschnitt 4.4

vorkommen dürfen, auf die Einführung von *continuations*[3] verzichten. Indem wir von Rücksprüngen aus Verzweigungen, Schleifen und beliebigen Schritten einer Anweisungssequenz absehen, erhalten wir einerseits besser lesbare Programme und vereinfachen andererseits die Spezifikation.

Die Semantik der Programmiersprache ist durch eine Hierarchie von algebraischen Spezifikationsmoduln (LSL-*traits*) gegeben. Eine algebraische Spezifikation in der Sprache LSL besteht im wesentlichen aus fünf Blöcken:

Titelblock Der Titelblock definiert den Namen und die Parameter des Spezifikationsmoduls. Ein Spezifikationsmodul wird als *trait* bezeichnet.

Importblock Der Importblock (`includes ...`) listet die importierten Spezifikationsmodule auf und instanziert deren Parameter. Die *includes*-Relation ist transitiv.

Signaturblock Der Signaturblock (`introduces ...`) definiert Operatoren und Sorten und damit die Syntax der Schnittstelle des spezifizierten ADT.

Axiomenblock Der Axiomenblock (`asserts ...`) spezifiziert die Semantik der Operatoren.

Theoremblock Im Theoremblock (`implies ...`) finden wir die Theoreme, welche aufgrund der Semantik der Operatoren gelten sollten. Diese Theoreme dienen der semantischen Untersuchung der Spezifikation. Sie müssen aus den Axiomen herleitbar sein.

Eine detaillierte Einführung in die Spezifikationssprache LSL findet man in [GH93]. Für eine formale Definition der Semantik algebraischer Spezifikationen verweisen wir auf die einschlägige Literatur [GH78, GTW78, PBB$^+$82, EGL89, Wir91].

4.2 Vorgehen

Wir definieren die algebraische denotationelle Semantik von $\mathsf{Oberon}_\mathsf{T}$ in folgenden Schritten:

1. Die Grammatik-Definition von Oberon enthält Iterationen, Optionen und Alternativen, die nicht direkt auf eine Signatur einer algebraischen Spezifikation abbildbar sind[4]. Sie wird deshalb einer geeigneten Transformation unterzogen. Ausserdem benötigen wir für die Implementation von Werkzeugen mit GIPSY [Hel95] eine LL(1)-Grammatik, so dass z.B. auch Linksrekursionen eliminiert werden müssen.

[3] *Continuations* sind semantische Bereiche, mit deren Hilfe die Bedeutung von Anomalien des Kontrollflusses, wie z.B. `GOTO`- oder `RETURN`-Anweisungen, beschrieben werden [SK95, All86].

[4] siehe Abschnitt 4.3.1

2. Aus der transformierten Grammatik erzeugen wir die Signatur der konkreten Syntax von $\mathsf{Oberon}_\mathsf{T}$.

3. Ausgehend von der Signatur der konkreten Syntax, definieren wir diejenige der abstrakten Syntax.

4. Die syntaktischen Kategorien ergeben sich aus den Sorten der Signatur der abstrakten Syntax.

5. Die semantischen Bereiche können unabhängig von den bisherigen Schritten spezifiziert werden.

6. Mit Hilfe der semantischen Funktionen wird den syntaktischen Konstrukten ihre Bedeutung im semantischen Bereich zugeordnet. An dieser Stelle wird auch die Fehlerbehandlung vorgenommen, indem die Funktionen partiell sind.

In den folgenden Abschnitten werden die einzelnen Schritte der Semantik-Definition genauer betrachtet.

4.3 Signatur der konkreten Syntax

4.3.1 Kontextfreie Grammatik → Signatur

Jeder kontextfreien Grammatik $G = \langle N, T, s_0, P \rangle$ kann eine Signatur $\Sigma(G) = \langle S, \Omega \rangle$ wie folgt zugeordnet werden:

1. Die Sorten sind gegeben durch $S = N$.

2. Die Operatoren ergeben sich folgendermassen:

   - Für jede Produktion $n \rightarrow t$, mit $t \in T$, $n \in N$, definieren wir einen Operator $t :\rightarrow n$.

   - Für jede Produktion $n \rightarrow n_1 \ldots n_k$, mit $n \in N$, $n_1 \ldots n_k \in N^*$, definieren wir einen Operator $n : n_1, \ldots, n_k \rightarrow n$.

Man beachte, dass die meisten Grammatik-Definitionen in EBNF beschrieben sind. Um die obigen Regeln anwendbar zu machen, müssen solche Beschreibungen zuerst einer geeigneten Transformation unterzogen werden, so dass sie nur noch Produktionen der Form $n \rightarrow n_1 \ldots\ n_k$ und $n \rightarrow t$ enthalten.

4.3.2 Grammatik-Transformationen

Mit Hilfe der Regeln gemäss Tabelle 4.1 erzeugen wir aus der eingeschränkten Oberon-Grammatik eine LL(1)-Grammatik, aus welcher direkt die Signatur der konkreten Syntax abgeleitet werden kann. Die Regeln werden auf alle Produktionen rekursiv so lange angewendet, bis keine Transformation mehr möglich ist. In der Grammatik gemäss Anhang A.1.2 wurden Alternativen und Optionen noch nicht eliminiert und die Terminal-Symbole noch nicht extrahiert.

Ziel	Produktionen	
	Original	transformiert
Extraktion von Terminal-Symbolen	$x \rightarrow mtn$	$x \rightarrow mkn$ $k \rightarrow t$
Elimination von Iterationen	$x \rightarrow m\{n\}$	$x \rightarrow mk$ $k \rightarrow [nk]$
Elimination von Alternativen	$x \rightarrow m(n_1 \mid n_2)$	$x \rightarrow mn_1$ $x \rightarrow mn_2$
Elimination von Optionen	$x \rightarrow m[n]$	$x \rightarrow mn$ $x \rightarrow m$
Elimination von Linksrekursionen	$x \rightarrow m \mid xn$ $x \rightarrow m \mid xnm$ $x \rightarrow m\{nm\}$ $x \rightarrow m \mid xn[m]$	$x \rightarrow m\{n\}$ $x \rightarrow m\{nm\}$ $x \rightarrow m[nx]$ $x \rightarrow m[n[x]]$

x, m, n, n_1, n_2, k: Nichtterminal-Symbole; t: Terminal-Symbol

Tabelle 4.1: Grammatik-Transformationen

4.3.3 Signatur der konkreten Syntax

Als Beispiel betrachten wir an dieser Stelle die erste Produktion der Oberon_T-Grammatik gemäss Anhang A.1.2:

```
Module ::= 'MODULE' Ident ';' DeclSeq1 'END' ident '.'
Module ::= 'MODULE' Ident ';' 'IMPORT' Import Imports ';'
           DeclSeq1 'END' ident '.'
```

wird in folgende Signatur abgebildet:

```
module: Ident, DeclSeq1, Ident → Module
module: Ident, Import, Imports, DeclSeq1, Ident → Module
ident: → Ident % Extraktion des Terminal-Symbols ‘ident’
```

Man könnte die Schlüsselwörter wie die anderen Terminal-Klassen behandeln, da sie aber für die Semantik-Definition nicht von Bedeutung sind, lassen wir sie in der Signatur weg.

Die vollständige Signatur der konkreten Syntax von Oberon$_T$ findet man in Anhang A.1.3.

4.4 Die abstrakte Syntax von Oberon$_T$

Im Gegensatz zur *konkreten Syntax* abstrahiert man bei der *abstrakten Syntax* einer Sprache von sämtlichen Details, welche ausschliesslich für die syntaktische Analyse (*parsing*) benötigt werden [SK95]. Dadurch wird die Beschreibung bedeutend kompakter und enthält nur noch Informationen, welche für die Untersuchung der Semantik von Interesse sind. Die abstrakte Syntax einer Sprache ist nicht eindeutig definiert. Sie kann somit der angewendeten Technik zur Beschreibung der Semantik und dem untersuchten Aspekt angepasst werden.

Abbildung 4.1 zeigt die Signatur der abstrakten Syntax von Oberon$_T$. Sie besteht aus den syntaktischen Konstrukten, deren Bedeutung wir festlegen möchten[5]. Der Parameter `SynCat` kann mit einer beliebigen syntaktischen Kategorie instanziert werden.

```
AbstractSyntax(SynCat):trait
includes
  DecimalLiterals(INTEGER for N)

introduces
  TRUE,FALSE:→ BOOLEAN
  __&__,__|__,__=*__,__#__:BOOLEAN,BOOLEAN → BOOLEAN
  ¬__:BOOLEAN → BOOLEAN
  -__:INTEGER → INTEGER
  __+__,__-__,__*__,DIV,MOD:INTEGER,INTEGER → INTEGER
  __<__,__<=__,__>__,__>=__,__=*__,__#__:INTEGER,INTEGER → BOOLEAN
  __;__:Stmt,Stmt → Stmt
% IF_THEN_ELSE_END:BOOLEAN,Stmt,Stmt → Stmt
  IfThenElse:BOOLEAN,Stmt,Stmt → Stmt
% FOR__:=_TO_DO_END:INTEGER,INTEGER,INTEGER,Stmt → Stmt
  LOOP:INTEGER,INTEGER,INTEGER,Stmt → Stmt
  RETURN:SynCat → Stmt
  __:=__:SynCat,SynCat → Stmt
  __=*__,__#__:SynCat,SynCat → BOOLEAN
```

Abbildung 4.1: Signatur der abstrakten Syntax von Oberon$_T$

[5] Der Oberon-Gleichheitsoperator '=' ist in allen LSL-*traits* mit '=*' oder '=*' bezeichnet, damit er vom LSL-Gleichheitsoperator '=' unterscheidbar ist (siehe auch Abschnitt 4.8.1).

4.5 Syntaktische Kategorien von $\mathsf{Oberon_T}$

Da wir bei der Verifikation von Programmen Verhaltensäquivalenzen untersuchen, welche durch typisierte Gleichungen definiert sind, unterscheiden wir bereits bei der abstrakten Syntax zwischen Ausdrücken mit verschiedenen Resultattypen. Dies führt dazu, dass sich die Menge der syntaktischen Kategorien aus einem konstanten Teil für Anweisungen und Ausdrücke in Basistypen und einem variablen Teil mit Ausdrücken in neu deklarierten Typen zusammensetzt. Wir betrachten also die Deklaration neuer Typen als eine Erweiterung der Sprache.

Ein $\mathsf{Oberon_T}$-Programm wird durch folgende syntaktische Kategorien modelliert:

`Stmt` : Anweisungen

`BOOLEAN` : boolsche Ausdrücke

`INTEGER` : Ausdrücke mit ganzen Zahlen

Bezeichner werden als Ausdrücke des entsprechenden Typs interpretiert. Für jeden neu deklarierten Typen *Type* wird diese Menge um die syntaktische Kategorie `Type` erweitert. Strukturierte Typen (*Records* und *Arrays*) werden in die semantischen Bereiche `Record` bzw. `Array` abgebildet. Für jeden deklarierten *Record*-Typen *RecType* wird eine syntaktische Kategorie *RecType*`_Fld` eingeführt, in der die Namen der entsprechenden *Record*-Felder enthalten sind.

4.6 Semantische Bereiche von $\mathsf{Oberon_T}$

Wir verwenden in unserer Definition folgende semantische Bereiche:

`Bool` : boolsche Werte (Anhang B.1.1)

Spezifiziert die üblichen Operationen einer booleschen Algebra. Zu ihren Theoremen gehören auch Gleichungen aus anderen *traits* (`AC`, `Distributive`, `Involutive`, `Transitive`), die in [GH93] näher erläutert sind.

`Int` : ganze Zahlenwerte (Anhang B.2.1)

Die Zahlen-Algebra wurde so spezifiziert, dass die Darstellung jeder Zahl genau eine Normalform hat und dass aus den Gleichungen ein effizientes Termersetzungssystem erzeugt werden kann, welches die Relationen $<, \leq, >, \geq$ für jedes Zahlenpaar *automatisch* entscheidet. Die Spezifikation in Anhang B.2.1 ist deshalb verschieden von jener in [GH93][6].

[6] Das mit LP (Standardeinstellungen) erzeugte Termersetzungssystem für den `Integer`-*trait* aus [GH93] reduziert z.B. den Term $\mathtt{3} < \mathtt{5}$, bzw. $\mathtt{s(s(s(0)))} < \mathtt{s(s(s(s(s(0)))))}$ nicht ohne Benutzer-Interaktion zu `true`.

`Array` : *Array*-Werte (Abb. 4.2)

Die Standard-Spezifikation für *Arrays* wurde um die Operatoren `newA: →Array` und `assNF: Array, Int, SV → Array` für die Darstellung eines leeren *Arrays* resp. eines *Arrays* in Normalform erweitert. Die Einführung einer Normalform reduziert die mittlere Länge der Terme, die im Laufe eines Beweises entstehen, und beschleunigt dadurch dessen maschinelle Ausführung.

```
Array(Array,Int):trait
includes
  Int(Int)
introduces
  newA:→ Array
  assNF:Array,Int,SV → Array
  assign:Array,Int,SV → Array
  __[__]:Array,Int → SV
asserts
  Array generated freely by newA,assNF
  Array partitioned by __[__]
  ∀a:Array,v,w,sv:SV,i,j:Int
  assNF(a,i,v)[j] == if i = j then v else a[j];
  assign(a,i,v)[j] == if i = j then v else a[j];
  assign(newA,i,v) == assNF(newA,i,v);
  assign(assNF(a,i,v),j,w) ==
    if i = j then assNF(a,i,w)
    else
      if i > j then assNF(assign(a,j,w),i,v)
      else assNF(assign(a,i,v),j,w);
  assign(assign(a,i,v),j,w) ==
    if i = j then assNF(a,i,w)
    else
      if i > j then assNF(assign(a,j,w),i,v)
      else assNF(assign(a,i,v),j,w);
implies
  converts
    assign,
    __[__] exempting ∀i:Int newA[i]
```

Abbildung 4.2: Algebraische Spezifikation für *Array*-Werte

`Field` : Namen von *Record*-Feldern (Abb. 4.3)

`Record` : *Record*-Werte (Abb. 4.3)

Die Spezifikation für die Darstellung von *Record*-Werten gleicht jener für *Arrays*, ausser dass die Indizes Bezeichner für die *Record*-Felder anstatt ganze Zahlen sind. Auch hier wurde zur Reduktion der mittleren Termlänge eine Normalform eingeführt.

```
Record(R,Field):trait
introduces
  newR:→ R
  setNF:R,Field,SV → R
  set:R,Field,SV → R
  __.__:R,Field → SV
asserts
  R generated freely by newR,setNF
  R partitioned by __.__
  ∀r:R,g,h:Field,v,w:SV
  set(r,g,v).h == if g = h then v else r.g;
  setNF(r,g,v).h == if g = h then v else r.g;
  set(newR,g,v) == setNF(newR,g,v);
  set(setNF(r,g,v),h,w) ==
    if g = h then setNF(r,g,w)
    else setNF(set(r,h,w),g,v);
  set(set(r,g,v),g,w) == set(r,g,w);
implies
  converts
    set,
    __.__ exempting ∀g:Field newR.g
```

Abbildung 4.3: Algebraische Spezifikation für *Record*-Werte

`Loc` : Adressen im Speicher (Abb. 4.5)

`Store` : Speichermodell (Abb. 4.4)

Der Speicher wird als eindimensionales *Array* mit Normalformdarstellung (siehe `Array`) modelliert, wobei alle im System vorkommenden Bezeichner für Variablen als Indizes dienen und die gespeicherten Werte (Parameter `SV`) beliebige Instanzen einfacher oder strukturierter Typen sein können.

`M` : Speicher mit Gültigkeits-Indikator für die Fehlerbehandlung[7] (Abb. 4.5)

Erweiterung von `Store`.

Wir verzichten auf die Einführung eines semantischen Bereichs *Environment*, in dem die Bindung von Bezeichnern für Variablen und Prozeduren an Adressen

[7]siehe Abschnitt 4.7

```
Store(Store,Loc,SV):trait
introduces
  new,err:→ Store
  putNF:Store,Loc,SV → Store
  put:Store,Loc,SV → Store
  __[__]:Store,Loc → SV
asserts
  Store generated freely by err,new,putNF
  Store partitioned by __[__]
  ∀s:Store,i,j:Loc,v,w:SV
  putNF(s,i,v)[j] == if i = j then v else s[j];
  put(new,i,v) == putNF(new,i,v);
  put(putNF(s,i,v),j,w) ==
    if i = j then putNF(s,i,w)
    else putNF(put(s,j,w),i,v);
implies
  converts
    put,
    __[__] exempting
      ∀i:Loc new[i],err[i]
```

Abbildung 4.4: Algebraische Spezifikation für den Speicher

im Speicher definiert wird. Da wir auf die Verwendung von dynamischen Datenstrukturen, Prozedurvariablen und Adressarithmetik verzichten, bietet diese Vereinfachung keine Probleme, bringt aber den Vorteil mit sich, dass die später zu untersuchenden Terme kürzer werden.

Bezeichner für Prozeduren werden durch semantische Gleichungen an eine Folge von Anweisungen gebunden.

4.7 Fehlerbehandlung

Einige der nachfolgend eingeführten Semantikfunktionen definieren die Bedeutung von syntaktischen Objekten, bei deren Evaluation Laufzeitfehler auftreten können. Trotzdem die Verhinderung solcher Fehler eines der wichtigsten Entwurfskriterien von $\mathsf{Oberon}_\mathsf{T}$ ist, sind prinzipiell folgende Fälle möglich:

- `INTEGER`-*overflow*: Überschreitung des Bereichs der ganzen Zahlen.
- `INTEGER`-*underflow*: Unterschreitung des Bereichs der ganzen Zahlen.
- `INTEGER`-*division-by-zero*: Division durch Null bei ganzen Zahlen.
- `ARRAY`-*overflow*: Überschreitung der Bereichsgrenzen eines *Arrays*.

- `ARRAY`-*underflow*: Unterschreitung der Bereichsgrenzen eines *Arrays*.

Die letzten drei Fehler könnten durch ein Verbot der `INTEGER`-Division bzw. den Verzicht auf *Arrays* von vornherein ausgeschlossen werden — beides wären aber nach unserer Ansicht zu starke Einschränkungen. Die `INTEGER`-Division wird für den Umgang mit physikalischen Grössen (Berechnung von Geschwindigkeiten, Beschleunigungen etc.) benötigt, während *Arrays* der Repräsentation von Datenstrukturen dienen. Eine andere Möglichkeit Laufzeitfehler zu verhindern, welche auf Bereichsüberschreitungen zurückzuführen sind, wäre die Verwendung von endlichen Zahlenkörpern statt des Bereichs der ganzen Zahlen. Man könnte dann z.B. definieren, dass $MAXINT + 1 = MAXINT$ oder $MAXINT + 1 = MININT$ gilt. Diese Semantik entspräche jener der realen Implementation auf einem Computer mit endlichen Ressourcen. Der Programmierer muss sich der Endlichkeit der darstellbaren Zahlen in jedem Fall bewusst sein und sie in den Systementwurf miteinbeziehen. "Abstrahiert" er von dieser Gegebenheit, besteht die Gefahr, dass beim Einsatz des Programmes Laufzeitfehler auftreten.

Um solche Probleme bei der symbolischen Untersuchung eines ADT aufzudecken, wird der Speicher M durch ein Paar $M = Store \times Bool$ modelliert, wobei der boolesche Wert angibt, ob die Evaluation eines Ausdrucks oder einer Anweisung durch die Semantikfunktion einen gültigen Wert ergeben hat. Falls die Bedeutung eines syntaktischen Konstrukts in einem bestimmten Kontext einen fehlerhaften Speicherzustand ergibt, ist das Resultat aller darauf angewandten Semantikfunktionen undefiniert. Das bei der Verifikation eingesetzte Termersetzungssystem, kann deshalb die entstehenden Terme nicht weiter reduzieren, wodurch der Korrektheitsbeweis blockiert wird.

Indem wir die Fehlerdetektion bei der Auswertung der Semantikfunktionen vornehmen, bleiben die Spezifikationen der semantischen Bereiche so einfach wie möglich. Trotzdem erkennen wir potentielle Probleme während der Verifikation und können ein Termersetzungssystem erzeugen, das Terme, in denen ein ungültiger Speicherzustand auftritt, nicht weiter reduziert. Damit ist es auch verhältnismässig einfach, einen Fehler zu lokalisieren.

Eine andere Möglichkeit der Fehlerbehandlung bestünde darin, jeden semantischen Bereich um ein Fehlerelement $\perp$ zu erweitern. Der Nachteil dieser Methode besteht darin, dass die Einführung solcher Fehlerelemente bei Algebren eine starke Zunahme der Anzahl und Komplexität der Gleichungen zur Folge hat und die Gefahr von inkonsistenten Spezifikationen zunimmt [GTW78]. Zudem würde die Reduktion von fehlerhaften Termen auf das Element $\perp$ die Lokalisierung von Fehlern wesentlich erschweren.

Eine elegantere Fehlerbehandlung erlauben *order-sorted algebras* [GM89] [GWMJ92], bei denen Sorten in eine partielle Ordnung von Untersorten aufgeteilt und kritische Operationen nur auf den entsprechenden Untersorten definiert werden. Beispielsweise lässt sich die Sorte Nat in die Untersorten $Nat > NzNat, Zero$ aufteilen. $NzNat$ ist dann die Sorte für die natürlichen Zahlen ohne

0 und *Zero* jene für 0. Die Division hat die Signatur $DIV : Nat, NzNat \to Nat$ und ist somit ausschliesslich für Divisoren definiert, die verschieden von 0 sind. Wir verzichten auf die Anwendung von *order-sorted algebras*, weil das verfügbare Termersetzungssystem OBJ3 die Spezifikationen ausschliesslich auf der Basis initialer Algebra Semantik interpretiert.

4.8 Semantikfunktionen

Wir führen drei verschiedene Semantikfunktionen ein:

$$[\![\_\_]\!]_{\_\_} : \mathtt{SynCat}, \mathtt{M} \to \mathtt{SemDom}$$

für die Bedeutung von Bezeichnern (*Identifiers*) und Ausdrücken (*Expressions*) auf dem momentanen Speicherzustand. Diese Klasse von Semantikfunktionen wird für jeden Datentypen instanziert (z.B. mit $\mathtt{SynCat} \leftarrow \mathtt{BOOLEAN}$, $\mathtt{SemDom} \leftarrow \mathtt{Bool}$).

$$[\![\_\_]\!]_{\_\_} : \mathtt{Stmt}, \mathtt{M} \to \mathtt{M}$$

für die Bedeutung von Anweisungen (*Statements*) und

$$[\![\_\_]\!]_{\_\_} : \mathtt{Stmt}, \mathtt{M} \to \mathtt{SemDom}$$

für die Bedeutung von Funktionsprozeduren bzw. der `RETURN`-Anweisung. Auch diese Signatur beschreibt eine Klasse von Semantikfunktionen, die je nach Datentyp des Rückgabewertes instanziert wird (z.B. mit $\mathtt{SemDom} \leftarrow \mathtt{Int}$).

Ausgehend vom Speichermodell beschreiben wir mit Hilfe dieser Semantikfunktionen die Bedeutung der $\mathsf{Oberon}_\mathsf{T}$-Sprachkonstrukte.

4.8.1 Gleichheitsoperatoren

Im Rahmen der Semantik-Spezifikation und auch später bei der Verifikation treten vier verschiedene Gleichheitsoperatoren auf, deren Bedeutung hier näher betrachtet werden soll.

'=' tritt als Vergleichsoperator sowohl in der Oberon- als auch in der LSL-Syntax auf:

- Der Oberon-Vergleichsoperator hat die Bedeutung, welche ihm durch die Semantikfunktionen zugeordnet wird. Damit er in den LSL-*traits* vom LSL-Gleichheitsoperator unterscheidbar ist, bezeichnen wir ihn dort mit '`=*`' oder mit '$=^*$'.
- Die Bedeutung des LSL-Gleichheitsoperators ist eine Kongruenz auf der Termalgebra.

'$=^*$', '=*' ist eine Umbenennung für den Oberon-Vergleichsoperator (siehe '=').

'==' hat die gleiche Bedeutung wie der LSL-Gleichheitsoperator '=', bindet aber schwächer als dieser, d.h. `a = b == c = d` ist identisch mit `(a = b) == (c = d)`.

'$\sim$' ist die Kongruenz auf Oberon-Termen, welche durch die Verhaltensäquivalenz[8] definiert wird.

Zwischen den einzelnen Operatoren gelten folgende Beziehungen:

$$\begin{aligned}
&\forall \mathtt{t1}, \mathtt{t2} : \mathtt{SynCat}, \mathtt{s} : \mathtt{Store}. \\
&\quad \forall \mathtt{s} : \mathtt{Store}. [\![\mathtt{t1}]\!][\mathtt{s}, \mathtt{true}] = [\![\mathtt{t2}]\!][\mathtt{s}, \mathtt{true}] \Rightarrow \mathtt{t1} \sim \mathtt{t2} && (4.1) \\
&\forall \mathtt{t1}, \mathtt{t2} : \mathtt{SynCat}, \mathtt{s} : \mathtt{Store}. \\
&\quad \mathtt{t1} = \mathtt{t2} \Rightarrow \forall \mathtt{s} : \mathtt{Store}. [\![\mathtt{t1}]\!][\mathtt{s}, \mathtt{true}] = [\![\mathtt{t2}]\!][\mathtt{s}, \mathtt{true}] && (4.2) \\
&\forall \mathtt{t1}, \mathtt{t2} : \mathtt{SynCat}, \mathtt{s} : \mathtt{Store}. \\
&\quad [\![\mathtt{t1} =^* \mathtt{t2}]\!][\mathtt{s}, \mathtt{true}] == [\![\mathtt{t1}]\!][\mathtt{s}, \mathtt{true}] = [\![\mathtt{t2}]\!][\mathtt{s}, \mathtt{true}] && (4.3)
\end{aligned}$$

Für die Implikation 4.1 gilt die Umkehrung im allgemeinen nicht, weil beispielsweise zwei identische Mengen durch verschiedene *Arrays* repräsentiert werden können. Die *Arrays* wären in diesem Fall verhaltensäquivalent, aber nicht gleich. Implikation 4.2 ist im allgemeinen nicht umkehrbar, da zwei identische Werte auch von verschiedenen Bezeichnern referenziert werden können.

4.8.2 Speichermodell

Einfache Speicherzugriffe (Abb. 4.5) bilden wir auf eindimensionale *Array*-Zugriffe ab. Die Hilfsfunktionen `sv: SemDom` $\rightarrow$ `SV` und deren Inverse `val: SV` $\rightarrow$ `SemDom` werden benötigt, damit Werte jedes beliebigen Typs in den Speicher geschrieben bzw. vom Speicher gelesen werden können (Typkonversion).

Speicherzugriffe bei *Arrays* (Abb. 4.8)

Das Lesen von Daten aus strukturierten Typen bietet keine besonderen Probleme. Schwieriger ist hingegen die Beschreibung der Semantik von Zuweisungen. Wir stossen dabei auf folgende Probleme:

[8] Die algebraische Spezifikation eines abstrakten Datentypen kann *Verhaltensäquivalenzen* spezifizieren, indem sie Mengen von Operatoren auf einer Sorte als sogenannte *Beobachtungsoperatoren* (*Observer*) auszeichnet. Zwei Terme der beobachteten Sorte gelten als gleich, wenn sie durch die Applikation aller Beobachtungsoperatoren der entsprechenden Sorte nicht voneinander unterscheidbar sind. Wir nennen sie dann *verhaltensäquivalent*.

```
Storable(SynCat,SemDom):trait
includes
  Store(Store,Loc,SV),
  VariableId(Loc),
  AbstractSyntax(SynCat)
M tuple of store:Store,valid:Bool
introduces
  sv:SemDom → SV
  val:SV → SemDom
  adr:SynCat → Loc
  __':SynCat → SynCat
  ⟦__⟧__:SynCat,M → SemDom
  ⟦__⟧__:Stmt,M → M
  ⟦__⟧__:Stmt,M → SemDom
  ⟦__⟧__:BOOLEAN,M → Bool
  __∼__:SynCat,SynCat → Bool
  __=>__:BOOLEAN,Bool → Bool
asserts
  SV partitioned by val:SV → SemDom
  SemDom partitioned by sv:SemDom → SV
  ∀s1,s2:Stmt,t1,t2:SynCat,s:Store,b:BOOLEAN,
    v1:SemDom,b1:SV
  val(sv(v1)) == v1;
  sv(val(b1)) == b1;
  ⟦t1' := t2⟧[s,true] ==
    [put(s,adr(t1),sv(⟦t2⟧[s,true])),true];
  (⟦s1; s2⟧[s,true]):SemDom ==
    (⟦s2⟧(⟦s1⟧[s,true]):M):SemDom;
  ⟦t1'⟧[s,true] ==
    if s ≠ new then val(s[adr(t1)]
    else ⟦t1'⟧[err,false];
  ⟦RETURN(t1)⟧[s,true] == ⟦t1⟧[s,true];
  (∀s(⟦b⟧[s,true] =>
    (⟦t1⟧[s,true] = ⟦t2⟧[s,true]))) => (b => (t1 ∼ t2));
  (∀s(⟦t1⟧[s,true] = ⟦t2⟧[s,true])) => (t1 ∼ t2);
  (t1 = t2) => ∀s(⟦t1⟧[s,true] = ⟦t2⟧[s,true]);
```

Abbildung 4.5: Semantikfunktionen für einfache Speicherzugriffe

1. In $\mathsf{Oberon_T}$ ist die Zuweisung von ganzen Strukturen möglich. Da wir in unserem Modell von der konkreten Berechnung von Adressen im Speicher abstrahieren, müssen wir einen anderen Weg finden, um Zuweisungen an ein *Array* so abzubilden, dass sowohl Zugriffe auf die ganze Struktur als auch auf Teile davon auf einfache Art und Weise möglich sind.
2. Wie Abbildung 4.6 zeigt, können *Designatoren*[9] beliebig komplex aufgebaut sein.

```
        TYPE
          Data = RECORD
            f1: BOOLEAN;
            f2: ARRAY 64 OF INTEGER;
          END;

          Column = ARRAY 256 OF Data;

          Field = ARRAY 128 OF Column;

        PROCEDURE Assign(...);
        VAR
          r: Column;
          f,g: Field;
          d: Data;
          i,j: INTEGER;
        BEGIN
          ...
          (* ganzes zweidimensionales Array zuweisen *)
          g := f;
          (* einzelnes Feld der Struktur zuweisen *)
(* 2 *)   g[2*i][i+j].f2[k] := 2*(i+j);
          (* ganzes eindimensionales Array zuweisen
          (Teil der Struktur) *)
          g[2*i-1] := r;
          ...
```

Abbildung 4.6: Zuweisungen an strukturierte Typen

Wir lösen diese Probleme folgendermassen:

1. Strukturierte Typen werden als gekapselte Objekte modelliert, die als Ganzes wie unstrukturierte Werte zu behandeln sind. Die Zuweisung an einzelne Teile der Struktur erfolgt gemäss den Erläuterungen unter Punkt (2.).

[9] Bezeichner für linke Seiten der Zuweisung

2. Zuweisungen an komplexe Designatoren werden in eine Sequenz von einfachen und *eindimensionalen* Zuweisungen zerlegt. Als *eindimensional* bezeichnen wir Zuweisungen von der Form

```
Assignment ::= Designator ':=' Expression
Designator ::= ident ('.' ident | '[' Expression ']').
```

Dazu führen wir für jeden strukturierten Typen eine von aussen nicht sichtbare Variable (_h) ein. Komplexe Designatoren werden von rechts nach links analysiert, indem wir feststellen, ob es sich um eine Zuweisung an ein *Array*- (endet mit `[`*Index*`]`) oder ein *Record*-Feld (endet mit `.`*Feldname*) handelt. Abbildung 4.7 zeigt anhand eines Beispiels, wie eine komplexe Zuweisung modelliert wird.

```
⟦ g[2*i][i+j].f2[k] := 2*(i+j) ⟧ [s,true] == ⟦
  _h:Data_f2_Type := g[2*i][i+j].f2; % einfache Zuweisung
  _h:Data_f2_Type[k] := 2*(i+j); % eindimensionale Zuweisung
  g[2*i][i+j].f2 := _h:Data_f2_Type; % neue komplexe Zuweisung

  _h:Data := g[2*i][i+j] % einfache Zuweisung
  _h:Data.f2 := _h:Data_f2_Type; % eindimensionale Zuweisung
  g[2*i][i+j] := _h:Data; % neue komplexe Zuweisung ...

  _h:Column := g[2*i];
  _h:Column[i+j] := _h:Data;
  g[2*i] := _h:Column;

  _h:Field := g;
  _h:Field[2*i] := _h:Column;
  g := _h:Field ⟧ [s,true] % ... einfache Zuweisung
```

Abbildung 4.7: Semantik der Zuweisung (2) aus Abb. 4.6

Speicherzugriffe bei *Records* (Abb. 4.9) werden analog zu jenen bei *Arrays* modelliert, ausser dass die Indizes Bezeichner für die *Record*-Felder anstatt ganze Zahlen sind.

4.8.3 Ausdrücke (*Expressions*)

Boolesche Ausdrücke (Abb. 4.10) werden in der zu erwartenden Art und Weise auf den semantischen Bereich `Bool` abgebildet. Da wir Seiteneffekte ausschliessen, ergeben sich hier keine Probleme. Ansonsten müssten die Gleichungen entsprechend den Erläuterungen in Abschnitt 3.3 modifiziert werden.

```
ARRAY_OF_Type(ARRAY_OF_Type,len_Type,SynCat,SemDom):trait
includes
  Array(Array,Int),
  INTEGER,
  Storable(ARRAY_OF_Type,Array),
  Storable(SynCat,SemDom)
introduces
  __[__]:ARRAY_OF_Type,INTEGER → SynCat
  _h:→ ARRAY_OF_Type
  len_Type:→ INTEGER
asserts
  ∀s:Store,a:Loc,v:SV,b:SynCat,n:INTEGER,
    t,t1:ARRAY_OF_Type,b1:Bool
  % Zugriff auf ganzen Array:in Storable geregelt
  % Zugriff auf ein Feld:
  ⟦t[n]⟧[s,true] ==
    if s ≠ new
      ∧ inRange(0, ⟦n⟧[s,true],⟦len_Type⟧[s,true] - 1) then
      val((⟦t⟧[s,true]) [ (⟦n⟧[s,true]) ]
    else ⟦t[n]⟧[err,false];
  % Zuweisung ganzer Array:in Storable geregelt
  % Zuweisung einzelner Felder:
  % _h := t; _h[n] := b; t := _h
  (⟦t[n] := b⟧[s,true]):M ==
    if inRange(0,⟦n⟧[s,true],⟦len_Type⟧[s,true] - 1) then
      ⟦t := _h'⟧[put(s,
        adr(_h),
        sv(assign(⟦t⟧[s,true],
          ⟦n⟧[s,true],
          sv(⟦b⟧[s,true])))),true]
    else ⟦t[n] := b⟧[err,false];
  ⟦t =* t1⟧[s,true] == ⟦t⟧[s,true] = ⟦t1⟧[s,true];
  ⟦t # t1⟧[s,true] == ⟦t⟧[s,true] ≠ ⟦t1⟧[s,true];
```

Abbildung 4.8: Semantikfunktionen für Speicherzugriffe bei *Arrays*

Bemerkung zu Abbildung 4.8: Die Sequenz `_h:=t; _h[n]:=b; t:=_h` für die Zuweisung einzelner Felder wird durch *eine* Gleichung beschrieben. Dies verhindert einerseits das Auftreten einer endlosen Rekursion, da sonst `_h[n]:=b` wieder neu instanziert werden könnte, und ist andererseits effizienter bei der Termersetzung.

```
RECORD_Type(RECORD_Type,Field,SynCat,SemDom):trait
includes
  Record(Record,Field),
  Storable(RECORD_Type,Record),
  Storable(SynCat,SemDom)
introduces
  __.__:RECORD_Type,Field → SynCat
  _h:→ RECORD_Type
asserts
  ∀s:Store,a:Loc,v:SV,b:SynCat,g:Field,t,t1:RECORD_Type,
  b1:Bool
  % Zugriff auf ganzen Record:in Storable geregelt
  % Zugriff auf ein Feld:
  ⟦t.g⟧[s,true] == val((⟦t⟧[s,true]).g);
  % Zuweisung ganzer Record:in Storable geregelt
  % Zuweisung einzelner Felder:
  % _h := t; _h.g := b; t := _h;
  (⟦t.g := b⟧[s,true]):M ==
    ⟦t := _h'⟧[put(s,
      adr(_h),
      sv(set(⟦t⟧[s,true],g,sv(⟦b⟧[s,true])))),true];
  ⟦t =* t1⟧[s,true] == ⟦t⟧[s,true] = ⟦t1⟧[s,true];
  ⟦t # t1⟧[s,true] == ⟦t⟧[s,true] ≠ ⟦t1⟧[s,true];
```

Abbildung 4.9: Semantikfunktionen für Speicherzugriffe bei *Records*

```
BOOLEAN:trait
includes
  Storable(BOOLEAN,Bool)
asserts
  ∀b1,b2:BOOLEAN,s:Store
  ⟦TRUE⟧[s,true] == true;
  ⟦FALSE⟧[s,true] == false;
  ⟦¬ b1⟧[s,true] == ¬ ⟦b1⟧[s,true];
  ⟦b1 & b2⟧[s,true] == ⟦b1⟧[s,true] ∧ ⟦b2⟧[s,true];
  ⟦b1 | b2⟧[s,true] == ⟦b1⟧[s,true] ∨ ⟦b2⟧[s,true];
  ⟦b1 =* b2⟧[s,true] == ⟦b1⟧[s,true] = ⟦b2⟧[s,true];
  ⟦b1 # b2⟧[s,true] == ⟦b1⟧[s,true] ≠ ⟦b2⟧[s,true];
```

Abbildung 4.10: Semantikfunktionen für boolesche Ausdrücke

Ausdrücke mit ganzen Zahlen (Anhang B.2.2) behandeln wir analog zu den booleschen Ausdrücken, ausser dass bei kritischen Operationen zusätzlich eine Überprüfung auf Bereichsüberschreitungen (*overflow* und *underflow*) bzw. Divisionen durch Null vorgenommen wird.

4.8.4 Anweisungen (*Statements*)

Zuweisungen (Abb. 4.5) Die Semantik der Zuweisung wurde bereits bei der Beschreibung des Speichermodells behandelt.

Sequenz, Verzweigung, Schleife (Abb. 4.11) haben die übliche Semantik. Es wird z.B. spezifiziert, dass die Ausführung der Sequenz `s1; s2` auf einem bestimmten Speicherzustand den gleichen Effekt hat, wie die Ausführung von `s1` auf dem alten und die anschliessende Ausführung von `s2` auf dem neuen Speicherzustand. `LOOP(d,f1,f2,s1)` hat die Bedeutung einer `FOR`-Schleife (`FOR d := f1 TO f2 DO s1 END`).

`RETURN`**-Anweisung** (Abb. 4.5) Mit Hilfe der `RETURN`-Anweisung wird einer Sequenz von Anweisungen ein einzelner Wert (Resultat einer Funktionsprozedur) zugeordnet. Ihre Bedeutung ist somit das Resultat der Evaluation des Argumentausdrucks auf dem aktuellen Speicherzustand. Indem wir festlegen, dass `RETURN`-*Statements* ausschliesslich als letzte Anweisung innerhalb eines Prozedurrumpfs vorkommen, vereinfachen wir die Semantik von Anweisungssequenzen wesentlich.

4.9 Hierarchie der Semantik-Spezifikation

Abbildung 4.12 vermittelt einen Überblick über die `includes`-Beziehungen zwischen den einzelnen *traits* der Semantik-Spezifikation von $\mathsf{Oberon}_\mathsf{T}$.
Ausser den bereits behandelten Spezifikationen enthält Abbildung 4.12 die folgenden *traits* (Anhang B.2.3):

`OberonT` ist die Schnittstelle der Semantik-Spezifikation der Implementationssprache und wird in jede Spezifikation eines neuen ADT importiert.

`StandardTypes` importiert alle Standard-Typen und wird zusammen mit `ARRAY_OF_Type` und `RECORD_Type` für die Festlegung der Semantik von neu definierten Typen gebraucht.

`Orders` spezifiziert Ordnungsrelationen auf binär repräsentierten ganzen Zahlen.

`BinaryNumbers` spezifiziert arithmetische Operationen auf binär repräsentierten ganzen Zahlen.

```
Statements:trait
includes
  INTEGER,
  BOOLEAN
introduces
  skip:→ Stmt
asserts
  ∀b:BOOLEAN,d,f1,f2:INTEGER,s1,s2:Stmt,
    s:Store,a:Loc,v:SV
  (⟦s1; s2⟧[s,true]):M ==
    (⟦s2⟧(⟦s1⟧[s,true]):M):M;
  (⟦IfThenElse(b,s1,s2)⟧[s,true]):M ==
    if(⟦b⟧[s,true]) then ⟦s1⟧[s,true] else ⟦s2⟧[s,true];
  ⟦LOOP(d,f1,f2,s1)⟧[s,true] ==
    if(⟦f1 <= f2⟧[s,true]) then
      ⟦LOOP(d,succ(f1),f2,s1)⟧(⟦s1⟧(⟦d := f1⟧[s,true]))
      else[s,true];
  ⟦skip⟧[s,true] == [s,true];
```

Abbildung 4.11: Semantikfunktionen für Anweisungen

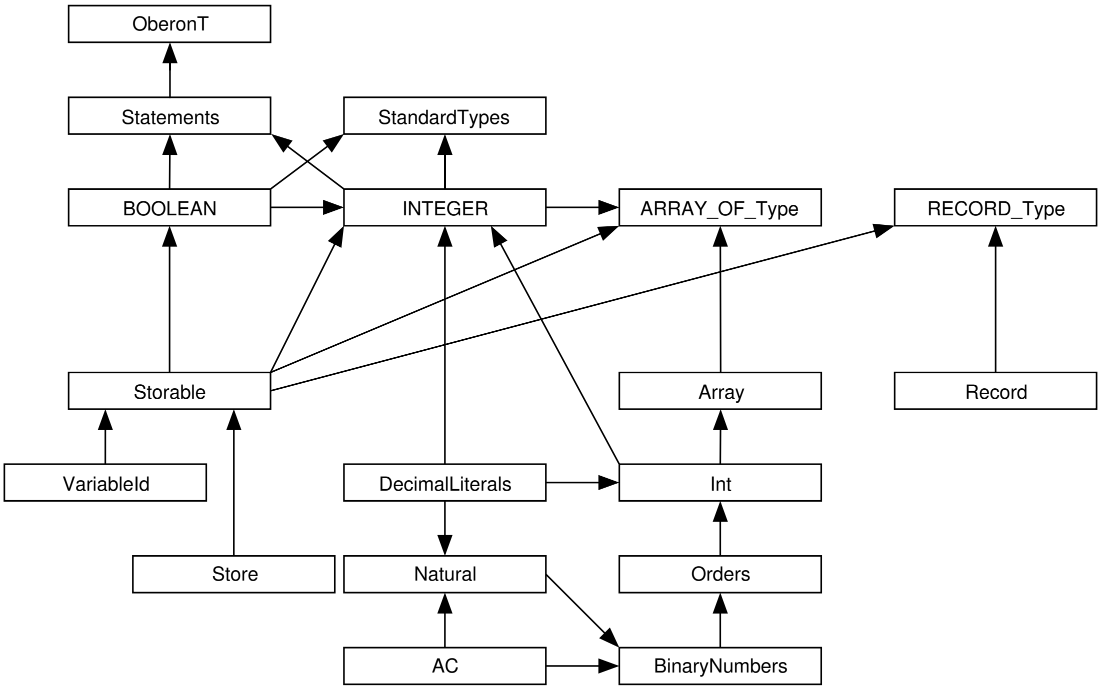

Abbildung 4.12: Hierarchie der LSL-*traits* für die Semantik-Spezifikation

`Natural` spezifiziert die Algebra natürlicher Zahlen.

`AC` spezifiziert die Algebra assoziativ-kommutativer Operatoren.

`DecimalLiterals` spezifiziert die Algebra dezimaler Literale (`1 == s(0); 2 == s(1); ...`).

`VariableId` legt fest, dass jeweils zwei verschiedene Bezeichner für eine Variable auch verschiedene Adressen im Speicher bezeichnen.

Literaturverzeichnis

[All86] L. Allison. *A practical introduction to denotational semantics.* Number 23 in Cambridge Computer Science Texts. Cambridge University Press, 1986.

[Bau90] B. Baumgarten. *Petri-Netze.* BI - Wiss. - Verlag, 1990.

[BHK89] J. A. Bergstra, J. Heering, and P. Klint, editors. *Algebraic Specification.* Addison-Wesley, 1989.

[DS96] C. Denzler and D. Schweizer. PESCA: Programming Environment for Safety Critical Applications. http://www.tik.ee.ethz.ch/˜pesca/, 1996.

[EGL89] H. D. Ehrich, M. Gogolla, and U. W. Lipeck. *Algebraische Spezifikation abstrakter Datentypen.* Teubner, 1989.

[Gan83] H. Ganzinger. Denotational semantics for languages with modules. In D. Bjorner, editor, *Formal description of programming concepts.* North-Holland, 1983.

[GH78] J. V. Guttag and J. J. Horning. The algebraic specification of abstract data types. *Acta Informatica*, 10:27–52, 1978.

[GH93] J. V. Guttag and J. J. Horning, editors. *Larch: Languages and Tools for Formal Specification.* Texts and Monographs in Computer Science. Springer-Verlag, 1993. With Stephen J. Garland, Kevin D. Jones, Andrés Modet, and Jeannette M. Wing.

[GM89] J. A. Goguen and J. Meseguer. Order-Sorted Algebra I. CSL Technical Report SRI-CSL-89-10, SRI International, Computer Science Laboratory, July 1989.

[GPG81] J. A. Goguen and K. Parsaye-Ghomi. Algebraic denotational semantics using parameterized abstract modules. In J. Dias and I. Ramos, editors, *Formalization of programming concepts*, LNCS 107. Springer-Verlag, 1981.

[Gri81] D. Gries. *The science of programming.* Texts and Monographs in Computer Science. Springer-Verlag, 1981.

[GTW78] J. A. Goguen, J. W. Thatcher, and E. G. Wagner. An initial algebra approach to the specification, correctness and implementation of abstract data types. In R. Yeh, editor, *Current Trends in Programming Methodology IV: Data Structuring.* Prentice-Hall, 1978.

[GWMJ92] J. A. Goguen, T. Winkler, J. Meseguer, and J.-P. Jouannaud. Introducing OBJ. CSL Technical Report SRI-CSL-92-03, SRI International, Computer Science Laboratory, March 1992.

[Hel95] A. Helbling. Spezifizieren und Generieren von integrierten Umgebungen mit GIPSY. TIK-Report 12, Institut TIK, ETH Zürich, http://www.tik.ee.ethz.ch, May 1995.

[Hoa72] C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1:271–281, 1972.

[Hoa87] C. A. R. Hoare. An overview of some formal methods for program design. *IEEE Computer*, pages 85–91, September 1987.

[How95] D. Howe. On-line dictionary of computing. http://wombat.doc.ic.ac.uk/, May 1995.

[Inm88] INMOS Limited. *Transputer Reference Manual*, 1988.

[Mar94] R. Marti. *GIPSY: Ein Ansatz zum Entwurf integrierter Softwareentwicklungssysteme.* PhD thesis, Institut TIK, ETH Zürich, 1994. Diss. ETH No. 10463.

[MoD91] UK Ministry of Defence. The Procurement of Safety Critical Software in Defence Equipment, Part1: Requirements. In *Def Stan 00-55 (Part1) / Issue 1.* Directorate of Standards, 1991.

[Par90] H. Partsch. *Specification and Transformation of Programs.* Texts and Monographs in Computer Science. Springer-Verlag, 1990.

[PBB+82] P. Pepper, M. Broy, F. L. Bauer, H. Partsch, W. Dosch, and M. Wirsing. Abstrakte Datentypen: Die algebraische Spezifikation von Rechenstrukturen. *Informatik-Spektrum*, 5:107–119, 1982.

[Rei86] W. Reisig. *Petrinetze.* Springer-Verlag, 1986.

[Rei92] M. Reiser. *Programming in Oberon.* ACM Press, 1992.

[Rus95] J. Rushby. Formal Methods and their Role in the Certification of Critical Systems. CSL Technical Report CSL-95-1, SRI International, Computer Science Laboratory, March 1995.

[Sch] D. Schweizer. *Ein neuer Ansatz zur Verifikation von Programmen für sicherheitskritische Systeme*. PhD thesis, Institut TIK, ETH Zürich. in preparation.

[SK95] K. Slonneger and B. L. Kurtz. *Formal Syntax and Semantics of Programming Languages*. Addison-Wesley, 1995.

[Sta90] P. H. Starke. *Analyse von Petri-Netz-Modellen*. Leitfäden und Monographien der Informatik. Teubner, 1990.

[TM92] R. H. Thayer and A. D. McGettrick, editors. *Software Engineering*. IEEE Computer Society Press, 1992.

[Wir91] M. Wirsing. Algebraic Specification: Semantics, Parameterization and Refinement. In E. J. Neuhold and M. Paul, editors, *Formal Description of Programming Concepts*, IFIP State-of-the-Art Reports, pages 259–318. Springer-Verlag, 1991.

Anhang A

Syntax-Definitionen

A.1 $\mathsf{Oberon}_\mathsf{T}$-Grammatik

A.1.1 Scanner-Spezifikation

```
SCANNER OberonT;

CHARACTERS
eofC = {0X}.
tabC = {09X}.
eolC = {0DX}.
blankC = {020X}.
any = {0X..0FFX} - tabC - eofC.
letter = {'A'..'Z','a'..'z'}.
digit  = {'0'..'9'}.
binaryDigit = {'0','1'}.
hexDigit = digit + {'A'..'F'}.
noQuote1 = any - {"'"} - eolC.
noQuote2 = any - {'"'} - eolC.

IGNORE
blankC

TOKENS
eof = eofC.
tab = tabC.
eol = eolC.
ident  = letter {letter | digit}.
integer = digit {digit} |digit {digit} CONTEXT ('..') |digit {hexDigit} 'H' | binaryDigit {binaryDigit} 'B'.
char = "'" noQuote1 "'" | '"' noQuote2 '"' | digit {hexDigit} 'X'.
string = "'"[noQuote1 noQuote1 {noQuote1}]"'" | '"'[noQuote2 noQuote2 {noQuote2}]'"'.

COMMENT FROM '(*' TO '*)' NESTED

LITERALS
'MODULE', 'IMPORT', 'BEGIN', 'END', ':=', 'CONST', 'TYPE', 'VAR', 'PROCEDURE', '=', '.', ',',
'OF', 'ARRAY', 'RECORD', '[', ']', '(', ')', ';', '-', 'TO', ':', '*', 'DO',
'RETURN', 'IF', 'THEN', 'ELSE', '|', '..', 'FOR', 'BY',
'WITH', '#', '<', '<=', '>', '>=', 'IN', '+','OR', 'DIV', 'MOD', '&', '~', '{', '}'

END OberonT
```

A.1.2 EBNF von $\mathsf{Oberon}_\mathsf{T}$

```
GRAMMAR OberonT;

SCANNER OberonTScanner;

SKIP
eol, tab;

NONTERMINAL Module
::= 'MODULE' ident ';' ['IMPORT' Import Imports ';'] DeclSeq1 'END' ident '.'.

NONTERMINAL Imports
::= [',' Import Imports].

NONTERMINAL Import
::= ident.

NONTERMINAL DeclSeq1
::= DeclParts1 ProcedureDecls.

NONTERMINAL DeclSeq2
::= DeclParts2.

NONTERMINAL DeclParts1
::= ['CONST' ConstDeclarations] ['TYPE' TypeDeclarations].

NONTERMINAL DeclParts2
::= ['VAR' VarDeclarations].

NONTERMINAL ConstDeclarations
::= [ConstDeclaration ';' ConstDeclarations].

NONTERMINAL TypeDeclarations
::= [TypeDeclaration ';' TypeDeclarations].

NONTERMINAL VarDeclarations
::= [VarDeclaration ';' VarDeclarations].

NONTERMINAL ProcedureDecls
::= ['PROCEDURE' ProcedureDecl ProcedureDecls].

NONTERMINAL ConstDeclaration
::= IdentDef '=' ConstExpression.

NONTERMINAL VarDeclaration
::= IdentList ':' TypeQualIdent.

NONTERMINAL IdentList
::= IdentDef [',' IdentList].

NONTERMINAL ConstExpression
::= Expression.

NONTERMINAL TypeDeclaration
::= IdentDef '=' Type.

NONTERMINAL Type
::= TypeQualIdent | ArrayType | RecordType.

NONTERMINAL TypeQualIdent
::= Object ['.' ident].

NONTERMINAL ArrayType
::= 'ARRAY' [ConstExpressions] 'OF' Type.
```

```
NONTERMINAL ConstExpressions
::= ConstExpression [',' ConstExpressions].

NONTERMINAL RecordType
::= 'RECORD' FieldLists 'END'.

NONTERMINAL FieldLists
::= FieldList [';' FieldLists].

NONTERMINAL FieldList
::= [Fields ':' Type].

NONTERMINAL Fields
::= IdentDef [',' Fields].

NONTERMINAL ProcedureDecl
::= ProcedureHeading ';' ProcedureBody ident ';'.

NONTERMINAL ProcedureHeading
::= ['*'] IdentDef FormalParameters.

NONTERMINAL ProcedureBody
::= DeclSeq2 ['BEGIN' StatSeq] 'END'.

NONTERMINAL FormalParameters
::= '(' [FPSections] ')' ':' TypeQualIdent.

NONTERMINAL FPSections
::= FPSection [';' FPSections].

NONTERMINAL FPSection
::= Idents ':' TypeQualIdent.

NONTERMINAL Idents
::= ident [',' Idents].

NONTERMINAL StatSeq
::= Statement [';' StatSeq].

NONTERMINAL Statement
::= [ AssOrCall
| IfStatement
| ForStatement
| 'RETURN' Expression
].

NONTERMINAL AssOrCall
::= Designator [ ':=' Expression].

NONTERMINAL IfStatement
::= 'IF' Expression 'THEN' StatSeq
['ELSE' StatSeq]
'END'.

NONTERMINAL ForStatement
::= 'FOR' ident ':=' Expression 'TO' Expression 'DO' StatSeq 'END'.

NONTERMINAL Expression
::= SimpleExpr [RelOp SimpleExpr].

NONTERMINAL RelOp
::= '=' | '#' | '<' | '<=' | '>' | '>='.

NONTERMINAL SimpleExpr
```

```
::=  ['+' | '-'] AddOpExpr.

NONTERMINAL AddOpExpr
::=  Term [AddOp AddOpExpr].

NONTERMINAL AddOp
::= '+' | '-' | 'OR'.

NONTERMINAL Term
::= Factor [MulOp Term].

NONTERMINAL MulOp
::= '*' | 'DIV' | 'MOD' | '&'.

NONTERMINAL Factor
::= '(' Expression ')'
| '~' Factor
| Designator
| integer
| char
| string.

NONTERMINAL Designator
::= Object Extensions.

NONTERMINAL Extensions
::= [('.' ident
| '[' Expression ']'
| '(' [Expression Expressions] ')')
Extensions ].

NONTERMINAL Expressions
::= [',' Expression Expressions].

NONTERMINAL IdentDef
::= ident ['*'|'-'].

NONTERMINAL Object
::= ident.

END OberonT
```

A.1.3 Signatur der konkreten Syntax

```
OberonTTerminals:trait
introduces
  ident:→ Ident
  integer:→ Integer
  char:→ Char
  string:→ String
  eq,neq,lower,leq,greater,geq:→ RelOp
  plus,minus,or:→ AddOp
  times,div,mod,and:→ MulOp
  plus,minus:→ Sign
asserts
  sort Ident generated freely by
    ident:→ Ident
  sort Integer generated freely by
    integer:→ Integer
  sort Char generated freely by
    char:→ Char
  sort String generated freely by
    string:→ String
  sort RelOp generated freely by
    eq,neq,lower,leq,greater,geq
  sort AddOp generated freely by
    plus,minus,or
  sort MulOp generated freely by
    times,div,mod,and
  sort Sign generated freely by
    plus,minus
```

```
OberonTSyntax:trait
includes
  OberonTTerminals
introduces
  module:Ident,Import,Imports,DeclSeq1,Ident → Module
  module:Ident,DeclSeq1,Ident → Module
  imports:Import,Imports → Imports
  imports:→ Imports
  import:Ident → Import
  declSeq1:DeclParts1,ProcedureDecls → DeclSeq1
  declSeq2:DeclParts2 → DeclSeq2
  declParts1:→ DeclParts1
  declParts1:ConstDeclarations → DeclParts1
  declParts1:TypeDeclarations → DeclParts1
  declParts1:ConstDeclarations,TypeDeclarations → DeclParts1
  declParts2:→ DeclParts2
  declParts2:VarDeclarations → DeclParts2
  constDeclarations:→ ConstDeclarations
  constDeclarations:ConstDeclaration,ConstDeclarations → ConstDeclarations
  typeDeclarations:→ TypeDeclarations
  typeDeclarations:TypeDeclaration,TypeDeclarations → TypeDeclarations
  varDeclarations:→ VarDeclarations
  varDeclarations:VarDeclaration,VarDeclarations → VarDeclarations
  procedureDecls:→ ProcedureDecls
  procedureDecls:ProcedureDecl,ProcedureDecls → ProcedureDecls
  constDeclaration:IdentDef,ConstExpression → ConstDeclaration
  varDeclaration:IdentList,TypeQualIdent → VarDeclaration
  identList:IdentDef → IdentList
  identList:IdentDef,IdentList → IdentList
  constExpression:Expression → ConstExpression
  typeDeclaration:IdentDef,Type → TypeDeclaration
  type:TypeQualIdent → Type
  type:ArrayType → Type
  type:RecordType → Type
```

```
  typeQualIdent:Object → TypeQualIdent
  typeQualIdent:Object,Ident → TypeQualIdent
  arrayType:Type → ArrayType
  arrayType:ConstExpressions,Type → ArrayType
  constExpressions:ConstExpression → ConstExpressions
  constExpressions:ConstExpression,ConstExpressions → ConstExpressions
  recordType:FieldLists → RecordType
  fieldLists:FieldList → FieldLists,
  fieldLists:FieldList,FieldLists → FieldLists
  fieldList:→ FieldList,
  fieldList:Fields,Type → FieldList
  fields:IdentDef → Fields,
  fields:IdentDef,Fields → Fields
  procedureDecl:ProcedureHeading,ProcedureBody,Ident → ProcedureDecl
  procedureHeading:IdentDef,FormalParameters → ProcedureHeading
  procedureBody:DeclSeq2 → ProcedureBody
  procedureBody:DeclSeq2,StatSeq → ProcedureBody
  formalParameters:TypeQualIdent → FormalParameters
  formalParameters:FPSections,TypeQualIdent → FormalParameters
  fPSections:FPSection → FPSections
  fPSections:FPSection,FPSections → FPSections
  fPSection:Idents,TypeQualIdent → FPSection
  idents:Ident → Idents
  idents:Ident,Idents → Idents
  statSeq:Statement → StatSeq
  statSeq:Statement,StatSeq → StatSeq
  statement:→ Statement
  statement:AssOrCall → Statement
  statement:IfStatement → Statement
  statement:ForStatement → Statement
  statement:Expression → Statement
  assOrCall:Designator → AssOrCall
  assOrCall:Designator,Expression → AssOrCall
  ifStatement:Expression,StatSeq,StatSeq → IfStatement
  ifStatement:Expression,StatSeq → IfStatement
  forStatement:Ident,Expression,Expression,StatSeq → ForStatement
  expression:SimpleExpr → Expression
  expression:SimpleExpr,RelOp,SimpleExpr → Expression
  simpleExpr:AddOpExpr → SimpleExpr
  simpleExpr:Sign,AddOpExpr → SimpleExpr
  addOpExpr:Term → AddOpExpr
  addOpExpr:Term,AddOp,AddOpExpr → AddOpExpr
  term:Factor → Term
  term:Factor,MulOp,Term → Term
  factor:Expression → Factor
  factor:Factor → Factor
  factor:Designator → Factor
  factor:Integer → Factor
  factor:Char → Factor
  factor:String → Factor
  designator:Object,Extensions → Designator
  extensions:→ Extensions
  extensions:Ident,Extensions → Extensions
  extensions:Expression,Expressions,Extensions→ Extensions
  extensions:Extensions → Extensions
  expressions:→ Expressions
  expressions:Expression,Expressions → Expressions
  identDef:Ident → IdentDef
  object:Ident → Object
asserts
  sort Module generated freely by
    module:Ident,Import,Imports,DeclSeq1,Ident → Module,
    module:Ident,DeclSeq1,Ident → Module
  sort Imports generated freely by
    imports:Import,Imports → Imports,
```

```
  imports:→ Imports
sort Import generated freely by
  import:Ident → Import
sort DeclSeq1 generated freely by
  declSeq1:DeclParts1,ProcedureDecls → DeclSeq1
sort DeclSeq2 generated freely by
  declSeq2:DeclParts2 → DeclSeq2
sort DeclParts1 generated freely by
  declParts1:→ DeclParts1,
  declParts1:ConstDeclarations → DeclParts1,
  declParts1:TypeDeclarations → DeclParts1,
  declParts1:ConstDeclarations,TypeDeclarations → DeclParts1
sort DeclParts2 generated freely by
  declParts2:→ DeclParts2,
  declParts2:VarDeclarations → DeclParts2
sort ConstDeclarations generated freely by
  constDeclarations:→ ConstDeclarations,
  constDeclarations:ConstDeclaration,ConstDeclarations → ConstDeclarations
sort TypeDeclarations generated freely by
  typeDeclarations:→ TypeDeclarations,
  typeDeclarations:TypeDeclaration,TypeDeclarations → TypeDeclarations
sort VarDeclarations generated freely by
  varDeclarations:→ VarDeclarations,
  varDeclarations:VarDeclaration,VarDeclarations → VarDeclarations
sort ProcedureDecls generated freely by
  procedureDecls:→ ProcedureDecls,
  procedureDecls:ProcedureDecl,ProcedureDecls → ProcedureDecls
sort ConstDeclaration generated freely by
  constDeclaration:IdentDef,ConstExpression → ConstDeclaration
sort ConstExpression generated freely by
  constExpression:Expression → ConstExpression
sort TypeDeclaration generated freely by
  typeDeclaration:IdentDef,Type → TypeDeclaration
sort Type generated freely by
  type:TypeQualIdent → Type,
  type:ArrayType → Type,
  type:RecordType → Type
sort TypeQualIdent generated freely by
  typeQualIdent:Object → TypeQualIdent,
  typeQualIdent:Object,Ident → TypeQualIdent
sort ArrayType generated freely by
  arrayType:Type → ArrayType,
  arrayType:ConstExpressions,Type → ArrayType
sort ConstExpressions generated freely by
  constExpressions:ConstExpression → ConstExpressions,
  constExpressions:ConstExpression,ConstExpressions → ConstExpressions
sort RecordType generated freely by
  recordType:FieldLists → RecordType
sort FieldLists generated freely by
  fieldLists:FieldList → FieldLists,
  fieldLists:FieldList,FieldLists → FieldLists
sort FieldList generated freely by
  fieldList:→ FieldList,
  fieldList:Fields,Type → FieldList
sort Fields generated freely by
  fields:IdentDef → Fields,
  fields:IdentDef,Fields → Fields
sort ProcedureDecl generated freely by
  procedureDecl:ProcedureHeading,ProcedureBody,Ident → ProcedureDecl
sort ProcedureHeading generated freely by
  procedureHeading:IdentDef,FormalParameters → ProcedureHeading
sort ProcedureBody generated freely by
  procedureBody:DeclSeq2 → ProcedureBody,
  procedureBody:DeclSeq2,StatSeq → ProcedureBody
sort FormalParameters generated freely by
```

```
  formalParameters:TypeQualIdent → FormalParameters,
  formalParameters:FPSections,TypeQualIdent → FormalParameters
sort FPSections generated freely by
  fPSections:FPSection → FPSections,
  fPSections:FPSection,FPSections → FPSections
sort FPSection generated freely by
  fPSection:Idents,TypeQualIdent → FPSection
sort Idents generated freely by
  idents:Ident → Idents,
  idents:Ident,Idents → Idents
sort StatSeq generated freely by
  statSeq:Statement → StatSeq,
  statSeq:Statement,StatSeq → StatSeq
sort Statement generated freely by
  statement:→ Statement,
  statement:AssOrCall → Statement,
  statement:IfStatement → Statement,
  statement:ForStatement → Statement,
  statement:Expression → Statement
sort AssOrCall generated freely by
  assOrCall:Designator → AssOrCall,
  assOrCall:Designator,Expression → AssOrCall
sort IfStatement generated freely by
  ifStatement:Expression,StatSeq,StatSeq → IfStatement,
  ifStatement:Expression,StatSeq → IfStatement
sort ForStatement generated freely by
  forStatement:Ident,Expression,Expression,StatSeq → ForStatement
sort Expression generated freely by
  expression:SimpleExpr → Expression,
  expression:SimpleExpr,RelOp,SimpleExpr → Expression
sort SimpleExpr generated freely by
  simpleExpr:AddOpExpr → SimpleExpr,
  simpleExpr:Sign,AddOpExpr → SimpleExpr
sort AddOpExpr generated freely by
  addOpExpr:Term → AddOpExpr,
  addOpExpr:Term,AddOp,AddOpExpr → AddOpExpr
sort Term generated freely by
  term:Factor → Term,
  term:Factor,MulOp,Term → Term
sort Factor generated freely by
  factor:Expression → Factor,
  factor:Factor → Factor,
  factor:Designator → Factor,
  factor:Integer → Factor,
  factor:Char → Factor,
  factor:String → Factor
sort Designator generated freely by
  designator:Object,Extensions → Designator
sort Extensions generated freely by
  extensions:→ Extensions,
  extensions:Ident,Extensions → Extensions,
  extensions:Expression,Expressions,Extensions→ Extensions,
  extensions:Extensions → Extensions
sort Expressions generated freely by
  expressions:→ Expressions,
  expressions:Expression,Expressions → Expressions
sort IdentDef generated freely by
  identDef:Ident → IdentDef
sort Object generated freely by
  object:Ident → Object
```

Anhang B

Oberon$_\top$ Semantik-Spezifikationen

B.1 Boolesche Algebra

B.1.1 Semantischer Bereich

```
Boolean: trait
introduces
  true, false: → Bool
  ¬__: Bool → Bool
  __∧ __, __∨__, __⇒__: Bool, Bool → Bool
asserts
  Bool generated by true, false
  ∀ b: Bool
    ¬ true == false;
    ¬ false == true;
    true ∧ b == b;
    false ∧ b == false;
    true ∨ b == true;
    false ∨ b == b;
    true ⇒ b == b;
    false ⇒ b == true
implies
  AC (∧ , Bool),
  AC (∨, Bool),
  Distributive (∨ for +, ∧ for *, Bool for T),
  Distributive (∧ for +, ∨ for *, Bool for T),
  Involutive (¬__, Bool),
  Transitive (⇒ for ◇, Bool for T)
  ∀ b1, b2, b3: Bool
    ¬(b1 ∧ b2) == ¬b1 ∨ ¬b2;
    ¬(b1 ∨ b2) == ¬b1 ∧ ¬b2;
    b1 ∨ (b1 ∧ b2) == b1;
    b1 ∧ (b1 ∨ b2) == b1;
    b2 ∨ ¬b2;
    (b1 = b2) ∨ (b1 = b3) ∨ (b2 = b3);
    b1 ⇒ b2 == ¬b1 ∨ b2
```

B.2 Algebra ganzer Zahlen

B.2.1 Semantischer Bereich

```
Int(Int):trait
includes
  DecimalLiterals(s for succ,Int for N),
  Orders(Int)


Orders(B):trait
includes
  BinaryNumbers(B)
introduces
  __<__,__<=__,__>__,__>=__:B,B → Bool
  inRange,range:B,B,B → Bool
asserts
  ∀x,y,z:B,n:N,i,j:Digit,a,b:Bool
  x < x == false;
  x < y == isNeg(x - y);
  x <= y == ¬(y < x);
  x > y == y < x;
  x >= y == ¬(x < y);
  inRange(x,y,z) == x <= y ∧ y <= z;
  range(x,y,z) == if isNeg(z - x) then false else
    range(nat(0),nat(y - x),nat(z - x));
implies
  ∀x,y,z,x1,z1:B,n:N
  range(x,y,z) => inRange(x,y,z);
  (x1 <= x ∧ z <= z1) => (inRange(x ,y,z) => inRange(x1 ,y,z1));
  converts
    __<__:B,B → Bool,
    __<=__:B,B → Bool,
    __>__:B,B → Bool,
    __>=__:B,B → Bool,
    inRange,range:B,B,B → Bool


BinaryNumbers(B):trait
includes
  AC(+,B),
  AC(*,B),
  Natural(N)
introduces
  0,1:→ Digit
  0,1:→ B
  __.__:B,Digit → B
  s:B → B
  __+__,__*__,__-__,div,mod:B,B → B
  -__:B → B
  isZero,isNeg:B → Bool
  isNeg:B,B → Bool
  inv:B → B
  compl:B → B
  abs:B → B
  nat:B → N
  int:N → B
asserts
  sort Digit generated freely by 0,1
  sort B generated by 0,1,__.__
  ∀x,y:B,n:N,i:Digit
  s(0) == 0.1;
  s(1) == 0;
```

```
  s(x.0) == x.1;
  s(x.1) == s(x).0;
  x + 0 == x;
  l + l == l.0;
  x.0 + l == (x + l).1;
  x.1 + l == x.0;
  x.i + y.0 == (x + y) . i;
  x.1 + y.1 == s(x + y) . 0;
  x * 0 == 0;
  x * l == compl(x);
  x * y.0 == (x*y).0;
  x * y.1 == ((x*y).0) + x;
  -x == compl(x);
  x - y == x + compl(y);
  isZero(x) == x = 0;
  isNeg(0) == false;
  isNeg(l);
  isNeg(x.0) == isNeg(x);
  isNeg(x.1) == isNeg(x);
  isNeg(x,y) == (isNeg(x) ∧ ¬isNeg(y)) ∨ (¬isNeg(x) ∧ isNeg(y));
  inv(0) == l;
  inv(l) == 0;
  inv(x.0) == inv(x).1;
  inv(x.1) == inv(x).0;
  compl(x) == s(inv(x));
  0:B = l == false;
  0.0 == 0;
  0 = x.1 == false;
  l = x.0 == false;
  l.1 == l;
  x.0 = y.1 == false;
  x.1 = y.0 == false;
  x.0 = y.0 == x = y;
  x.1 = y.1 == x = y;
  nat(0) == 0;
  nat(x.0) == 2 * nat(x);
  nat(x.1) == (2 * nat(x)) + 1;
  int(nat(x)) == x;
  int(0) == 0;
  int(s(n)) == s(int(n));
  nat(int(n)) == n;
  abs(x) == if isNeg(x) then -x else x;
  div(x,y) == if isNeg(x,y) then -int(div(nat(abs(x)),nat(abs(y))))
    else int(div(nat(abs(x)),nat(abs(y))));
  mod(x,y) == int(mod(nat(abs(x)),nat(abs(y))));
implies
  ∀x,y,z:B,n:N,i,j:Digit,a,b:Bool
  inv(inv(x)) = x;
  compl(compl(x)) = x;
  nat(x) = n == x = int(n);
  x - y = 0 == x = y;
  x + y.i = z.j == x = z.j - y.i;
  x + y.i = 0 == x = -y.i;
  a ∨ b == ¬(¬a ∧ ¬b);
  -(x + y) == (-x) + (-y);
  x + z = y + z == x = y;
  x - z = y - z == x = y;
  converts
    isZero,isNeg:B → Bool,isNeg:B,B → Bool,inv,compl,abs,
    s:B → B,-__:B → B,
    __+__:B,B → B,__*__:B,B → B,__-__:B,B → B,
    div:B,B → B,mod:B,B → B,
    nat,int
    exempting nat(l)
```

```
Natural(N):trait
includes
  DecimalLiterals(s for succ),
  AC(+,N),
  AC(*,N)
introduces
  0:→ N
  s:N → N
  __+__,__-__,__*__:N,N → N
  div,mod:N,N → N
  __<__,__<=__,__>__ ,__>=__:N,N → Bool
  range:N,N,N → Bool
asserts
  N generated freely by 0,s
  ∀x,y,i:N
  x + 0 == x;
  x + s(y) == s(x + y);
  x - 0 == x;
  s(x) - s(y) == x - y;
  x * 0 == 0;
  x * s(y) == x + (x * y);
  s(x) = s(y) == x = y;
  0 = s(x) == false;
  s(x) = 0 == false;
  x < 0 == false;
  0 < s(x);
  s(x) < s(y) == x < y;
  x <= y == ¬(y < x);
  x > y == y < x;
  x >= y == ¬(x < y);
  range(0,i,0) == i = 0;
  range(0,i,s(x)) == ¬(i ≠ s(x) ∧ ¬range(0,i,x));
  range(s(x),i,0) == ¬(i ≠ s(x) ∧ ¬range(x,i,0));
  range(s(x),i,s(y)) == ¬(i ≠ s(x) ∧ i ≠ s(y) ∧ ¬range(x,i,y));
  div(0,s(x)) == 0;
  div(s(x),s(y)) == if s(x) < s(y) then 0 else s(div(x - y,s(y)));
  mod(0,s(x)) == 0;
  mod(s(x),s(y)) == if s(x) < s(y) then s(x) else mod(x - y,s(y));
implies
  converts
    __+__,__-__,__*__:N,N → N,
    __<__,__<=__,__>__ ,__>=__:N,N → Bool,
    range:N,N,N → Bool,
    div,mod
    exempting ∀x:N 0 - x,div(x,0),mod(x,0)
```

B.2.2 Semantische Funktion

```
INTEGER:trait
includes
  BOOLEAN,
  Int(Int),
  Storable(INTEGER,Int)
introduces
  range:INTEGER,INTEGER,INTEGER → BOOLEAN
  MAXINT,MININT:→ Int
asserts
  ∀j,j1,j2:INTEGER,s:Store
  MAXINT == 0.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1;
  MININT == 1.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0;
  ⟦0⟧[s,true] == 0;
  ⟦1⟧[s,true] == 1;
```

```
⟦j1 + j2⟧[s,true] == if inRange(MININT,⟦j1⟧[s,true] + ⟦j2⟧[s,true],MAXINT) then
  ⟦j1⟧[s,true] + ⟦j2⟧[s,true] else ⟦j1 + j2⟧[err,false];
⟦j1 - j2⟧[s,true] == if inRange(MININT,⟦j1⟧[s,true] - ⟦j2⟧[s,true],MAXINT) then
  ⟦j1⟧[s,true] - ⟦j2⟧[s,true] else ⟦j1 - j2⟧[err,false];
⟦j1 * j2⟧[s,true] == if inRange(MININT,⟦j1⟧[s,true] * ⟦j2⟧[s,true],MAXINT) then
  ⟦j1⟧[s,true] * ⟦j2⟧[s,true] else ⟦j1 * j2⟧[err,false];
⟦j1 < j2⟧[s,true] == ⟦j1⟧[s,true] < ⟦j2⟧[s,true];
⟦j1 <= j2⟧[s,true] == ⟦j1⟧[s,true] <= ⟦j2⟧[s,true];
⟦j1 > j2⟧[s,true] == ⟦j1⟧[s,true] > ⟦j2⟧[s,true];
⟦j1 >= j2⟧[s,true] == ⟦j1⟧[s,true] >= ⟦j2⟧[s,true];
⟦j1 =* j2⟧[s,true] == ⟦j1⟧[s,true] = ⟦j2⟧[s,true];
⟦j1 # j2⟧[s,true] == ⟦j1⟧[s,true] ≠ ⟦j2⟧[s,true];
⟦DIV(j1,j2)⟧[s,true] == if(⟦j2⟧[s,true] ≠ 0) then
  div(⟦j1⟧[s,true],⟦j2⟧[s,true]) else ⟦DIV(j1,j2)⟧[err,false];
⟦MOD(j1,j2)⟧[s,true] == if(⟦j2⟧[s,true] ≠ 0) then
  mod(⟦j1⟧[s,true],⟦j2⟧[s,true]) else ⟦MOD(j1,j2)⟧[err,false];
⟦succ(j1)⟧[s,true] == s(⟦j1⟧[s,true]);
⟦-j1⟧[s,true] == if MININT ≠ ⟦j1⟧[s,true] then
  -(⟦j1⟧[s,true]) else ⟦-j1⟧[err,false];
⟦range(j1,j,j2)⟧[s,true] == range(⟦j1⟧[s,true],⟦j⟧[s,true],⟦j2⟧[s,true]);
```

B.2.3 Diverse

```
OberonT:trait
includes
  Statements


StandardTypes:trait
includes
  BOOLEAN,
  INTEGER

AC(○,T):trait
  introduces __ ○ __:T,T → T
  asserts ∀x,y,z:T
    (x ○ y) ○ z == x ○ (y ○ z);
    x ○ y == y ○ x
  implies
    Associative,
    Commutative(T for Range)


DecimalLiterals(N):trait
  % A built-in trait schema given here
  % for documentation only
  introduces
    0,1,2,3,4,5,6,7,8,9,10,11 %,...
      :→ N
    succ:N → N
  asserts equations
    1 == succ(0);
    2 == succ(1);
    3 == succ(2);
  % ... as far as needed for any literals
  % of sort N appearing in the including trait

VariableId(A):trait
introduces
  var:L → Bool
  uneq:A,L → Bool
```

```
  __|__:A,L → L
  []:→ L
asserts
  sort L generated freely by [],|
  ∀a,b:A,h:L
  var(a | h) == uneq(a,h) ∧ var(h);
  uneq(a,b | h) == (a ≠ b) ∧ uneq(a,h);
  uneq(a,[];
  var[];
implies
  converts
    var,uneq
```