Konzeption und Implementierung eines Interpreters für die sprachenübergreifende Programmrepräsentation IML

Diplomarbeit

Philippe Maurice Schober
Matrikelnummer: 1441621

14.05.2007

Fachbereich 3: Mathematik / Informatik
Studiengang Informatik

1. Gutachter: Prof. Dr. Rainer Koschke
2. Gutachter: Prof. Dr. Jan Peleska

Erklärung

Ich versichere, die Diplomarbeit ohne fremde Hilfe angefertigt zu haben. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen sind, sind als solche kenntlich gemacht.

Bremen, den 14.05.2007

.....
(Philippe Maurice Schober)

Danksagung

Danken möchte ich Iris Safo, die mir während der sechs Monate, in der ich diese Arbeit geschrieben habe, beigestanden hat, um auch die stressigsten Situationen heil zu überstehen. Desweiteren möchte ich meinen Eltern, Barbara und Joachim Schober, danken, ohne die das Studium nicht möglich gewesen wäre.

Mein Dank gilt auch Rainer Koschke, der bereitwillig alle meine Fragen bezüglich des Bauhaus-Projektes und der IML ausführlich beantwortet hat. Desweiteren möchte ich Jan Peleska dafür danken, daß er sich so kurzfristig dafür bereiterklärt hat, als Zweitgutachter zur Verfügung zu stehen.

INHALTSVERZEICHNIS

KAPITEL 1

Einführung

Das Ziel der Arbeit ist die Konzeption und Entwicklung eines Interpreters für die Programmrepräsentation *Intermediate Modelling Language* (IML), die ein Teil des Bauhaus-Projektes darstellt. Die IML wird aus Quellcode generiert und ursprünglich als Grundlage für Analysen verwendet. Durch die Interpretation der IML soll dessen Korrektheit verifiziert werden, so daß sowohl die IML-Generierung als auch dessen Optimierung geprüft werden kann.

1.1 Das Bauhaus-Projekt

Bei dem *Bauhaus*-Projekt[1] handelt es sich um ein Forschungsprojekt, das 1996 von der Universität Stuttgart und dem Fraunhofer Institut für Experimentelles Software Engineering in Kaiserslautern[2] gegründet wurde. Ziel des Projektes ist die Analyse und Wartung bestehenden Quelltextes beliebiger Komplexität zu vereinfachen. Die Software ist in der Lage unnötigen Code zu finden, Zusammenhänge grafisch darzustellen und zahlreiche andere Operationen durchzuführen, die den Nutzern helfen sollen, Programme anhand des Quelltextes zu verstehen.

Um die Analysen durchführen zu können, wird der Quellcode in eine Zwischensprache umgewandelt, die einem abstrakten Semantikgraphen gleicht. Die Sprache des Graphen heißt *Intermediate Modelling Language*, durch dessen Generierung die durchgeführten Analysen unabhängig von der ursprünglichen Programmiersprache sind.

Zur Erzeugung dieser IML gibt es bereits mehrere Frontends für die jeweiligen unterstützten Sprachen. Für C wird `cafe`[11] verwendet, für C++ `cafe++`[15], für Java `jafe`[15] und für Ada95 `ada2iml`[6].

Desweiteren ist es möglich aus der IML die Zwischendarstellung *Resource Flow Graph* (RFG) zu erzeugen. Diese wird hauptsächlich für Architekturanalysen verwendet und ist daher für die Entwicklung des Interpreters irrelevant.

1.2 Intermediate Modelling Language

Bei der *Intermediate Modelling Language*, kurz IML, handelt es sich um einen abstrakten Semantikgraphen, der unabhängig von der ursprünglichen Programmiersprache ist. Gegenüber abstrakter Syntaxbäume verfügt ein Graph über zusätzliche Verbindungen zwischen den einzelnen Knoten, die den jeweiligen Typ des Knotens als auch weitere Informationen wie zum Beispiel dazugehörige Parameter, Konstruktoren und Definitionen abbilden.

Anhand dieses Graphen können Analysen durchgeführt werden, die mit einem simplen abstrakten Syntaxbaum nicht möglich wären. So kann zum Beispiel die Frage beantwortet werden, wo eine Variable initialisiert wird oder was für einen Typ das Ergebnis einer bestimmten Multiplikation hat.

Deutlicher zu sehen ist dies in den Abbildungen 1.1 und 1.2, die die gleiche Multiplikation jeweils als abstrakten Syntaxbaum und als abstrakten Semantikgraphen darstellen.

Genauer erläutert wird der Aufbau der IML in Kapitel 2 und in Anhang A sind alle relevanten Knoten referenziert und werden anhand von Beispielen erläutert.

1.3 Ziele der Arbeit

Die Aufgaben dieser Arbeit sind:

- Entwicklung eines Konzepts für die Interpretation der IML, das es ermöglicht, sämtliche Variationen der IML auszuführen. Möglichst viele Programmiersprachen, die durch die IML abgebildet werden, sollten unterstützt werden können.

[1] http://www.bauhaus-stuttgart.de/bauhaus/
[2] http://www.iese.fraunhofer.de/fhg/iese_DE/

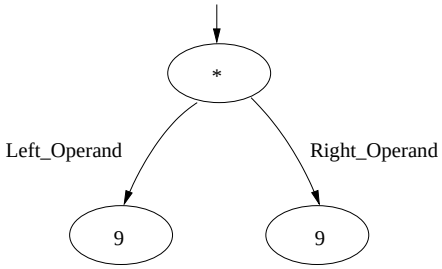

Abbildung 1.1: Abstrakter Syntaxbaum

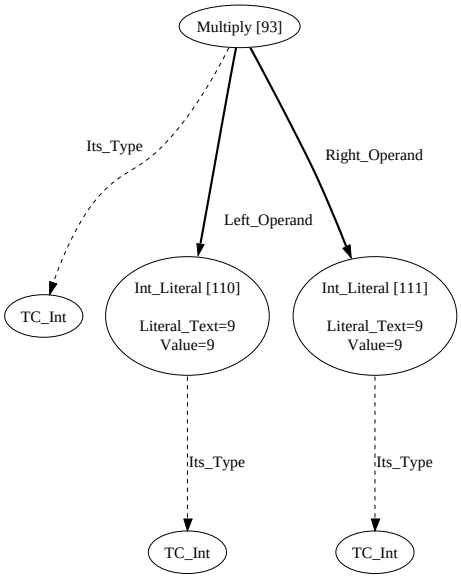

Abbildung 1.2: Abstrakter Semantikgraph

- Implementierung des Interpreters mit Schwerpunkt auf die Unterstützung von IML-Graphen, die aus C-Quellcode generiert wurden. Hierzu gehört die Einbindung möglichst vieler Funktionen aus der *Standard Library*. Desweiteren soll der Interpreter unter Linux entwickelt werden, Kompatibilität mit anderen Betriebssystemen ist zwar wünschenswert aber nicht Ziel der Arbeit.
- Falls technisch möglich die Erweiterung des Interpreters zur Unterstützung von C++ und Objektorientierung.
- Falls technisch möglich die Erweiterung des Interpreters auf die restlichen unterstützten Sprachen, Java1.4[3] und Ada95[4].
- Test der Korrektheit und der Performanz des Interpreters im Vergleich zu den kompilierten Programmen, dessen IML-Darstellungen durch den Interpreter ausgeführt werden.

1.4 Aufbau der Arbeit

In Kapitel 2 wird die *Intermediate Modelling Language* und ihr Verwendungszweck genauer beschrieben. Das dritte Kapitel geht auf den Aufbau des Interpreters und die dabei gefällten Entscheidungen ein. Kapitel 3 beschreibt die unterstützten Sprachen und welche derer Eigenschaften und Standard-Funktionen unterstützt werden. Im Kapitel 3.5 ist beschrieben, wie der Interpreter installiert und ausgeführt wird. Die Ergebnisse der Arbeit bezüglich der Korrektheit und Performanz des Interpreters sind in Kapitel 4 aufgeführt. Ansatzpunkte für weitere Arbeiten sind in Kapitel 4.4 beschrieben.

Im Anhang A befindet sich die IML-Knotenreferenz, in der sämtliche für die Ausführung relevanten Knoten der *Intermediate Modelling Language* aufgeführt sind. Am Ende des Dokuments befindet sich ein Index, über den sich Beschreibungen der einzelnen Methoden und Knoten schnell referenzieren lassen.

1.5 Hinweis an die Leser

Auch wenn in dieser Arbeit versucht wird, die zum Verständnis nötigen Grundlagen detailliert zu erläutern, so ist dies dennoch nicht immer möglich. Daher ist eine Kenntnis der Programmiersprachen C, C++, Java und Ada95 hilfreich. Weiterhin erleichtern Kenntnisse bezüglich der IML und des Bauhaus-Projektes das Verständnis.

[3] http://java.sun.com/j2se/1.4.2/
[4] http://www.adahome.com/rm95/

KAPITEL 2

Intermediate Modelling Language

Die IML ist Teil des Bauhaus-Projektes und wird als Zwischensprache aus Programmquellcode generiert. Sie ist die Grundlage für zahlreiche Analysen und kann desweiteren in einen *Resource Flow Graph* konvertiert werden.

Bei ihr handelt es sich um einen abstrakten Semantikgraphen, der die logische Erweiterung des abstrakten Syntaxbaums darstellt. Während abstrakte Syntaxbäume nur die Struktur eines Programms abbilden, so beinhalten Semantikgraphen wesentlich mehr Informationen. Jedem Knoten sind zahlreiche Kanten angehängt, die nicht nur zu dessen Operanden verweisen, sondern auch dessen Typknoten, eventuelle Initialierungsknoten oder weitere Knoten verweisen, die zusätzliche Aspekte darstellen.

Eine arithmetische Multiplikation besteht zum Beispiel aus einem *Multiply*-Knoten, deren Faktoren über die Kanten *Left_Operand* und *Right_Operand* angesprochen werden können. Sowohl die Knoten auf die über diese Kanten verwiesen wird, als auch der *Multiply*-Knoten verweisen auf einen Typknoten, der bestimmt, um was für eine Multiplikation es sich handelt. Solch ein Beispiel ist genauer am Ende des Kapitels erläutert.

Die einzelnen Knoten gehören jeweils in eine von vier möglichen Kategorien. Es handelt sich bei jedem Knoten entweder um eine *Hierarchical_Unit*, die zum Beispiel Klassen und Methoden repräsentiert, oder um einen *Value*, einer Anweisung beziehungsweise einem Ausdruck. Desweiteren gibt es *Symbol_Node*s, die jeweils entweder einen *O_Node* oder einen *T_Node* darstellt. Während die *T_Node*s die einzelnen Datentypen darstellen, repräsentieren die *O_Node*s die Datenwerte des Programms. Dazu gehören sowohl die Variablen als auch die Parameter einzelner Funktionen.

Jeder *Value*-Knoten verweist auf einen *T_Node* über eine *Its_Type*-Kante, die darstellt, was für eine Wertigkeit er hat. Ein *Multiply*-Knoten hat zum Beispiel als Typ den des Ergebnisses seiner Berechnung (siehe Kapitel 3.4.3).

Alle Knoten verfügen innerhalb des Graphen über eine eindeutige ID, ihrem Index, anhand derer sie identifiziert werden können. Die *O_Node*s, die sämtliche Variablen repräsentieren, werden vom Interpreter so unterschieden und entsprechend abgespeichert.

Die meisten Programmiersprachen teilen sich die gleichen möglichen Konstrukte, die der Programmierer verwenden kann. So gibt es in nahezu jeder Sprache Schleifen, die zwar die gleiche Funktionalität besitzen, aber anders dargestellt werden. So gibt es `for`-Schleifen die die Initialisierung von Variablen erlauben und jene bei denen das nicht erlaubt ist. In Ada wird der Zähler der Schleife bei jeder Iteration um den gleichen Wert erhöht, während in C, C++ und Java dies frei definiert werden kann.

Um die Analysen dieser Konstrukte zu vereinfachen, werden sie in der IML auf generische Knoten abgebildet. Sollte ein Knoten in der Darstellung nicht mächtig genug sein, um die Variante einer speziellen Sprache darzustellen, so existiert in der IML dafür ein von der

Basisklasse des Konstruktes abgeleiteter Knoten, der die entsprechenden Eigenschaften repräsentieren kann. Knoten die nur für bestimmte Sprachen verwendet werden, haben einen entsprechenden Prefix: `C_`, `Cpp_`, `Ada_` oder `Java_`.

Wie die einzelnen Knoten aufgebaut sind und wie der Interpreter sie verarbeitet, ist detailliert in Anhang A aufgeführt.

Das folgende Beispiel soll den Aufbau der IML-Graphen verdeutlichen und die Vorgehensweise des Interpreters erläutern. In Abbildung 2.1 ist ein vereinfachter Graph, eine Initialisierung einer Variable mit einem Produkt zweier Werte, abgebildet. Dies ist nur ein Teil eines möglichen IML-Graphen, sämtliche Deklarationen und die Repräsentation der eigentlichen Funktionen wurden entfernt.

Die Quelle (*Source*) der Zuweisung ist hier ein *Multiply*-Knoten, der die arithmetische Multiplikation repräsentiert. Das Ziel (*Target*) stellt ein *Entity_L_Value*-Knoten dar, der auf eine Variable verweist. In diesem Fall ist es die Variable `x` vom Typ *TC_Int*. Die Zahlen in den eckigen Klammern hinter den Namen der Knoten sind die Indizes, anhand denen jeder Knoten eindeutig identifiziert werden kann.

Die Aufgabe des Interpreters ist es, den Knoten auf den die *Source*-Kante verweist, zu einem Wert aufzulösen und so ein Symbol zu erhalten. Dieses beinhaltet einen Verweis auf den Speicherbereich, der das Ergebnis der Multiplikation beinhaltet. Nach Ermittlung des *Target*-Symbols kann der Interpreter den Speicher aus dem Bereich des *Source*-Symbols in den des *Target* kopieren. Sollte der *Initialize*-Knoten selbst als Wert verwendet werden, so hätte dieser den Typ *TC_Int*, wie der gestrichelten Kante des Knotens entnommen werden kann.

Die eigentliche Berechnung, die der Interpreter durchführt, findet beim *Multiply*-Knoten statt, nachdem die beiden Operanden zu einem Wert aufgelöst wurden. Das Ergebnis der Multiplikation, das ebenfalls vom Typ *TC_Int* ist, wie durch die *Its_Type*-Kante bestimmt werden kann, wird dann an den Initialisierungsknoten zurückgegeben. Dort wird das Ziel der Initialisierung zu einem Symbol, das alle relevanten Informationen über die betroffene Variable beinhaltet, aufgelöst und der Wert in dessen Speicherbereich geschrieben.

Nach dieser Zuweisung springt der Interpreter zu dem über der Initialisierung stehenden Knoten, meist vom Typ *Statement_Sequence*, zurück und führt die nächsten Anweisungen aus.

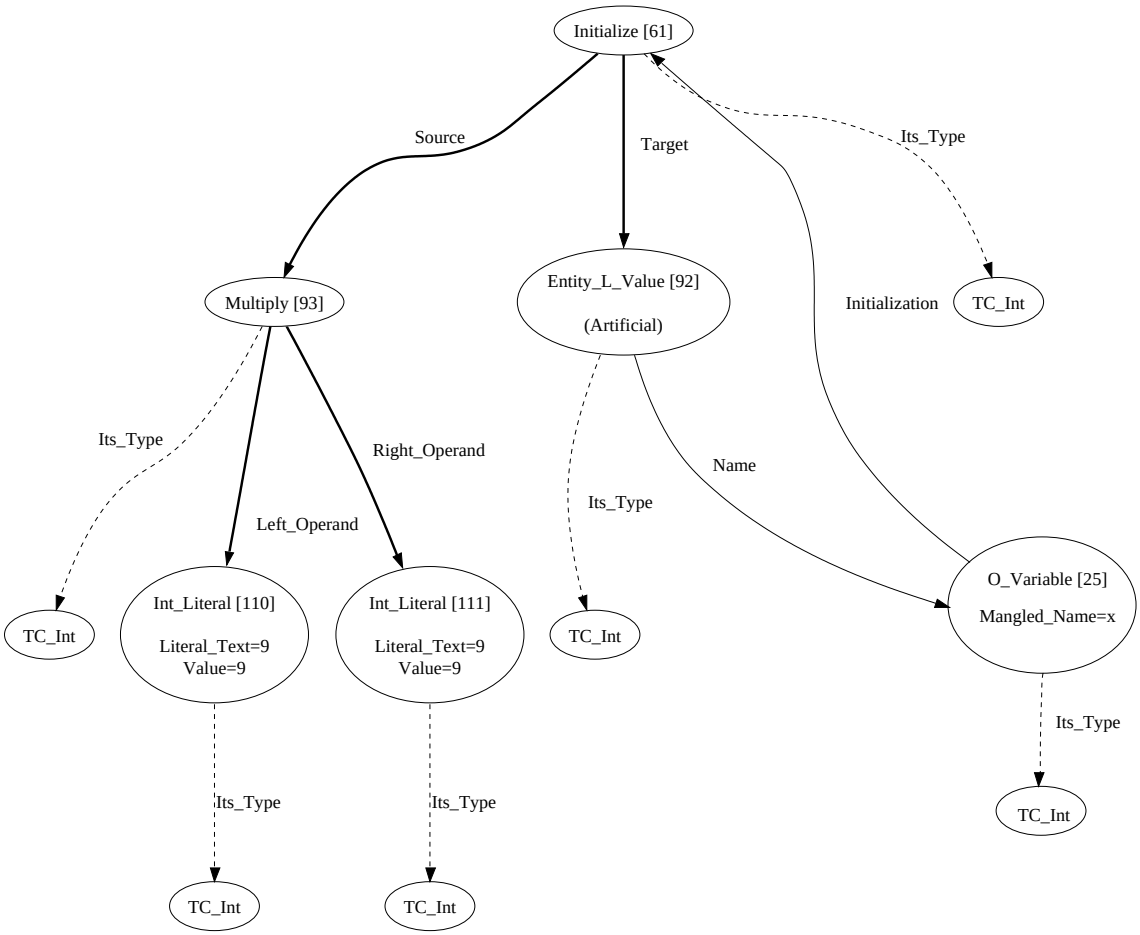

Abbildung 2.1: Vereinfachter Teil eines IML-Graphen. Dargestellt ist hier eine Zuweisung, bei der es sich bei dem Quelloperanden um eine Multiplikation handelt: `int x = 9 * 9`

KAPITEL 3

Interpreter

In diesem Kapitel wird der Aufbau des Interpreters und seiner einzelnen Komponenten näher erläutert. Sowohl verworfene als auch verwendete Ansätze werden hier aufgeführt.

Die IML verfügte zu dem Zeitpunkt der Entwicklung des Interpreters über keinerlei Laufzeitverhalten und diente bisher als Grundlage für statische Analysen des repräsentierten Quellcodes.

Es handelt sich bei dem Interpreter um ein eigenständiges Programm, das gewöhnliche IML-Dateien einliest und diese, so die abgebildete Sprache, beziehungsweise die vorkommenden Knoten und deren Kombinationen und Konfigurationen, unterstützt werden, ausführt.

Entwickelt wurde der Interpreter komplett in Ada95. Die Entscheidung fiel auf diese Sprache, da der Großteil des Bauhaus-Projektes in Ada95 geschrieben wurde und der Zugriff auf dessen Komponenten so erleichtert wurde. Nur einzelne Hilfsprogramme des Interpreters zur Generierung von Quellcode wurden in C geschrieben (siehe Kapitel 3.5.1.2).

Das Hauptaugenmerk lag bei der Implementierung bei der möglichst exakten und kompletten Unterstützung von aus C-Quellcode generierter IML. Die Anbindung weiterer Sprachen sollte erst anschließend erfolgen.

Wie einzelne Knoten vom Interpreter behandelt werden, ist Anhang A zu entnehmen. Dort sind alle relevanten Knoten näher erläutert und anhand von Beispielen wird genauer auf sie eingegangen.

Das nächste Kapitel beschreibt, welche Speichermodelle für den Interpreter in Frage kamen und welches letztendlich verwendet wurde. Anschließend wird auf die unterstützten Sprachen eingegangen und welche ihrer Funktionalitäten integriert wurden. Wie der Interpreter die einzelnen Datentypen der Sprachen repräsentiert ist dem Kapitel 3.3 zu entnehmen. In Kapitel 3.4 wird erklärt, wie interne Funktionen repräsentiert und externe, jene die nicht in der IML abgebildet sind, unterstützt werden. Wie der Interpreter installiert und verwendet wird, wird in Kapitel 3.5 detailliert beschrieben.

3.1 Speichermodell

Dieses Kapitel beschreibt die Anforderungen an das Speichermodell des Interpreters und welche Modelle verwendet wurden und welches letztendlich verwendet wird.

Ein wichtiger Punkt bei der Entwicklung eines Interpreters ist die Entscheidung für ein Speichermodell. Nicht nur müssen Variablen so abgelegt werden, daß schnell auf diese zugegriffen werden kann, sondern sollten auch Transformationen und Arithmetiken möglich sein, die genau denen der abgebildeten Sprache gleichen. Dies gilt bei der Wahl für ein Speichermodell zu berücksichtigen.

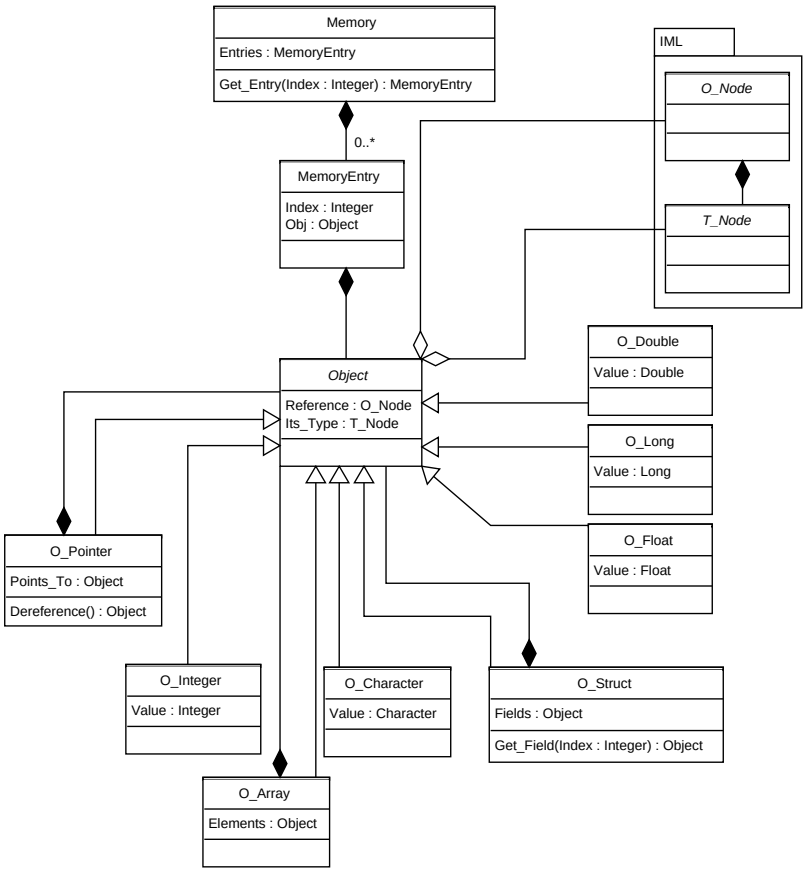

Abbildung 3.1: UML-Klassendiagramm des abstrakten Speichermodells

Da die abzubildenden Sprachen sich stark unterscheiden, liegt die Verwendung eines möglichst abstrakten Speichermodells nahe. Da aber auf der anderen Seite auch grundlegende Operationen auf dem Speicher möglich sein müssen, wie sie bei C üblich sind, empfiehlt sich ein Speichermodell das dem der Ursprungssprache gleicht.

In den folgenden Abschnitten wird näher auf die verwendeten und verworfenen Methoden eingegangen und die jeweiligen Entscheidungen erläutert.

3.1.1 Speicherung der Variablen

Ein relevanter Punkt für das Speichermanagement ist die Aufbewahrung der einzelnen Datenwerte. Entweder kann dies dem Betriebssystem überlassen werden oder der Interpreter beansprucht einen bestimmten zusammenhängenden Speicherbereich und verwaltet diesen selbst.

In Frage kamen bei der Entwicklung des Interpreters zwei unterschiedliche Modelle, die die verwendeten Variablen unterschiedlich verwaltet haben. Beide Modelle, das abstrakte und das binäre, werden in den folgenden Kapiteln beschrieben und deren Vor- und Nachteile aufgeführt.

3.1.1.1 Abstraktes Modell

Die erste implementierte Version des Speichers des Interpreters bestand nicht aus binären Speicherblöcken, sondern aus einem Hash Mapping (siehe [12, Seiten 201–211]) von dem jeweiligen Index innerhalb der IML auf das entsprechende Objekt. Die Werte der einzelnen Variablen waren in den Symbolen selbst gespeichert, die direkt in der Symboltabelle lagen und zusätzliche Informationen wie die Größe, den Typ, den Namen und den Index beinhalteten. Eine Trennung des eigentlichen Wertes und dieser Informationen existierte nicht.

Diese Variante wies einige Schwachstellen auf, die im Folgenden erläutert werden. Daher wurde sie verworfen und das binäre Modell, das im nächsten Kapitel beschrieben wird, verwendet.

Die Verwaltung des Speichers des Interpreters wurde also dem System überlassen, es bestand kein Zusammenhang zwischen dem Aufbau des Speichers des interpretierten Programms und der Abbildung innerhalb des Interpreters.

Solch eine Symboltabelle existierte für jede einzelne Funktion und einmal global, für sowohl globale als auch statische Variablen. Die einzelnen, getrennten Mappings waren nötig, um Probleme bei der Rekursion von Funktionen zu verhindern, bei der einzelne Objekte mit dem gleichen Index zur selben Zeit im Speicher verweilen müssen (siehe Kapitel 3.1.2.1).

Für jeden Variablentyp, sowohl primitive als auch komplexere wie *Structs* und *Arrays*, gab es eine eigene Klasse, die die nötigen Funktionen und Informationen beinhaltete. Zu diesen Informationen gehörte die Anzahl der Elemente, die Aufzählung der Membervariablen und der Typ der enthaltenden Objekte.

Dieses Modell war in seiner Funktionalität begrenzt. *Pointer*-Algorithmen funktionierten nur innerhalb des jeweiligen Objektes und unsichere Operationen, also jene die den Speicherbereich des Objektes normalerweise verlassen würden, waren nicht möglich. Auch Unions ließen sich nicht realisieren (siehe Kapitel 3.3.3), ohne Teile des eigentlichen Modells zu verändern.

Selbst wenn die Objekte alle hintereinander im Speicher liegen würden, so wären die Zeigeroperationen nicht abbildbar, da zwischen den eigentlichen Datenwerten benachbarter Objekte immernoch die zusätzlichen Informationen des Objekts, die für die Nutzung mit dem Interpreter nötig sind, liegen würden. So beinhalten Symbole unter anderem Verweise auf die abgebildete Variable innerhalb des IML-Graphen (ein Knoten vom Typ *O_Node*) und einen direkten Verweis auf den Typ der Variable (ein Knoten vom Typ *T_Node*).

3.1.1.2 Binäres Modell

In Sprachen wie C, in denen mit Hilfe von Zeigern direkt auf den Speicher zugegriffen werden kann und keine zwingende Typensicherheit existiert, versagt das abstrakte Speichermodell. Denn in dem Modell ist es nicht möglich, mit Zeigern die Grenzen des eigentlichen Elements (zum Beispiel eines *Arrays*) auf das verwiesen wird, zu über- oder unterschreiten. Ebenso ist es nicht möglich mittels eines Zeigers auf einen Speicherbereich zuzugreifen, der von einem anderen Typ ist.

Um solche komplexeren und unsicheren Operationen korrekt abbilden zu können, ist ein binäres Speichermodell notwendig. Dieses besteht im wesentlichen aus zwei Teilen, dem binären Speicherblock und den dazugehörigen Symboltabellen. Alle Informationen die zur Nutzung des Objektes benötigt werden, sind in dem dazugehörigen Symbol gespeichert, so daß im binären Speicherblock nur die eigentlichen Datenwerte stehen, wie es bei dem interpretierten Programm auch der Fall ist. So lassen sich alle Operationen auf dem Speicher ohne Einschränkungen ausführen.

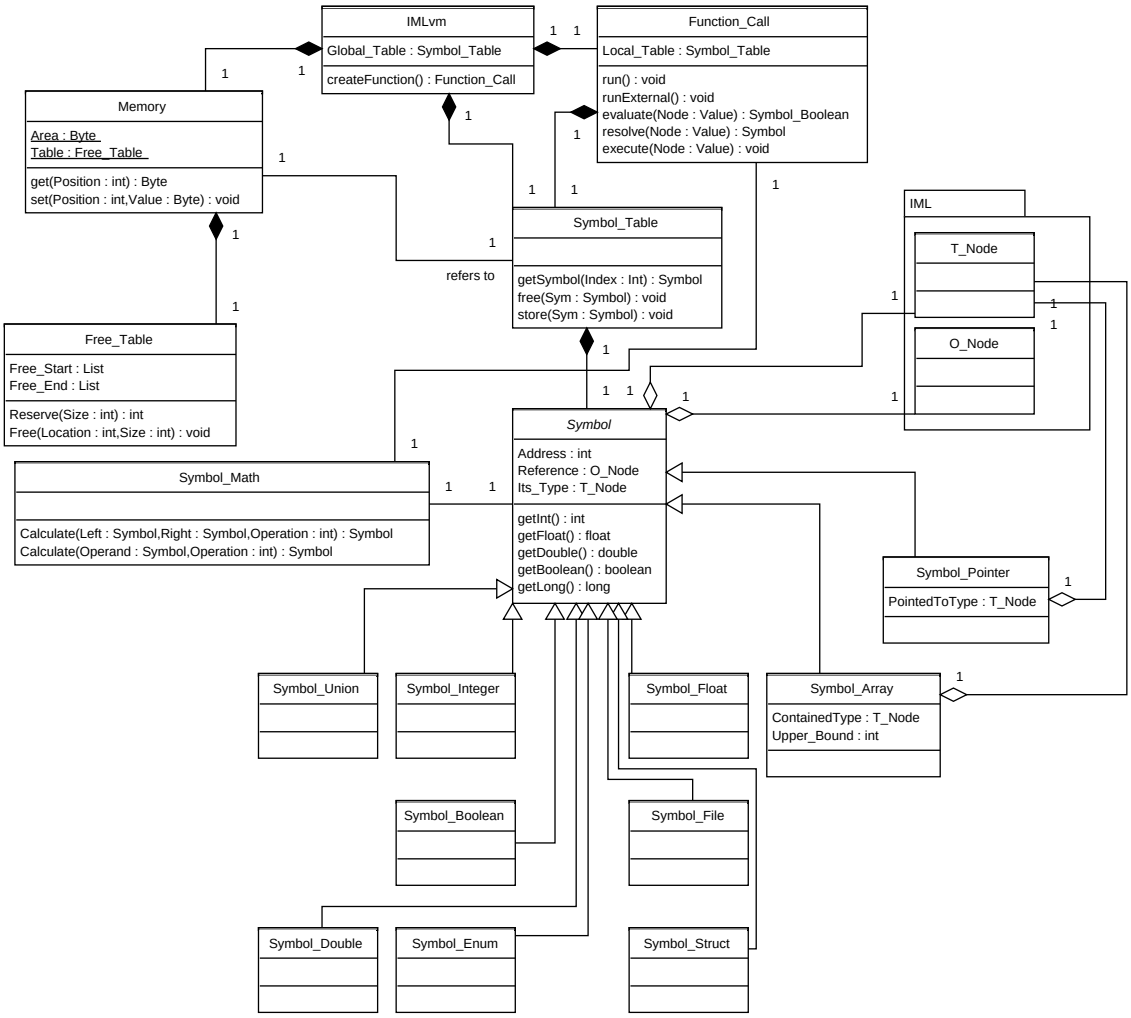

Abbildung 3.2: UML-Klassendiagramm des binären Speichermodells

Da die Größe des zur Verfügung stehenden Speichers begrenzt und bekannt ist, wird sofort erkannt falls eine Operation auf einen Speicherbereich angewendet wird, der außerhalb des definierten Bereichs liegt.

Die statische Klasse *Memory* beinhaltet den gesamten binären Speicher des Interpreters, in dem alle Variablen abgelegt werden. Desweiteren verfügt sie über die so genannte *Free-Table*, in der die noch freien Speicherblöcke eingetragen werden. Die Klasse hat keinerlei Kenntnis von den existierenden Symboltabellen oder davon, was genau in dem Speicher abgelegt ist. Diese Information wird nur in den Symboltabellen gespeichert, von denen jeder Funktion eine zugeordnet ist.

Wird ein neues Symbol erstellt, so wird dessen Index anhand des IML-Graphen ermittelt; temporäre Variablen, wie Zwischenergebnisse von Berechnungen, erhalten negative fortlaufende Indizes. Dabei kommt es zu keinem Konflikt mit den Indizes aus dem IML-Graphen, da diese stets positiv sind.

Ist der Index ermittelt, wird das Symbol in die entsprechende Stelle der Tabelle eingetragen. Dabei handelt es sich um ein *Hash Mapping*, bei dem die eigentliche *Hash*-Funktion die Identität des Index ist. Da kein Symbol mit dem gleichen Index zur selben Zeit in einer Tabelle existieren kann, kann es hierbei zu keinerlei Konflikten kommen (siehe Kapitel 3.1.2.1).

Nach dem Eintrag in die Symboltabelle wird freier Speicher mit der jeweiligen Größe des Datentyps der neuen Variable angefordert. Die Liste der noch freien Blöcke wird dazu nach einem Block durchsucht, der groß genug ist, um ein Objekt der geforderten Größe zu beinhalten. Wurde solch ein Block gefunden, wird die Anfangsadresse des Blocks in das Symbol geschrieben und die Einträge des noch verfügbaren Speichers entsprechend angepasst.

Jedes Byte des für den Interpreter reservierten Speichers ist anfangs mit Null initialisiert. Wird ein bereits vorher benutzter Speicherbereich erneut vergeben, so wird er zuvor nicht erneut mit Null initialisiert, sondern behält seine vorherigen Werte bei.

3.1.2 Symboltabelle

Die Aufgabe der Symboltabelle ist die Zuordnung von Variablen auf den dazugehörigen Speicherbereich und somit ihren Wert. Wird eine neue Variable angelegt, so wird deren Wert in den Speicher geschrieben und ein Eintrag an die Symboltabelle angefügt, der einen Verweis auf sowohl den Variablennamen, die Adresse im Speicher und andere wichtige Informationen beinhaltet.

Es ist nicht nur wichtig, welche Variable wo im Speicher abgelegt ist, sondern auch wieviel Platz sie belegt und von welchem Typ sie ist. Die Größe ist besonders beim Kopieren oder Freigeben des Speicherbereichs wichtig. Für die Nutzung selbst ist sie beim binären Speichermodell unerheblich, da Daten hier immer mit der geforderten Größe ausgelesen werden. Sollte dabei der eigentliche Speicherbereich der Variable überschritten werden, so werden Teile eines für die Variable fremden Speicherbereichs ausgelesen. Dabei handelt es sich zwar nicht um eine sichere Operation, dennoch ist sie bei einigen Sprachen erlaubt und auch durchaus üblich.

Da die Indizes der Symbole eindeutig sind und ein Knoten maximal einmal in die Tabelle eingetragen wird, bietet sich ein *Hash Mapping* an, um einen möglichst schnellen Zugriff auf die Symbole zu ermöglichen.

3.1.2.1 Einzelne Tabelle

Wenn nur eine einzelne Symboltabelle verwendet wird, entspricht dies einer simplen Zuordnung vom Index des entsprechenden IML-Knoten auf das dazugehörige interne Objekt.

Ob es sich bei diesem internen Objekt um die eigentliche Variable oder aber nur um einen Verweis auf den binären Speicherbereich handelt, ist dabei unerheblich, und daher unabhängig von dem verwendeten Symboltabellenmodell.

Die Variablen innerhalb der IML verfügen über einen Index anhand dessen sich jeder Knoten innerhalb des Graphen eindeutig identifizieren läßt. Diesen Index als Identifikation für die Symbole zu verwenden, behebt das Problem gleichbenannter Variablen in unterschiedlichen Funktionen, das auftritt, wenn man sie nur anhand ihrer gegebenen Namen unterscheidet.

Dies würde aber immernoch bedeuten, daß von jeder Variable jederzeit nur eine Instanz im Speicher liegen könnte. Das mag zwar in den meisten Fällen einen Sinn ergeben, sollte eine Funktion aber rekursiv oder zumindest mehrmals, bevor die vorherige terminiert, aufgerufen werden, so kommt es zu einem Konflikt zwischen dem alten und dem neuen Wert. Somit würden im Endeffekt alle Variablen innerhalb einer Funktion behandelt, als wären sie als statisch deklariert worden, da sie sich einen Speicherbereich teilen würden.

Dies läßt sich gut am Beispiel der Fakultät näher erläutern. Gegeben ist die Funktion `fak` (siehe Listing 3.1), die rekursiv die Fakultät des übergebenen Parameters berechnet. Die Abbruchbedingung ist erfüllt, wenn dieser Parameter kleiner oder gleich Eins ist und in dem Fall wird ebenso Eins zurückgegeben; ansonsten das Produkt des Parameters und des rekursiven Aufrufs, mit dem dekrementierten Parameter. Nun würde diese Funktion so lange rekursiv aufgerufen werden, bis die Abbruchbedingung erfüllt wäre.

Da sich nach diesem Modell alle Funktionen der gleichen Symboltabelle bedienen, wäre auch für alle Funktionen die Variable `x` im gleichen Speicherbereich. Somit würde sich der Aufruf `fak(5)` auflösen lassen in `1 * 1 * 1 * 1 * 1`, statt in `5 * 4 * 3 * 2 * 1`. Nun könnte man argumentieren, das ließe sich umgehen, wenn nach dem Auslesen des ersten Operanden der Multiplikation dieser zwischengespeichert werden würde. Dies würde aber nicht das gleiche Problem für den Fall des folgenden rekursiven Aufrufs lösen: `return fak(x-1) * x;`. Denn steht die nach der angeblichen Lösung zu zwischenspeichernde Variable an zweiter Stelle, wird diese schon im ersten rekursiven Aufruf verändert. Die Reihenfolge in der die Operanden aufgelöst werden, ist dabei vom jeweiligen Standard der repräsentierten Sprache festgelegt oder sogar den Compilern überlassen (siehe Kapitel 3.4.4).

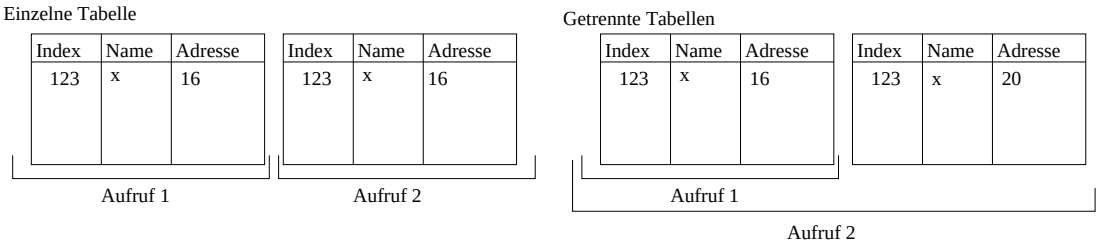

Abbildung 3.3: Bei einer einzelnen Symboltabelle wird bei jedem rekursiven Aufruf der Funktion das Symbol in der Tabelle überschrieben, während bei einer Trennung jeder Aufruf über seine eigene Tabelle verfügt und sich die Symbole daher nicht überschreiben und nicht den Speicherbereich teilen.

Listing 3.1: rekursive Fakultät

```
1 int fak(int x){
2   if (x <= 1) return 1;
3   return x * fak(x-1);
4 }
```

Dieses Problem legt eine Trennung der Tabellen nahe, bei der jede Funktion über eine eigene Symboltabelle verfügt und sich so deren Sichtbarkeit nicht überdeckt.

3.1.2.2 Getrennte Tabellen

Eine eigene Tabelle für jede aufgerufene Funktion beseitigt das Problem, wirft aber ein weiteres auf. Denn wenn eine Funktion nur eine eigene lokale Symboltabelle besitzt, hat sie keinerlei Kenntnis von globalen Variablen. Diese werden außerhalb von Funktionen definiert und können somit nicht in den Symboltabellen der einzelnen Funktionen gefunden werden.

Daher empfiehlt es sich, eine weitere globale Symboltabelle für eben jene Variablen einzuführen, die global oder statisch sind. Da die Symboltabellen der einzelnen Funktionen nicht statisch sind und bei jedem Aufruf der Methode neu erstellt werden, können in dieser nicht die statischen Variablen der jeweiligen Funktionen abgelegt werden. Hier läßt sich die Tatsache nutzen, daß in der IML alle Variablen anhand einer eindeutigen Nummer, dem IML-Index, identifiziert werden können und dies nicht über den Namen der Variable geschieht. Daher können in der globalen Tabelle auch mehrere statische Variablen unterschiedlicher Funktionen abgelegt werden, die den gleichen Namen tragen.

In der globalen Tabelle stehen demnach sowohl alle globalen als auch statischen Variablen der Funktionen, während die lokalen Tabellen sämtliche Variablen beinhalten die nicht statisch sind und innerhalb der jeweiligen Funktion deklariert oder als Parameter übergeben wurden.

Soll nun eine Variable ausgelesen werden, so wird zuerst in der lokalen Tabelle geschaut. Wird sie dort nicht gefunden, wird die globale Tabelle kontrolliert und gegebenfalls das gefundene Symbol zurückgegeben. Wird ein Symbol nicht gefunden, gibt die Symboltabelle `null` zurück.

Verschachtelte Funktionen, wie sie in Ada existieren, werden durch diesen Aufbau nicht abgedeckt. Es ist in Ada möglich Funktionen innerhalb von Funktionen zu definieren, um das Duplizieren von Quellcode zu umgehen und einzelne Funktionalität aus dem eigentlichen Funktionsrumpf auszulagern. Jede verschachtelte Funktion hat Kenntnis von den Variablen der übergeordneten Methode und kann uneingeschränkt auf diese zugreifen.

Um dieses Problem zu umgehen, sind die einzelnen Tabellen miteinander verknüpft. Bei der Erstellung einer neuen Tabelle, beim Aufbau eines neuen Funktionsaufrufs, wird in dieser sowohl die globale, als auch die übergeordnete Tabelle als Verweis verknüpft, so dies denn nötig ist. So ist es möglich, sollte ein gesuchtes Symbol nicht in der aktuellen Tabelle gefunden werden, in den überliegenden Tabellen danach zu suchen.

Listing 3.2: Definition der getrennten verlinkten Symboltabelle

```
1 type Symbol_Table_Class is record
2    Symbols : Symbol_Mappings.Mapping;
3    Last_Temporary : Integer := -1;
4    Is_External    : Boolean := False;
5
6    Parent : access Symbol_Table_Class := null;
7    Global : access Symbol_Table_Class := null;
8 end record;
```

Die *Parent*-Verknüpfung, die auf die übergeordnete Tabelle verweist, existiert dabei nur bei verschachtelten Funktionen, da ansonsten jeder Funktionsaufruf auf die Variablen der aufru-

fenden Funktion zugreifen könnte. Solch ein unzulässiger Zugriff auf nicht sichtbare Symbole beziehungsweise Variablen, wird schon bei der Erstellung der IML abgefangen.

Die Variable *Is_External* bestimmt, ob es sich bei dieser Tabelle um eine für externe Funktionsaufrufe handelt. In solch einem Fall werden hier nur die Parameter und der Rückgabewert der jeweiligen Funktion abgelegt. Nur bei emulierten externen Funktionen wird sie eventuell für weitere Symbole benötigt. Externe Symboltabellen sind nie mit anderen Tabellen verlinkt, da diese Funktionen keine Information über die aufrufende Umgebung haben und daher auf diese nicht zugreifen können.

3.2 Unterstützte Sprachen

Obwohl es sich bei der IML um eine Repräsentation handelt, die unabhängig von der ursprünglichen Programmiersprache ist, so müssen doch alle Eigenheiten der unterstützten Sprachen in ihr abgebildet werden können.

So existieren in der IML Knotentypen, die nur für einzelne Sprachen verwendet werden oder jeweils unterschiedliche Knoten für eine Anweisung, die zwar in mehreren Sprachen verwendet wird, aber unterschiedlich behandelt werden muss.

Daher ist auch beim Interpreter auf die Besonderheiten der einzelnen von ihm zu unterstützenden Sprachen zu achten. Dazu gehören neue Datentypen aber auch die komplette Unterstützung von Objektorientierung oder anderen, exotischeren Möglichkeiten, die die Sprachen bieten.

Welche der durch die IML repräsentierten Sprachen vom Interpreter unterstützt werden und welche nicht, wird in den nachfolgenden Kapiteln erläutert.

3.2.1 Sprache: C

Als Besonderheit der Sprache C gegenüber anderen unterstützten Sprachen (C++ da es auf C aufbaut ausgenommen) ist die freie Verwendung von *Pointern* und die damit verbundene Arithmetik, die direktes Arbeiten auf dem Speicherbereich des Programms erlaubt. Dies erzwingt ein binäres Speichermodell, wie es in den vorherigen Kapiteln erläutert wurde.

Seit der Entwicklung von C sind drei internationale Standards erschienen, die die grundsätzliche Funktionalität darstellen beziehungsweise erweitern. Dabei handelt es sich sowohl um Änderungen an den bestehenden Header-Dateien von C als auch um komplett neue, die zum Beispiel die Verwendung von komplexen Zahlen ermöglichen.

Die einzelnen Standards und in wie weit sie vom Interpreter unterstützt werden, wird im Folgenden erläutert. Der Status der in Anhang B befindlichen Tabellen beschreibt, in wie weit die jeweilige Funktion vom Interpreter unterstützt wird. “Interface” steht dabei dafür, daß die Funktion über die in Ada integrierten Schnittstellen angesprochen wird. Funktionen die mit “Emulation” markiert sind, werden vom Interpreter emuliert, da ihre Ausführung über die Schnittstellen nicht möglich ist. Falls eine Einbindung nicht möglich war, wird die Funktion nicht in den Tabellen aufgeführt, sondern entsprechend in den folgenden Kapiteln kommentiert.

3.2.1.1 Standard: ANSI-C89/ISO-C90

Der ANSI-C89[1]- und der ISO-C90[2]-Standard definieren die Mindestanforderungen an bestimmte Header-Dateien, die die grundlegenden Funktionen von C zur Verfügung stellen sollen.

Bei einigen Funktionen erlaubt der Standard auch die ersatzweise Verwendung von Makros. In dem Fall wird die jeweilige Funktion komplett in der IML abgebildet, da diese vom Präprozessor aufgelöst und vollständig in den eingebundenen Headern definiert sind.

stdlib.h Dieser Header beinhaltet einen Großteil aller verfügbaren Standardfunktionen des C-Standards von 1989. Nur die Methoden zur Speicherverwaltung (`malloc`, `calloc`, `free`) mussten emuliert werden, da ansonsten neuer Speicher belegt werden würde, der außerhalb des Bereichs des emulierten binären Speichers liegen würde. In Tabelle B.1 sind alle unterstützten Funktionen aus diesem Header aufgeführt.

Die folgenden Funktionen können nicht interpretiert werden:

`bsearch` Diese Funktion führt normalerweise eine binäre Suche über einen bestimmten Speicherbereich durch und vergleicht die Elemente anhand einer per Funktionspointer angegebenen Vergleichsfunktion. Da hier auch Funktionen angegeben werden können, die in der IML enthalten sind und sonst nirgends im Speicher residieren, ist es nicht möglich einen *Pointer* auf diese Funktionen zu erzeugen und dann über die Schnittstelle an C weiterzugeben (siehe Kapitel 3.3.3).

`atexit` Mittels dieser Funktion läßt sich ein Funktionspointer auf eine Methode speichern, die aufgerufen wird, wenn das Programm endet. Da sowohl interne als auch externe Funktionen angegeben werden können, ist ein Aufruf über die Schnittstellen nicht möglich. Daher wird diese Funktion nicht vom Interpreter unterstützt.

`qsort` Diese Funktionen benötigt als Parameter genau wie die `bsearch`- und `atexit`-Funktion einen Funktionspointer. Daher ist auch hier eine Einbindung über die Schnittstellen nicht möglich.

time.h Diese Header-Datei stellt Funktionen zur Verfügung, mit denen Informationen über die aktuelle Zeit ausgelesen werden können. Welche Funktionen unterstützt werden, kann der Tabelle B.2 entnommen werden.

In diesem Header ist ein statisches Kontrukt vom Typ `tm` enthalten, das das Ergebnis von bestimmten Funktionsaufrufen beinhaltet. Die Funktionen, die einen Verweis auf dieses Konstrukt zurückgeben, lassen sich daher nicht über die Schnittstellen aufrufen. Da das Konstrukt außerhalb des Speicherbereichs des Interpreters liegt, sind Zugriffe auf dessen Felder nicht möglich. Folgende Funktionen lassen sich deswegen nicht nutzen: `gmtime` und `localtime`.

math.h Dieser Header stellt grundlegende mathematische Funktionen zur Verfügung, die in der Tabelle B.3 aufgeführt sind. Dazu gehören zum Beispiel geometrischen Funktionen wie `sin`, `cos` und `tan`, definiert für den Datentyp `double`. Sämtliche Funktionen dieses Headers werden vom Interpreter unterstützt.

stdio.h Die in dieser Headerdatei enthaltenen Funktionen dienen der Ein- und Ausgabe, wie sie im Standard definiert sind. Welche Funktionen wie unterstützt werden, ist der Tabelle B.4 zu entnehmen.

[1] American National Standards Institute X3.159-1989 Programming Language C
[2] Internationale Organisation für Normung 9899:1990

Folgende Funktionen werden nicht unterstützt:

`vprintf, vfprintf, vsprintf` Diese Methoden verhalten sich im Grunde wie ihre Äquivalente `fprintf`, `printf` und `sprintf`, erhalten anstatt optionaler Parameter eine Parameterliste vom Typ `va_list`, der in dem Header `stdarg.h` definiert ist. Diese Funktionen werden hauptsächlich von Methoden verwendet, die selbst optionale Parameter bekommen und mit diesen die jeweiligen Ausgabefunktionen aufrufen.

Da die Makros aus dem Header `stdarg.h`, die für die Nutzung benötigt werden, von den IML-Generatoren ignoriert werden, können diese Funktionen nicht sinnvoll genutzt werden. Zwar ist ein Aufruf der Funktionen möglich aber die Parameter können nicht entsprechend generiert werden, wie dem Abschnitt des Headers zu entnehmen ist.

Dies führt dazu, daß die Funktionen zwar aufgerufen werden, ihr Verhalten aber undefiniert ist, da die Parameter nicht initialisiert werden können.

locale.h Dieser Header definiert ein Locale-Struct (`lconv`), das die unterschiedlichen Konventionen einer Locale[3] als Strings beinhaltet. Zu diesen Variablen gehören unter anderem das Währungszeichen und die Darstellungsart von Zahlen.

Da es sich bei diesem Struct um eine statische Instanz handelt, die außerhalb des vom Interpreter definierten Speicherbereichs liegt, ist eine Verwendung nicht direkt möglich. Zwar kann eine Kopie des Structs angelegt werden, auf dem anstelle des statischen Structs gearbeitet wird, sämtliche Änderungen die in externen Funktionen durchgeführt werden, wären dann aber ohne Auswirkungen.

Daher wird die Funktion `localeconv` nicht unterstützt. Die unterstützten Funktionen sind in Tabelle B.6 aufgeführt.

ctype.h Die hier aufgelisteten Funktionen dienen der Identifikation einzelner Zeichen bezüglich des verwendeten Zeichensatzes. Alle gegebenen Funktionen werden über die integrierte Schnittstelle angesprochen, wie der Tabelle B.6 zu entnehmen ist.

Problematisch gestaltet sich dies jedoch, wenn der `gcc` als Compiler verwendet wird. Denn dieser definiert die Funktionen als Makros, die auf ein lokalisiertes Zeichenarray verweisen, auf das der Interpreter nicht zugreifen kann. Daher ist eine Verwendung der Funktionen in dem Fall nicht möglich. Wird der gleiche Quellcode mit dem `g++` kompiliert, der Funktionen statt Makros verwendet, können die Methoden über die Schnittstellen entsprechend angesprochen werden.

string.h Die Funktionen bezüglich C-Strings sind in diesem Header definiert. Sämtliche Funktionen die hier definiert sind (siehe Tabelle B.7), lassen sich über das Interface ansprechen und benötigen keine besondere Behandlung.

assert.h Dieser Header definiert das Makro `assert`, das die Ausführung des Programms beendet, wenn die angegebene Zusicherung nicht erfüllt ist. Wie der daraus in der IML generierte Knoten behandelt wird, kann Kapitel A.15.8 entnommen werden.

errno.h In diesem Header werden Konstanten definiert, anhand derer Fehlercodes identifiziert werden können. Daher ist keine besondere Behandlung durch den Interpreter nötig.

[3] Eine Klasse oder ein Konstrukt das sämtliche Informationen über eine Sprache beinhaltet, die nötig sind um Werte zu formatieren. Dazu gehören sowohl Währungs- als auch Trennzeichen.

stddef.h Keine neuen Funktionen sind in diesem Header enthalten und daher wird auch keine besondere Behandlung durch den Interpreter benötigt.

stdarg.h Dieser Header beinhaltet Funktionen und Datentypen für die Behandlung von variablen Parameteranzahlen. Bei der Generierung der IML werden die in diesem Header definierten Makros komplett ignoriert. An ihrer Stelle hängen im IML-Graphen *Null_Expressions*, weswegen die entsprechenden Variablen, die für die Nutzung von Parameterlisten nötig sind, nicht initialisiert werden können. Ohne eine Erwähnung dieser Makros innerhalb der IML, ist auch eine Emulation nicht möglich. Daher ist die Verwendung von Methoden, die auf diese Makros benötigen (wie zum Beispiel `vprintf`) nicht möglich.

signal.h Dieser Header beinhaltet sowohl Makros als auch Funktionen zur Auslösung und Behandlung von Signalen. Diese werden in der Form von dem Interpreter nicht unterstützt (siehe Kapitel 3.4.2.1.2).

setjmp.h Die hier enthaltenen Funktionen dienen zur Manipulation des *Stacks* und ermöglichen das Springen an andere Stellen innerhalb des Programms bezüglich der Ausführung. Da die damit verbundenen Funktionen extern sind, ist das Ansprechen über die Schnittstellen nicht möglich. Die nicht unterstützten Funktionen sind: `longjmp`, `siglongjmp`, `sigsetjmp` und `setjmp`.

limits.h Die Wertegrenzen einzelner Datentypen sind in diesem Header definiert. Da es sich dabei nur um Konstanten handelt, ist eine Behandlung durch den Interpreter unnötig.

3.2.1.2 Standard: C95

Die größte Neuerung dieses Standards ist, neben Verbesserungen des Vorgängers, die Einführung der Unterstützung von *wide char*s. Diese erlauben es, Zeichenketten aus Buchstaben zu bilden, die größer als ein Byte sind und so die Darstellung und Abbildung von zum Beispiel Unicode-Zeichen ermöglichen.

Die IML ist zu diesen Zeitpunkt nicht in der Lage mit diesen *wide char*s umzugehen (siehe Kapitel 3.3.4). Daher ist eine Unterstützung durch den Interpreter nicht gegeben.

stdlib.h Hierbei handelt es sich nicht um einen neuen Header, sondern um die Erweiterung des `stdlib`-Headers aus dem vorherigen Standard. Die neuen, folgenden Funktionen dienen zur Unterstützung von *wide char*s und werden vom Interpreter nicht unterstützt.

`mbstowcs, mbtowc, wcstombs, wctomb, mblen` Diese Funktionen dienen der Verarbeitung von `wide char`s und `multi byte char`s, wie sie von der IML momentan nicht unterstützt werden. Daher erfolgte keine Anbindung dieser Funktionen.

iso646.h Dieser Header definiert einige Makros zur Erhöhung der Lesbarkeit des Quellcodes. So ist es zum Beispiel möglich mit diesem Header `and` anstelle von `&&`, einer bedingten Verundung, zu schreiben. Andere Unterschiede gibt es nicht, eine besondere Behandlung ist also nicht nötig, da dies komplett vom Präprozessor behandelt wird.

wchar.h Der Typ `wchar_t` der durch diesen Header eingeführt wird, wird von der IML nicht unterstützt. Näheres dazu kann dem Kapitel 3.3.4 entnommen werden. Aufgrund dieser mangelnden Unterstützung wurde auf eine Anbindung dieser Funktionen verzichtet.

wctype.h Für die Methoden dieser Headerdatei, die im Grunde die gleiche Funktionalität für `wchar_t` bietet, wie `ctype.h` für `char`, gilt das gleiche wie im vorherigen Abschnitt beschrieben.

3.2.1.3 Standard: C99

Bei diesem ISO-Standard[4] wurden aus C++ bekannte Erweiterungen integriert. Hierzu gehören zum Beispiel die Einführung der Kommentarzeichen `//` und von *inline*-Funktionen. Ebenso wurde die Definition von impliziten `int`s und Funktionen verboten. Die für den Interpreter relevanten Neuerungen sind im Folgenden erläutert.

complex.h Dieser Header definiert ein *Struct*, der komplexe Zahlen repräsentieren soll. Um mit diesem Konstrukt Berechnungen durchzuführen, muss der `tgmath.h`-Header inkludiert werden, der die dazugehörigen Funktionen beinhaltet.

Die Verwendung wird von der verwendeten Bauhaus-Installation[5] nicht unterstützt. Daher ist eine Interpretation nicht möglich.

fenv.h Dieser Header ermöglicht die Manipulation der Floating-Point-Umgebung des laufenden Programms. Mittels des `fenv_t`-Typen läßt sich zum Beispiel die Art beeinflussen, in der `float`s gerundet werden. Diese Einstellung hat innerhalb des Interpreters keinen Einfluß auf Rundungen die in internen, interpretierten Funktionen geschehen. Nur Operationen innerhalb von externen C-Standardfunktionen werden von dieser Einstellung beeinflußt.

Desweiteren ist es möglich, aufgetretene Floating-Point-Exceptions anhand des in diesem Header definierten Datentyps `fexcept_t` auszulesen, zu setzen und zu werfen. Welche Funktionen unterstützt werden, ist der Tabelle B.8 zu entnehmen. Auch in diesem Fall werden nur die in externen Funktionen aufgetretenen Exceptions berücksichtigt.

inttypes.h In diesem Header wird sowohl ein Struct, Macros als auch einige Funktionen definiert, die zur Behandlung, Ein- und Ausgabe von Integern bestimmter Größe gedacht sind. Die Macros lassen sich zu String-Literalen auflösen, die zur Nutzung innerhalb von `printf`- und `scanf`-Aufrufen zur Formatierung von Integern verwendet werden können. Da diese Makros vom Präprozessor aufgelöst werden, ist eine Behandlung durch den Interpreter nicht notwendig.

Die beinhalteten Funktionen definieren die Division ähnlich wie die Funktion `div` (siehe Tabelle B.1) und `ldiv` für den größten Integertypen, der in diesem Header definiert ist. Der Aufbau des Structs `imaxdiv_t`, das an sich dem `div_t` aus dem Header `math.h` ähnelt, ist im Standard selbst nicht definiert. Die Reihenfolge, in der der *Quotient* und der *Remainder* in diesem abgelegt sind, ist vom Entwickler der jeweiligen Compiler frei wählbar. Daher ist eine Interpretation der `imaxdiv` nicht möglich. Die Methoden `wcstoimax` und `wcstoumax` basieren auf `wide char`s, weswegen ihre Interpretation ebenfalls nicht möglich ist.

Die restlichen Funktionen sind abhängig von den Typen `intmax_t` und `uintmax_t`, die den größtmöglichen Integer beziehungsweise größtmöglichen positiven Integer darstellen. Da deren Definition von der jeweiligen Plattform und den verwendeten Compiler abhängt, ist auch hier eine Interpretation nicht möglich. Zu diesen Methoden gehören `imaxabs`, `strtoimax` und `strtoumax`.

[4] ISO/IEC 9899:1999
[5] Revision 22050, 26.02.07, 09:57

stdbool.h Dieser Header definiert weitere Makros für die Verwendung von `_Bool`, dem booleschen Datentypen in C, und bedarf daher keiner weiteren Beachtung bei der Implementierung des Interpreters, da diese vom Präprozessor aufgelöst werden.

stdint.h Dieser Header definiert Integer-Typen mit fixer Größe. Denn die Größe des generischen Typs `int` ist vom Compiler und der Plattform abhängig. Die in diesem Header definierten Typen sind unabhängig von der verwendeten Umgebung und decken in der Regel Größen von 8 bis 32 Bit ab.

Da keine neuen Funktionen in diesem Header zum C-Standard hinzugekommen sind, muss der Interpreter keine weiteren Methoden über die Schnittstellen ansprechen.

Die neu definierten Typen werden direkt von den Bauhaus-Tools erkannt und in die IML eingebunden, so daß auch hier keine besondere Behandlung nötig ist.

tgmath.h In diesem Header sind die mathematischen Funktionen für Berechnungen mit komplexen Zahlen definiert. Da komplexe Zahlen vom Interpreter nicht unterstützt werden, sind diese Funktionen über die Schnittstellen nicht ansprechbar.

3.2.2 Sprache: Java

Es existieren zur Zeit keine Interfaces von Ada zu *Java*, daher ist es nicht möglich die Standardbibliotheken von *Java* zu interpretieren. Aufgrund der Komplexität der vorhandenen Bibliotheken ist es nicht möglich diese zu emulieren.

Das Interpretieren eines Java-Programms das völlig ohne Fremd- oder die Standardbibliotheken auskommt, wäre theoretisch möglich, da alles relevante in der IML abgebildet wird. Solch ein Programm hätte aber keinerlei Nutzen, da selbst auf jegliche Ein- und Ausgabe verzichtet werden müsste.

Daher wurde auf die Umsetzung von IML-Knoten, die Java-Quellcode repräsentieren, verzichtet.

3.2.2.1 Garbage Collector

Eine Emulation von aus Java-Quellcode generierter IML ist zwar aus den im vorherigen Kapitel genannten Gründen nicht möglich, wie der *Garbage Collector* von Java aber dennoch emuliert wird, um in späteren Versionen verwendet werden zu können, ist in diesem Kapitel beschrieben.

In Java werden Objekte per Referenz behandelt, läuft also die Sichtbarkeit einer Variable aus, so wird nur die Referenz gelöscht, das Objekt selbst bleibt im Speicher erhalten. Für die Entfernung dieser Objekte ist letztendlich der *Garbage Collector* verantwortlich, der prüft, ob noch Verweise auf ein im Speicher verweilendes Objekt existieren. Erst wenn dies nicht mehr der Fall ist, wird das Objekt gelöscht und der Speicher freigegeben.

Nicht betroffen von dem *Garbage Collector* sind primitive Datentypen, da diese gelöscht werden, beziehungsweise ihr Speicher freigegeben wird, wenn ihre Sichtbarkeit verlassen wird. Da es in Java keine *Pointer* gibt, die auf primitive Datentypen verweisen können, stellt dies kein Problem dar.

Um festzustellen, ob noch Referenzen auf ein Objekt existieren, müsste der Interpreter sämtliche Symboltabellen durchlaufen und jedes Symbol vom Typ *Symbol_Pointer* daraufhin kontrollieren, ob es auf das zu löschende Objekt beziehungsweise dessen Speicherbereich verweist. Ist dies der Fall, so dürfte zwar der *Pointer* entfernt werden, nicht aber jedoch das Objekt.

Problematisch ist hierbei, daß der Interpreter keine direkte Kenntnis von den existierenden Symboltabellen hat, nur die lokale Tabelle der aktuellen Funktion, sowie die globale Tabelle sind bekannt. Daher müsste der *Garbage Collector* jedesmal, wenn ein *Pointer* erstellt oder gelöscht wird, davon in Kenntnis gesetzt werden, so dieser *Pointer* denn auf eine Klasse verweist.

Das Fehlen von Destruktoren in Java vereinfacht die Arbeit des Collectors, denn so muss dieser nur den für das Objekt reservierten Speicher freigeben und braucht sonst keine weiteren Aktionen durchführen oder Funktionen aufrufen.

Der Collector benötigt demnach drei Methoden, über die er angesprochen werden kann. Es muss dem Interpreter möglich sein, neue Objekte zum *Garbage Collector* hinzuzufügen (`Add_Object`), neue *Pointer* zu registrieren (`Add_Pointer`) und wieder zu entfernen (`Remove_Pointer`).

Wird nun ein Objekt registriert im *Garbage Collector*, so wird ein Eintrag erstellt, der die Adresse und den Typ des neuen Objekts beinhaltet. Wird nun ein neuer *Pointer* auf eben jenes Objekt, das anhand seiner Speicheradresse identifiziert wird, angelegt, wird dessen Zähler, der mit Null initialisiert ist, inkrementiert. Beim Löschen eines *Pointers* wird dieser wiederrum um einen dekrementiert.

Erreicht dieser Zähler nach einer Dekrementierung den Wert Null, wird der Speicherbereich, der von dem Objekt belegt wird, freigegeben. Da Objekte die mit dem `new`-Operator erzeugt wurden, in keiner Symboltabelle stehen, existiert somit auch keine Kenntnis mehr von diesem Objekt.

3.2.3 Sprache: Ada

Mit dem Bauhaus-Tool `ada2iml` ist es möglich aus Ada-Quellcode IML-Graphen zu generieren. Diese unterscheiden sich jedoch erheblich von denen, die aus C- oder C++-Quellcode generiert werden. Die Funktionen sind anders aufgebaut und auch die Datentypen werden zusammen mit Informationen bezüglich ihrer Wertegrenzen im Speicher abgelegt. Das Auslesen der Werte innerhalb des Interpreters berücksichtigt dies momentan nicht.

Da es nicht möglich ist Ada-Packages zur Laufzeit einzubinden, kann auf externe Funktionen nicht zugegriffen werden. Eine Einbindung all der Standardfunktionalitäten, ist aufgrund des Umfangs[6] des Ada-Standards nicht realistisch.

Auf eine Interpretation von Knoten die nur in IML-Graphen, die aus Ada generiert wurden, vorkommen, wurde daher verzichtet. IML-Graphen die Ada-Quellcode repräsentieren werden demnach nicht unterstützt.

3.2.4 Sprache: C++

Die Umsetzung der Interpretation von aus *C++*-Quellcode generierter IML-Graphen orientiert sich an dem *C++*-Standard.

Aufgrund mangelnder Schnittstellen von Ada zu C++, ist es nicht möglich Quellcode zu interpretieren, der die Standard-Template-Library (STL) nutzt (siehe Kapitel 3.2.4.3). Auch Streams und die von C++ eingeführte String-Klasse können so nicht angesprochen und daher nicht verwendet werden. Eine Emulation wäre theoretisch möglich, wäre aber äußerst umfangreich und ist nicht das Ziel dieser Arbeit.

[6] Zum Vergleich: Ada hat über 60 Packages, während C weniger als 150 Standardfunktionen besitzt.

3.2.4.1 Klassen

Klassen stellen ein in C++ neu eingeführtes Konstrukt dar, das sowohl über Konstruktoren als auch Destruktoren verfügt. Der Speicherbereich den eine Instanz einer Klasse belegt, besteht nur aus den Werten seiner Membervariablen, die Funktionen werden an anderer Stelle abgelegt.

Die Verwendung von Klassen wird vom Interpreter unterstützt. Sowohl Konstruktoren als auch Destruktoren werden entsprechend dem Standard emuliert. Genaueres zur Interpretation von Klassen ist Kapitel 3.3.4 zu entnehmen.

3.2.4.2 Speicherverwaltung mit `new` und `delete`

Das Anlegen von Objekten mittels `new` und dessen Freigabe mit `delete` wird vom Interpreter emuliert. Wird ein neues Objekt angefordert, so reserviert der Interpreter einen entsprechend großen Speicherbereich und ruft gegebenenfalls den Konstruktor der Klasse auf.

Der `new`-Operator gibt einen *Pointer* auf das neu erstellte Objekt zurück. Endet die Sichtbarkeit des *Pointers*, wird dieser aus der Symboltabelle und somit aus dem Speicher gelöscht. Da das Objekt das erstellt wurde zwar Speicher belegt aber in keiner Symboltabelle abgelegt wurde, wird der Speicher nicht automatisch freigegeben, wenn die Funktion endet.

Beim Löschen eines Objekts aus dem Speicher mittels `delete` wird, falls es sich bei dem Objekt um eine Instanz einer Klasse handelt, der Destruktor aufgerufen. Anschließend wird der Speicherbereich wieder als verfügbar markiert.

3.2.4.3 Templates

Templates stellen in C eine Möglichkeit dar, generische Klassen und Methoden zu erstellen. Sie werden für einen generischen Datentypen definiert und nur Instanzen mit fest definiertem Datentyp können verwendet werden.

In der IML werden verwendete Templates mit den jeweiligen Datentypen dargestellt. Wird zum Beispiel die Funktion `min`[7] mit den Typen `int` und `double` verwendet, so wird sie zweimal mit den entsprechenden Parametern in der IML abgebildet. Daher sind Templates, die in dem repräsentierten Quellcode definiert sind, ohne Schwierigkeiten zu interpretieren, da sie wie normale Funktionen behandelt werden.

Sollen Templates aus Fremdbibliotheken wie der Standard Template Library (STL) verwendet werden, so ist eine Interpretation nicht möglich. Denn die Interfaces von Ada zu C++ erlauben es nicht, generische Parametertypen anzugeben. Daher muss beim Importieren von externen C++-Funktionen der zu verwendende Datentyp explizit angegeben werden. Da die Templates allerdings beliebige Datentypen erlauben, ist das Importieren aller Optionen zur Zeit nicht möglich, da diese unendlich sind. Eine Verwendung der STL ist daher nicht möglich.

3.2.4.4 Strings

Die String-Klasse von C++ erlaubt eine einfachere und übersichtlichere Verwendung von Zeichenketten. Der für die Verwendung benötigte Header `strings` inkludiert allerdings einige Template-Variablen, deren Unterstützung nicht möglich ist. Daher sind die IML-Generatoren

[7] Gib den kleineren der beiden übergebenen Parameter zurück.

nicht in der Lage aus Quellcode, der diesen Header verwendet, gültige IML zu generieren. Daher ist auch eine Interpretation nicht möglich.

3.3 Repräsentation der Datentypen

Wie die einzelnen Datentypen repräsentiert werden, hat große Auswirkungen darauf, welche Funktionen der Interpreter unterstützen kann. Abstrakte Modelle erleichtern zwar die Darstellung der Typen, erschweren aber grundlegende Operationen.

Alle Datentypen werden vom Interpreter als Symbole behandelt, die auf einen Speicherbereich verweisen und alle nötigen Informationen beinhalten, um sämtliche Operationen auf den Typen zu ermöglichen. Auch unsichere Operationen, wie das Auslesen einer Ganzzahl aus dem Speicherbereich auf den ein Fließkommasymbol zeigt, ist möglich, auch wenn das Ergebnis abhängig von der genutzten Plattform ist.

Zu unterscheiden ist zwischen primitiven und komplexen Datentypen. Während primitive Datentypen atomar sind, also aus keinen anderen Werten zusammengesetzt werden, bestehen komplexe Datentypen aus beliebigen primitiven und komplexen Typen. Welche es gibt und wie diese behandelt werden, ist den folgenden Kapiteln zu entnehmen.

3.3.1 Primitive Datentypen

Unter den primitiven Datentypen versteht man jene die atomar sind, demnach nicht aus mehreren Typen bestehen und in nahezu allen Programmiersprachen vorhanden sind.

Die gängigsten primitiven Datentypen sind `int`, `float`, `double` und `char`, auch wenn sie in einigen Sprachen anders benannt sind. Je nach Sprache kommen noch weitere Typen wie zum Beispiel `byte` und `bool` hinzu.

Diese Typen können noch mit sogenannten *Qualifiern* weiter modifiziert werden, zu denen `unsigned`, `short` oder `long` gehören. Welche primitiven Typen es gibt, ist der Tabelle 3.1 zu entnehmen. Bei den dargestellten Größen der Datentypen wird von einem 32-Bit System und dem gcc-Compiler[8] ausgegangen, da die Größen vom System und dem verwendeten Compiler abhängig sind. Desweiteren ist der Typ `long` den meisten Compilern als `long long` bekannt, `long` ist bei jenen Compilern identisch mit einem unmodifizierten `int`. Der Interpreter verwendet `long` zur Identifikation des 64-Bit Integers.

3.3.2 *Qualifier*

In vielen Programmiersprachen ist es möglich einzelne Variablen mit einem sogenannten *Qualifier* zu versehen. Zu diesen gehören zum Beispiel `signed`, `unsigned`, `volatile`, `static`, `const` und `final`. Die meisten dieser *Qualifier* sind für eine Ausführung des Quellcodes nicht relevant. Nur die *Qualifier* `signed`, `unsigned` und `static` werden vom Interpreter beachtet, da diese direkte Auswirkung auf die Art haben, wie die Variablen interpretiert werden müssen.

Variablen die mit einem `unsigned` markiert sind, gelten als vorzeichenlos. Sie repräsentieren positive Werte und können keinen negativen Wert annehmen. Würde ihr Wert durch eine Operation jedoch in den negativen Bereich fallen, so reagieren die verschiedenen Programmiersprachen unterschiedlich. In Ada[9] wird eine Exception geworfen und in C und C++

[8] http://gcc.gnu.org/

[9] In Ada wird der Typ `unsigned int` als `Natural` bezeichnet.

Tabelle 3.1: Primitive Datentypen

Typ	Größe in Byte		
	C	C++	Java
`boolean`	-	1	1
`byte`	-	-	1
`short int`	2	2	2
`unsigned short int`	2	2	-
`int`	4	4	4
`unsigned int`	4	4	-
`long`	8	8	8
`unsigned long`	8	8	-
`float`	4	4	4
`double`	8	8	8
`long double`	12	12	-
`char`	1	1	1
`unsigned char`	1	1	2
`wchar_t`	-	4	-

wird der eigentliche negative Wert zu dem maximalen positiven Wert addiert. Der *Qualifier* `signed` hingegen steht für eine Variable mit Vorzeichen und stellt den Standard für sämtliche Variablen in C und C++ dar.

Wird eine Variable als `static` definiert, so existiert von dieser Variable zur gesamten Laufzeit nur eine Instanz. Um dies zu realisieren, werden statische Variablen in der globalen Symboltabelle abgelegt. Alle Funktionen die auf diese Variable zugreifen möchten, können durch die Verknüpfung der lokalen Symboltabelle, mit der globalen, auf diese Variable zugreifen.

3.3.3 C-Datentypen

Die Datentypen in C werden im Speicher mit ihrem Wert abgelegt, ohne weitere Informationen wie ihrer Größe oder der Anzahl an Elementen zu beinhalten, falls es sich um ein Array handelt. Ohne das entsprechende Symbol zu dem abgelegten Wert, läßt sich dessen Typ nicht mehr bestimmen.

Array Ein Array stellt eine Sammlung eines bestimmten Datentyps dar, deren Werte im Speicher hintereinander abgelegt werden. Die Größe des Arrays muss schon vor der Laufzeit feststehen, damit ein entsprechend großer Speicherbereich reserviert werden kann. Soll ein Array mit einer zur Laufzeit bestimmten Größe erstellt werden, muss das Array mittels `malloc` bzw. `calloc` erstellt werden.

Das Symbol für ein Array hat zusätzlich zu den Eigenschaften des abstrakten Symbols noch einen Verweis auf den enthaltenen Datentyp und die obere Grenze des Arrays gespeichert. Nur mit Hilfe dieser Informationen läßt sich ein Array wieder vollständig aus dem Speicher entfernen, sollte dessen Sichtbarkeit, am Ende einer Funktion oder eines Blocks, ablaufen.

Pointer Ein *Pointer* stellt einen Verweis auf einen Speicherbereich dar und ermöglich so zum Beispiel die Übergabe von Referenzen auf Objekte an Funktionen. Auch ist es so möglich Operationen direkt auf dem Speicher durchzuführen, ohne Kenntnis von dessen Inhalt zu haben.

Während bei dem abstrakten Speichermodell das *Pointer*-Symbol den Index des Symbols, auf das gezeigt wird, zusammen mit einem Offset beinhaltete, besitzt das Symbol im binären Modell nur einen Verweis auf den Datentyp, auf den gezeigt wird (siehe Abbildung 3.4).

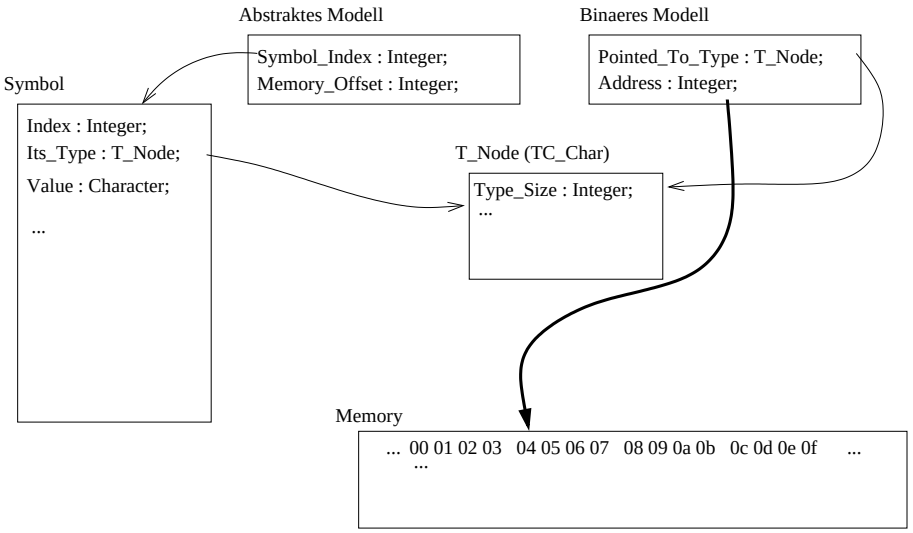

Abbildung 3.4: Beim binären *Pointer*-Modell verweist der *Pointer* auf einen Speicherbereich und kann entsprechend, unabhängig von dem vom ursprünglich benutzten Symbol genutzten Speicherbereichs, verschoben werden. Beim abstrakten Modell wird der *Memory_Offset* auf den `Value` des Symbols angewendet. Dies bedeutet, das ein *Pointer* auf einen primitiven Datentyp nicht verschoben werden kann, da er nur in den Grenzen seines Symbols bleiben muss. Nur bei Arrays oder Konstrukten, die mehrere primitive Datentypen beinhalte, würde dieser *Pointer* verschoben werden können.

Die erste Implementierung, die mit dem Index und einem Offset arbeitete, ermöglichte beliebige Operationen, solange diese nicht den Speicherbereich des Objektes, auf das gezeigt wird, verließen. Da es aber in manchen Programmiersprachen gängige Praxis ist, eben jenen Speicherbereich zu verlassen oder mit *Pointern* zu arbeiten, deren Typus sich von dem Zieldatentyp unterscheidet, wurde diese Version verworfen.

In der aktuellen Version, mit dem binären Speichermodell, besteht das *Pointer*-Symbol aus einem Verweis auf den Datentyp, auf den gezeigt wird, und der Adresse, an der diese Variable im Speicher steht. So verfügt das Symbol über alle wichtigen Informationen um alle möglichen Berechnungen durchzuführen. Durch den Verweis auf den Datentyp kann der *Pointer* entsprechend im Speicher verschoben werden und auch den eigentlichen Speicherbereich der Variable verlassen. Ein *Pointer* hat keine Kenntnis davon, daß er auf ein Element innerhalb eines Arrays zeigt, was im vorherigen Modell der Fall ist.

Der referenzierte Typ gibt desweiteren keine Auskunft darüber, um was für einen Typ es sich tatsächlich bei dem Objekt im Speicher handelt, sondern darüber, wie dieser Speicherbereich interpretiert werden soll, wenn er über den *Pointer* angesprochen wird.

Wäre der Typus des *Pointers* nicht bekannt, würde eine Operation, die den *Pointer* inkrementiert, also im Speicher verschiebt, nicht möglich[10], während er mit der Kenntnis des Typus um jeweils die Größe des Typs inkrementiert werden würde. Zeigt ein *Pointer* zum Beispiel auf einen Integer im Speicher und wird um Eins inkrementiert, so wird

[10] Die Größe des Typs, auf den mit einem `void`-*Pointer* verwiesen wird, ist mit 0 definiert. Daher sind Arithmetiken auf diesem *Pointer* nicht möglich.

er um vier Byte, der Größe eines Integers, verschoben. Dies ermöglicht das leichte Traversieren von Arrays, Listen und Vektoren, wie es sonst nur mit dem abstrakten Modell möglich ist.

Struct Ein Struct ist ein komplexer Datentyp, der aus mehreren Variablen bestehen kann. Dabei muss es sich nicht um primitive Datentypen handeln, sondern es können auch weitere komplexe Typen wie Structs, Unions oder *Pointer* sein.

Die Größe eines Structs ergibt sich aus der Summe der Größen aller seiner Felder. In der IML beinhaltet jedes definierte Struct einen Verweis auf alle seine Felder und auf den jeweiligen Speicheroffset, also jenen Abstand vom Anfang des Feldes im Speicher zu dem Anfang des kompletten Structs. Anhand dieser Informationen läßt sich direkt auf die einzelnen Felder zugreifen, ohne daß eine besondere Behandlung notwendig ist.

Mittels *Pointern* lassen sich auch Operationen auf den Feldern von Structs durchführen, die Auswirkungen auf mehrere Felder durch Überlappung haben. Dies ist sowohl in C als auch in C++ erlaubt und wird ebenfalls vom Interpreter unterstützt.

Union Schwieriger gestalten sich hierbei Unions, die ähnlich wie Structs definiert werden. Der Unterschied zu einem Struct ist, daß alle Felder sich den gleichen Speicherbereich teilen. Die Größe des Unions entspricht dem Maximum der Größen aller Felder. Da zu keinem Zeitpunkt festgestellt werden kann, was für ein Datentyp zuletzt in den Speicherbereich des Unions geschrieben wurde, handelt es sich bei dem Auslesen eines Feldes eines Unions um eine unsichere Operation.

Da es sich bei Ada um eine typensichere Sprache handelt, existiert dort kein Äquivalent. Zwar ist es möglich ähnliche Konstrukte zu erstellen, die in etwa die gleiche Funktionalität haben, allerdings muss der Aufbau des Unions dann zur Kompilierzeit bekannt sein. Daher lassen sich Unions mit dem abstrakten Speichermodell nicht implementieren, ohne den eigentlichen Gedanken des Modells zu verwerfen. Dies war einer der Hauptgründe für den Umstieg auf ein binäres Speichermodell.

Der Interpreter reserviert bei der Erstellung eines Unions ausreichend Speicher, der das größte der repräsentierten Datentypen beinhalten könnte. Wird auf eines der Felder zugegriffen, entspricht dies einem Zugriff auf den Speicherbereich mit dem entsprechenden *Pointer*. Im Grunde handelt es sich bei dem Union um ein Struct, bei dem alle Offsets der einzelnen Felder gleich Null sind.

Funktionspointer In C gibt es die Möglichkeit, *Pointer* auf einzelne Funktionen zu erstellen und diese wie normale *Pointer* zu verwalten. So können sie sowohl als Parameter als auch als Rückgabewert dienen und zur Erstellung von generischen Funktionen beitragen.

Bei der Erstellung eines solchen *Pointers* in C muss dessen Parameter und Rückgabewerte angegeben werden, Adressen von entsprechenden Funktionen werden dann akzeptiert und verwendet.

Problematisch gestaltet sich hierbei die Weitergabe von Funktionspointern an C-Standardfunktionen, die nur aufgerufen, aber, abgesehen von den Parametern, nicht beeinflusst werden können. Denn die in der IML beschriebenen Funktionen existieren zur Laufzeit nicht im Speicher, sondern werden interpretiert. Das heißt mit "realen *Pointern*" auf zu interpretierende Funktionen, läßt sich nicht arbeiten.

Die einzige Methode, abgesehen von der hier nicht vorliegenden Adresse, anhand der Funktionen eindeutig identifiziert werden können, ist deren IML-Index. Ein Funktionspointer beinhaltet statt der Adresse der Funktion daher den Index des *O_Routine*-Knotens, der zu der jeweiligen Funktion gehört.

Bei dem Aufruf einer Funktion über einen *Pointer* mittels des *Indirect_Call*-Knotens, wird aus dem *Pointer* der Index des *O_Routine*-Knotens ausgelesen, der die aufzurufende Funktion repräsentiert. Nachdem der entsprechende Knoten aus dem Graphen herausgesucht wurde, werden wie gewohnt die Parameter in die Funktion kopiert.

Die Parameter, die von dem *Indirect_Call*-Knoten referenziert werden, besitzen einen anderen Index als die, der eigentlich aufgerufenen Funktion. Denn sie repräsentieren die generischen Parameter, die jede Funktion, die mit dem Funktionspointer referenziert werden kann, besitzen muss. Die eigentliche dann ausgeführte Funktion, kann allerdings nur die eigenen Parameter anhand ihrer ID nutzen, da er keine Kenntnis von den IDs der generischen Parameter besitzt. Um dieses Problem zu umgehen, müssen die IDs der kopierten Parameter entsprechend angepasst werden.

Hierbei gilt allerdings zu beachten, daß die zu kopierenden Parameter andere IDs haben, als die Parameter der Funktion die über den Pointer aufgerufen wird. Dies liegt daran, daß die IDs der Parameter die der generischen Funktion sind, die durch die aufzurufende ersetzt wird. Daher müssen die IDs der Parameter entsprechend angepasst werden, damit diese denen der Parameter, der über den Pointer verwiesenen Funktion, widerspiegeln.

Es lassen sich daher nur interne Funktionen über Funktionspointer ansprechen. Das weiterreichen von internen Funktionspointer an externe Funktionen ist nicht möglich. Genauso ist es nicht möglich, die Adresse externer Funktionen zu ermitteln und diese so an weitere externe Funktionen weiterzugeben. Auch eine interne Behandlung über Funktionspointer von externen Funktionen ist nicht möglich.

Dateizeiger, `FILE*` Da sich nicht alle C-Dateifunktionen direkt aufrufen lassen (siehe Kapitel 3.4.2.1.1), wie es für viele andere C-Funktionen der Fall ist, kann nicht mit dem normalen `FILE`-Datentyp von C gearbeitet werden. Stattdessen wird der Datentyp `FILE*` (im Grunde nur eine Systemadresse) des C_Streams-Interfaces von Ada verwendet und im Speicher dessen Wert abgelegt.

So liegt das eigentliche Dateiobjekt nicht im binären Speicherbereich des Interpreters. Nur ein Verweis auf das Objekt wird dort abgelegt und repräsentiert so den `FILE`-Pointer, der für die I/O-Operationen der Standardbibliothek verwendet wird.

3.3.4 C++-Datentypen

Da es sich bei C++ um eine Erweiterung der Programmiersprache *C* handelt, existieren auch bei ihr alle Datentypen, wie es sie bei *C* gibt. Erweitert wurden die primitiven Datentypen um die folgenden.

Boolesche Variable: `bool` Bei dem `bool` handelt es sich um einen binären Datentyp, der entweder den Wert `true` oder `false` annehmen kann, was in *C* bisher auf `1` beziehungsweise `0` abgebildet wurde.

Weite Zeichen: wchar_t Dieser Datentyp belegt vier Bytes im Speicher, während ein normaler `char` nur einen Byte belegt. So können auch komplexere Zeichen, zum Beispiel Unicode, abgebildet werden. Da der Typ in der IML aber nicht entsprechend behandelt wird, ist eine Interpretation nicht möglich. Stringliterale des Typs werden wie normale Zeichenketten interpretiert, so daß aus dem String aus dem Listing 3.3 intern
`\0\0\0T\0\0\0e\0\0\0s\0\0\0t`
wird. Dabei entspricht das erste Byte nicht, wie es den Anschein hat, dem Nullbyte, sondern dem ASCII-Zeichen \, gefolgt von dem ASCII-Zeichen `0`.

Listing 3.3: String bestehend aus `wchar_t`

```
1 wchar_t *my_string = L"Test";
```

Daher ist die Verwendung von diesem Datentyp zu diesem Zeitpunkt nicht möglich.

Typendefinitionen Bei einem `typedef` handelt es sich um einen Alias für einen schon existenten Typen. Dies dient sowohl der Übersichtlichkeit und der Vereinfachung als auch zur Verwendung in Spezialfällen, wenn der eigentliche Datentyp nicht benutzt werden kann. Dies ist zum Beispiel bei der Verwendung von Pointern der Fall, wenn das `*`-Zeichen, das sie kennzeichnet, vom Parser des Compilers oder dem Nutzer selbst nicht immer eindeutig zugeordnet werden kann.

So ist auf den ersten Blick der Typ der folgenden Variablen nicht sofort klar erkennbar:

Listing 3.4: Übersichtlicher mit `typedef`s

```
1 int* x, y;
2
3 typedef int* int_ptr;
4 int_ptr x, y;
```

Es scheint als hätten sowohl `x` als auch `y` den Typ `int*`, wobei das `*` aber nur dem `x` zugeordnet wird und `y` somit vom Typ `int` ist. Die Lösung mit Hilfe des `typedef`s ist hingegen eindeutig.

Der Parser des Compilers stößt auf Probleme, wenn zum Beispiel ein Vektor eines Vektors definiert wird (`vector<vector<int>> x`). So endet die Definition mit dem Schließen der beiden Vektoren, was ebenfalls als Shift-Operator erkannt werden kann (`>>`).

Stößt der Interpreter auf den Datentyp `typedef`, so wird dieser zu seinem eigentlichen Typen aufgelöst. Zuständig hierfür ist die Methode `Resolve_Type` in der Unit `Symbols`.

Enumeratoren In der IML werden *Enum*s als konstante Variablen die auf Integer-Konstanten verweisen abgelegt, so entsteht eine Abbildung von identifizierbaren Werten auf Integer. Da diese Integer-Konstanten in der IML durch keinen Initialisierungsaufruf erreichbar sind, weiß der Interpreter erst von deren Existenz, wenn sie während der Ausführung verwendet werden. Dies stellte ein Problem dar, da so auf ein Symbol zugegriffen werden soll, das noch nicht initialisiert wurde. Um dies zu umgehen, wird bei dem Abfragen der Symboltabellen überprüft, ob es sich bei dem gesuchten Symbol um ein *Enum* handelt und in dem Fall zur entsprechenden Integer-Konstante aufgelöst.

Klassen Eine Klasse besteht, in Bezug auf den Speicher den sie belegt, aus einer Ansammlung von Membervariablen. Diese werden, in der Reihenfolge in der sie definiert wurden, im Speicher abgelegt. In der IML ist jede dieser Membervariablen mit einem *Offset* versehen, der Abstand zwischen dem ersten Byte der Klasse und dem ersten Byte der Variable. Anhand dieses *Offsets* läßt sich so die Position der Membervariable im Speicher bestimmten und diese auslesen und manipulieren.

Beim Aufruf einer Funktion eines Objektes, wird ein Pointer auf dieses Objekt als erster Parameter in den Funktionsaufruf kopiert. Dieser Pointer repräsentiert den `this`-Pointer und wird immer dann angesprochen, wenn auf Membervariablen oder -funktionen zugegriffen werden muss. Abgesehen von diesem zusätzlichen Parameter, verhalten sich die Funktionen wie reguläre Methoden (siehe Kapitel 3.4).

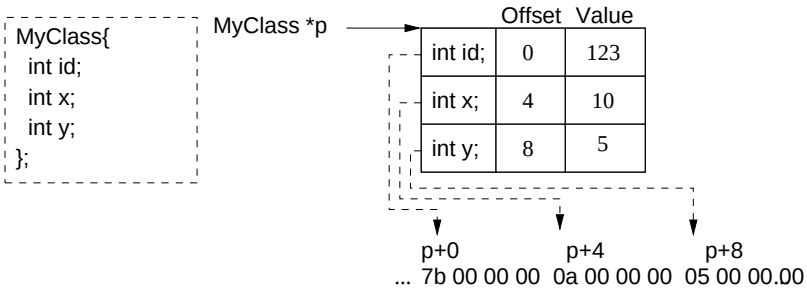

Abbildung 3.5: Die Klasse `MyClass` beinhaltet drei Membervariablen jeweils vom Typ `int`. Der Offset der jeweiligen Variablen ist daher ein Vielfaches von Vier. Anhand dieses Offsets kann der Interpreter die Lage der einzelnen Membervariablen im Speicher ermitteln.

3.4 Funktionen

Obwohl im Folgenden sowohl Methoden als auch Funktionen gemeint sind, wird der Einfachheit halber sowohl das Wort Funktion, Methode als auch Routine benutzt. Dabei sind, wenn nicht anders angegeben, Funktionen mit als auch ohne Rückgabewert gemeint.

Die Funktionen sind in zwei Kategorien aufgeteilt, die internen Funktionen und die externen. Die internen Funktionen werden vollständig in der IML abgebildet, da der IML-Generator kompletten Zugriff auf die Quelldateien hat. Bei externen Funktionen liegen diese nicht vor, weswegen der Generator die eingebundenen Header-Dateien nutzt, um die Signatur der Funktion in der IML abzubilden. Wie diese beiden Arten von Funktionen behandlet werden, ist in den folgenden Kapitel beschrieben.

3.4.1 Interne Funktionen

Die eigentliche Interpretation des IML-Graphen geschieht im so genannten *FunctionCall*, einer Klasse die alle relevanten Informationen über einen Funktionsaufruf und dessen Symboltabellen beinhaltet. Der Typ des Rückgabewertes (falls vorhanden) und die einzelnen Parameter stehen alle in dem referenziertem IML-Knoten vom Typ *O_Routine*.

Wird eine Funktion aufgerufen, prüft der Interpreter, ob diese Parameter besitzt oder es sich dabei um einen parameterlosen Aufruf handelt. Benötigte Parameter werden aufgelöst und entsprechend ihrem IML-Index in die Symboltabelle des neuen Aufrufs kopiert.

Jede Routine verweist auf eine *Statement_Sequence*, eine Liste vom Typ *Statement*, die vom Interpreter Schritt für Schritt ausgeführt wird. Diese *Statement*-Knoten werden an die generische `ExecuteStatement`-Funktion des Interpreters übergeben, die entsprechende Aktionen ausführt. Muss ein Knoten zu einem Wert aufgelöst werden, so wird dieser an die generische `ResolveStatement`-Funktion weitergeleitet, die dann den jeweiligen Wert als Symbol zurückgibt. Die `EvaluateStatement`-Funktionen des Interpreters lösen beliebige IML-Knoten in *wahr* oder *falsch* auf, repräsentiert durch ein *Boolean*-Symbol.

Da es sich um eine interne Funktion handelt, sind alle enthaltenen Operationen in der IML abgebildet. Aufrufe weiterer, externer Funktionen, die nicht in dem repräsentiertem Quellcode definiert sind, werden vom Interpreter nicht direkt behandelt.

Jede dieser Funktionen verfügt über eine seperate Symboltabelle, in der alle Symbole gespeichert werden, die zusätzlich zu den Parametern während der Ausführung generiert werden. Dieses Symboltabelle ist in der Regel nur mit der globalen Symboltabelle verknüpft, außer

es handelt es sich um eine Unterfunktion. In dem Fall ist sie noch mit der Symboltabelle der umgebenden Funktion verknüpft, um auf dessen Symbole und somit dessen Variablen zugreifen zu können.

Am Ende einer Funktion wird die `clean`-Methode der jeweiligen Symboltabelle aufgerufen, was zur Löschung aller dort abgelegten Symbole aus dem Speicher führt. Dies dient zur Aufspürung von Speicherlecks, sowohl in dem durch die IML repräsentierten Quellcode, als auch im Interpreter selbst. Handelt es sich bei den noch vorhandenen Symbolen um temporäre, so hat der Interpreter es versäumt sie zum gegebenen Zeitpunkt zu löschen. Anderenfalls liegt das Speicherleck im Quellcode oder in der IML-Repräsentation, da normalerweise alle Symbole dessen Ende der Sichtbarkeit am Ende der Funktion erreicht wurde, mittels des vorher erreichten *End_Of_Lifetime*-Knoten (siehe Kapitel A.1.1.2) gelöscht werden.

3.4.2 Externe Funktionen

Sollte das repräsentierte Programm externe Funktionen nutzen, also jene die nicht im Programm selbst definiert sind, so sind sie auch nicht in der IML abgebildet. Die Signatur der Funktion und welcher Parameter wie gefüllt wird, ist der Darstellung des Aufrufs in der IML jedoch zu entnehmen.

Da diese Funktionen selbst nicht repräsentiert werden, können diese auch nicht interpretiert werden. Diese Funktionen werden über die Schnittstellen des Interpreters aufgerufen, um so das gewünschte Ergebnis zu erzielen.

Bei der Verwendung von Standardbibliotheken, wie sie zum Beispiel bei C gegeben sind, lassen sich die vorhandenen Schnittstellen nutzen, um auf diese zuzugreifen. Falls in der entsprechenden Sprache die Möglichkeit fehlen sollte, Bibliotheken zur Laufzeit einzubinden, so ist es nicht möglich, Drittbibliotheken zu nutzen. Da zur Nutzung externer Funktionen diese entsprechend vom Interpreter eingebunden und aufgerufen werden müssen, was voraussetzt, daß sie schon zur Kompilierzeit bekannt und auch zur Laufzeit verfügbar sind. Dies bedeutet, daß gewünschte Drittbibliotheken beim Kompilieren vorliegen müssen und deren Aufruf in den Quellcode des Interpreters integriert werden muss.

Wie die Möglichkeiten bei den jeweiligen Sprachen aussehen, wird in den folgenden Kapiteln erläutert.

3.4.2.1 Externe Funktionen in C

Dank der Fähigkeit von Ada indirekt auf C-Funktionen über die integrierten Schnittstellen (siehe Anhang B des Ada95 Referenzhandbuchs [**?**, ada95]. zuzugreifen, lassen sich viele Funktionen des C-Standards einbinden. Allerdings muss für die einzubindende Funktion schon zur Kompilierzeit ihre Signatur bekannt sein.

Das Einbinden von C-Bibliotheken zur Laufzeit ist nicht möglich. Daher beschränkt sich der Interpreter auf die C-Standardfunktionen, die Darstellung von Grafik ist daher zum Beispiel nicht möglich, da diese vom Betriebssystem abhängig und nicht Teil des C-Standards ist.

Erwartet eine Funktion als Parameter einen *Pointer*, wird ihr die Systemadresse des Objektes übergeben, auf das der eigentliche *Pointer* verweist. Die Systemadresse ist vollkommen ausreichend, weitere Informationen über die Art des *Pointers* sind nicht notwenig. Diese würden nur benötigt werden, wenn auf das Objekt innerhalb des Interpreters zugegriffen werden soll. Dies geschieht jedoch nur in den externen Funktionen. Nach der Übergabe der Systemadresse findet automatisch eine implizite Konvertierung des *Pointer*-Typs statt.

Einige Funktionen, wie zum Beispiel `strtol` (siehe Tabelle B.7), geben Pointer zurück oder schreiben Speicheradressen in übergebene Parameter. Diese Adressen beziehen sich auf den Speicher des Interpreterprozesses und nicht, wie beim Speichermodell des Interpreters vorgesehen, auf den Index des binären Speicherarrays. Daher müssen anschließend an den Funktionsaufruf die Pointer in das Format des Interpreters konvertiert werden. Hierzu muss von dem vorhandenen Wert der des ersten Elements des Speicherarrays abgezogen werden.

3.4.2.1.1 Ein- und Ausgabe Die Ein- und Ausgabe über Dateien und die Konsole ist größtenteils in den C-Schnittstellen von Ada enthalten. Ausgenommen davon sind die Methoden `printf`, `scanf` und ihre Äquivalente, da diese mit variablen Parametern (Typ und Anzahl) arbeiten, was für Ada ein Problem darstellt (siehe Kapitel 3.4.2.1.3).

Für die Dateiein- und Dateiausgabe wird teilweise die *C_Streams*-Schnittstelle verwendet, die die meisten aller wichtigen C-Standard-I/O-Funktionen bereitstellt. Welche Funktionen unterstützt werden, ist dem Kapitel 3.2.1.1 im Abschnitt zu `stdio.h` zu entnehmen.

3.4.2.1.2 Signal handler Die Aufgabe des *Handlers* ist es, Signale vom System abzufangen und entsprechend zu verarbeiten. Zu den möglichen Signalen gehört zum Beispiel `SIGINT`, das auftritt, wenn der Benutzer das Programm mit Hilfe der Tastenkombination `CTRL-C` abbricht.

In C ist es die Funktion `signal`, die es ermöglicht, einem Signal eine Funktion zuzuordnen. Wird das entsprechende Signal empfangen, wird die zugeordnete Funktion aufgerufen. Mit der entsprechenden `raise`-Funktion, kann solch ein Signal auch manuell gesendet werden.

Signale die anhand der `raise`-Funktion manuell gesendet werden, können vom Interpreter auch abgefangen und entsprechend interpretiert werden, da es sich dabei um einen externen Funktionsaufruf handelt. Jene Signale die hingegen vom System selbst gesendet werden, müssen mit dem *Interrupt-Handler* von Ada abgefangen werden. Die Handlerfunktionen müssen zwar separat für jedes Signal angegeben werden, da diese aber auf parameterlose Funktionen verweisen, gibt es keine Möglichkeit innerhalb der Funktion festzustellen, welcher Interrupt zu der Ausführung führte. Ein Verweis auf die aufzurufenden Methoden ist nicht möglich, da diese interpretiert werden und nirgends im Speicher liegen (siehe Kapitel 3.3.3).

Daher muss als Handlerfunktion anstelle der Funktion eine Hilfsfunktion aufgerufen werden, die ermittelt, welche Funktion an den Interrupt eigentlich gebunden ist und diese entsprechend interpretieren. Da diese per *Interrupt* aufgerufene Funktion keinerlei Kenntnis von dem Typ des Interrupts hat und es auch keine Möglichkeit gibt, diesen zu ermitteln, ist es nicht möglich, die zu interpretierende Funktion herauszufinden.

Zwar ist es alternativ möglich, für jeden existierenden Signaltyp eine eigene Wrapperfunktion zu schreiben. Da der Signaltyp allerdings hunderttausende Werte annehmen kann, sind entsprechend viele Funktionen notwendig, was äußerst ineffizient ist.

Daher ist eine Interpretation von Interrupts des Systems nicht möglich.

3.4.2.1.3 Einschränkungen Obwohl die Schnittstellen von Ada nach C eine große Funktionalität aufweisen, so decken sie nicht alles ab. Dies ist zum Teil auf die Typensicherheit von Ada zurückzuführen, die sich nicht auf alle unsicheren Operationen von C abbilden läßt. Im Folgenden sind die aufgetretenen Einschränkungen aufgeführt.

Variable Parameteranzahlen In C und C++ gibt es die Möglichkeit, Funktionen mit variabler Parameterzahl zu erstellen. Die Anzahl der möglichen Parameter ist dabei

theoretisch unbegrenzt. Innerhalb der Standardbibliothek von C wird diese Option von den `printf`- und `scanf`-Methoden benutzt, um die darzustellenden beziehungsweise einzulesenden Variablen zu übergeben, deren verwendete Anzahl vom Aufbau des Formatierungsstrings abhängt.

Eine derartige Funktionalität ist bei Ada nicht gegeben und daher muss dies durch mehrfache Einbindung der jeweiligen Funktion mit den entsprechenden Parameterzahlen geschehen. Dies bedeutet, daß die Implementierung im Interpreter eine maximale Anzahl an solchen Parametern festsetzen muss. Sollte ein auszuführender IML-Graph allerdings mehr Parameter voraussetzen, muss der Interpreter die Ausführung abbrechen.

Bei einigen Funktionen ist es möglich, den Aufruf in mehrere Einzelaufrufe zu zerlegen. Über das Interface wird die Methode so mit nur je einem optionalen Parameter mehrmals aufgerufen. Zu den Funktionen, die so eingebunden werden, gehört `scanf`, wie in Kapitel 3.4.2.1.5 beschrieben wird.

Innerhalb der IML wird dies dargestellt, indem bei dem Aufruf der jeweiligen Funktion zum Kopieren des Parameters ein *Copy_In_Ellipsis*- anstelle eines *Copy_In*-Knoten verwendet wird, der keinen Verweis auf einen *O_Parameter*-Knoten hat (siehe Kapitel A.3.2.2). Ohne diesen Verweis kann dem Parameter kein eindeutiger IML-Index aus dem Graphen zugeordnet werden, weswegen ihm ein fortlaufender Index zugewiesen wird, der für jede einzelne Funktion eindeutig ist. Somit ist es möglich, beim Aufruf der Schnittstellen auf die einzelnen zusätzlichen und optionalen Parameter zuzugreifen.

Variable Parametertypen Ein noch größeres Problem stellen dynamische Parametertypen dar, wie sie zum Beispiel bei den Funktionen `printf` oder `sprintf` auftreten. Hier ist nicht nur die Anzahl der Parameter variabel, sondern auch deren Typus. Ähnlich wie bei den Funktionen mit variabler Parameteranzahl, muss jede mögliche Parameteranzahl mit allen möglichen Parametertypen kombiniert und letztendlich definiert werden. Daraufhin muss dann eine Abbildung des Speichers des Interpreters auf eben jene Parameter im Format der C-Datentypen geschehen.

Eine Emulation der Funktionen mit ihren Grundeigenschaften ist daher die elegantere Lösung. Zwar handelt es sich bei den genannten Methoden um recht umfangreiche Funktionen, jedoch haben alle diese Ausgabefunktionen große Teile gemeinsam. Das gleiche gilt entsprechend für die jeweiligen Eingabefunktionen (zum Beispiel `scanf` und `fscanf`).

Wie das Problem für die Methoden `printf`, `sprintf` und `fprintf` gelöst wurde, ist dem Kapitel 3.4.2.1.4 zu entnehmen. Anstelle der Emulation werden die möglichen Kombinationen der Parameter der `scanf`-Methode für bis zu 100 zusätzliche Parameter vom Interpreter verstanden. Werden mehr Parameter angegeben, so wird die Ausführung abgebrochen. Bei den `printf`-Methoden funktioniert dieser Ansatz nicht, da es sich bei dessen optionalen Parametern nicht um Pointer beziehungsweise Referenzen handelt.

Wie die `printf`-Methoden angesprochen werden, ohne dabei ihre Funktionalitäten zu beeintächtigen, ist in Kapitel 3.4.2.1.4 beschrieben.

Vordefinierte Datenkonstrukte Einige der C-Standardfunktionen erwarten als Parameter ein vordefiniertes Datenkonstrukt oder geben ein solches zurück. Zum Beispiel das `tm`-Konstrukt aus dem `time.h`-Header, ein *Struct* (siehe Kapitel 3.3.3) das mehrere Informationen über einen bestimmten Zeitpunkt beinhaltet. Der Aufbau dieser Konstrukte ist abhängig vom verwendeten Compiler; auch wenn diverse Implementierungen sich ähneln mögen, so ist nicht mit Sicherheit zu sagen, daß alle identisch sind.

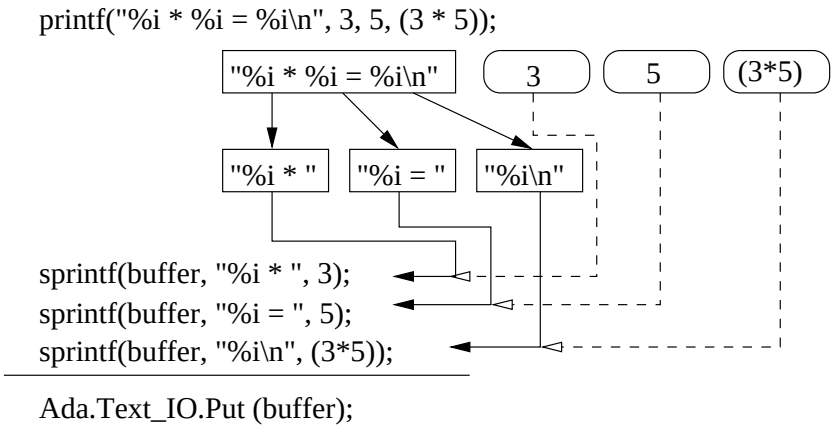

Abbildung 3.6: Der Aufruf von `printf` wird in mehrere Teile zerlegt und mittels einzelner `sprintf`-Aufrufe in Teilstrings umgewandelt, die am Ende wieder zusammengefügt und dann ausgegeben werden.

Der Standard gibt vor, wie vordefinierte *Structs* mindestens aufgebaut sind, woran sich der Interpreter orientiert. Da nur solche Datenkonstrukte eingebunden werden können, deren Aufbau schon vorher bekannt ist, können vordefinierte Datenkonstrukte aus anderen als den Standardbibliotheken nicht verwendet werden.

In der IML sind die verwendeten vordefinierten Datentypen der eingebundenen Header-Dateien abgebildet und können auch verwendet werden, solange keine externen Funktionen genutzt werden, die solch ein Konstrukt als Wert übergeben bekommen. Erwartet eine Funktion einen *Pointer* auf einen vordefinierten Datentyp, so stellt dies kein Problem dar, da in dem Fall ein *Pointer* auf den Speicherbereich innerhalb des Interpreters übergeben wird. Bei der Übergabe des Parameters als Wert (anstatt als Referenz oder *Pointer*) muss der Datentyp dem Interpreter allerdings als interner Datentyp bekannt sein, der schon definiert war, als der Interpreter kompiliert wurde. Dies ist nur bei den Datentypen aus den Standardbibliotheken der Fall, die bei solch einem Aufruf extra angelegt und mit den Daten aus dem Speicher des Interpreters gefüllt werden.

3.4.2.1.4 Funktionen `printf`, `sprintf`, `fprintf` Diese Methoden verfügen über eine variable Anzahl an Parametern, deren Typen nahezu beliebig sind. Solche Funktionen werden weder von Ada noch von dessen Schnittstellen zu C unterstützt. Daher ist eine direkte Einbindung über die Schnittstellen nicht möglich und eine komplette Emulation ist aufgrund der Komplexität der Methoden zu aufwendig. Da das Verhalten der Funktionen allerdings bekannt ist, ist ein anderer Ansatz möglich, der im Folgenden erläutert wird.

Wie die optionalen Variablen interpretiert werden sollen, kann dem *Format-String* der jeweiligen Methode entnommen werden. Dies ermöglicht es, diesen in seine Einzelteile zu zerlegen und so zu ermitteln wieviele Parameter verwendet werden und um was für Typen es sich dabei handelt.

Hierzu wird der *Format-String* in seine Einzelteile zerlegt[11]. Jedem dieser Teile wird einer der Parameter zugeordnet, so daß daraus einzelne `sprintf`-Aufrufe generiert werden, die einen zum Format zusätzlichen Parameter übergeben bekommen. Die Ergebnisse der einzelnen `sprintf`-Aufrufe werden an einen Buffer angehängt, so daß dieser am Ende vom Interpreter verwendet werden kann.

[11] Als Trennzeichen dient dabei das `%`-Zeichen

Kommt in einem Teilstring kein Prozentzeichen vor, so wird diesem Teil kein Parameter zugeordnet und der Text an den Buffer angehängt. Ansonsten wird der String nach dem ersten `%` durchsucht und das erste Vorkommen eines der Parameterzeichen[12] bestimmt den Typ des Parameters. Besonders behandelt werden müssen hierbei im Formatstring die Vorkommen von `%%` und `%n`. Während `%%` als einfaches Prozentzeichen interpretiert werden soll, was der Interpreter ohne Aufruf der C-Funktion übernimmt, wird bei `%n` die bisherige Anzahl an geschriebenen Zeichen in den Parameter, einem *Pointer* auf einen Integer, geschrieben. Auch dies wird vom Interpreter selbst übernommen, der die aktuelle Länge des Buffers in den Parameter einträgt.

Handelt es sich bei dem zu interpretierenden Aufruf um die Funktion `printf` so wird der erzeugte Buffer auf die Konsole ausgegeben. Bei der `fprintf`-Methode wird der Buffer mit der Methode `fputs` in die angegebene Datei geschrieben. Das Kopieren des Buffers in den ersten Parameter des `sprintf`-Aufrufs übernimmt komplett der Interpreter.

Zwar ist es möglich den resultierenden Buffer als Format-Parameter an die jeweiligen eigentlichen Funktionen zu übergeben, dies führt aber zu Problemen, wenn zum Beispiel folgender String ausgegeben werden soll: `%%i`. Der resultierende Buffer wäre `%i` und ein Aufruf von einer der drei Methoden würde dies nun so auswerten, als wenn ein weiterer Parameter erwartet wird, anstatt den Buffer so zu verwenden.

Bei der Bestimmung des Parametertypen wird das erste Vorkommen eines der "Parameterzeichen" im Formatstring nach dem Prozentzeichen gesucht und nicht das erste Zeichen direkt dahinter. Mit diesem Verfahren ist es möglich auch komplexere Formate wie `%+-#010.2f` korrekt zu erkennen. Interpretiert werden diese Formatierungsangaben dann direkt von den C-Funktionen.

3.4.2.1.5 Funktionen `scanf`, `sscanf`, `fscanf` Die Funktion `scanf` akzeptiert beliebig viele Parameter, wobei diese als Referenz, also als *Pointer*, übergeben werden. Daher ist es für den Aufruf der Funktion über das Interface irrelevant, um was für Datentypen es sich dabei handelt.

Das Problem der beliebig vielen Parameter bleibt aber dennoch bestehen, auch wenn deren Typ ignoriert werden kann. Eine Zerlegung wie bei `printf` ist nicht möglich, da zwar der Format-String sauber getrennt werden kann, dies aber nicht für den Eingabestring, auf den dieses Format angewendet werden soll, gilt.

Um eine solche Trennung durchzuführen, wird unabhängig davon, ob es sich bei der aufgerufenen Funktion um `scanf`, `sscanf` oder `fscanf` handelt, der zu parsende String gebuffert. Dieser Buffer wird jedesmal, wenn ein Teil des Formatstrings gefunden wird, um diesen gekürzt. Allerdings ist dies nicht möglich, da anhand des Formatstrings keine Aussage darüber getroffen werden kann, wie der abgeglichene Teilstring exakt aussieht oder wie lang er ist. Daher läßt sich ohne übertriebenen Mehraufwand nicht sagen, wo der Teil des Buffers beginnt, der noch nicht abgeglichen wurde.

Zur Verdeutlichung soll das folgende Beispiel dienen. Es soll der String "`10 products, id 14, cost 140.00$`" mit dem Formatstring "`%i products, id %i, cost %f$`" gescannt werden. Bei der Zerlegung des Formatstrings würden die einzelnen Teilformate entstehen: `%i products, id ` , `%i, cost ` und `%f$`. Wird der erste Formatstring auf den Buffer angewendet, wird die erste Variable eingelesen, was dem Interpreter vermitteln sollte, das mit dem nächsten fortgefahren werden kann. Nun muss zuerst der Bufferstring entsprechend verkürzt werden, damit der zweite Teilformatstring nicht auf den gleichen Teil des Buffers angewendet

[12] Den Parametertypen bestimmtende Zeichen: `p, i, d, u, x, X, o, c, f, e, E, g, G, s`

wird.

Was in diesem Fall kein großes Problem ist, da durch die konstanten gesuchten Strings ausreichend Unterscheidung möglich ist, werfen Formate wie `%i %i %i` erhebliche Probleme auf.

Daher verfolgt der Interpreter einen anderen Ansatz. Es werden eine bestimmte Anzahl an Funktionen bereitgestellt, um die jeweilige `scanf`-Funktion mit der entsprechenden Parameterzahl aufzurufen. Das entsprechende Package (`Scanfs`) wird automatisch generiert (siehe Kapitel 3.5.1.2) wobei als Kommandozeilenparameter angegeben werden muss, wieviele Parameter letztendlich unterstützt werden sollen.

Wird vom Interpreter aus eine `scanf`-Methode aufgerufen, die mehr als die zuvor generierten Parameter benötigt, beendet der Interpreter die Ausführung des IML-Graphen mit entsprechender Fehlermeldung.

Dies gilt entsprechend für die Methoden `sscanf` und `fscanf`, die sich nur minimal von `scanf` unterscheiden. Die Funktion `sscanf` erwartet einen weiteren Parameter, der den String beschreibt, der ausgewertet werden soll. Bei `fscanf` wird der extra angegebene Stream anstelle von `stdin` verwendet.

3.4.3 Arithmetische Konvertierungen

Alle grundlegenden Operationen, wie Mutliplikation und Addition, sind nur für gleiche Typen definiert. Unterscheiden sich die Typen der beiden Operatoren, so wird der mit niedrigerer Wertigkeit zu dem mit der höheren konvertiert.

Bei dieser Konvertierung werden in C/C++ folgende Regeln angewendet (siehe [13, Seite 253]). Da Java nicht über den Typ `unsigned` oder `long double` verfügt, fallen dort die entsprechenden Regeln weg. In Ada müssen diese Konvertierungen vom Programmierer selbst explizit vorgenommen werden, da der Compiler dies nicht übernimmt.

1. Variablen vom Typ `char`, `unsigned char`, und `short` werden zu `int` umgewandelt
2. Ist ein Operand vom Typ `long double`, so wird auch der zweite Operand zu `long double` gecastet
3. Ist ein Operand vom Typ `double`, so wird auch der zweite Operand zu `double` gecastet
4. Wenn sonst einer der Operanden vom Typ `unsigned long` ist, so wird auch der zweite Operand zu `unsigned long` gecastet
5. Wenn sonst einer der Operanden vom Typ `long` ist, so wird auch der zweite Operand zu `long` gecastet
6. Wenn sonst einer der Operanden `unsigned` ist, so wird auch der zweite Operand zu `unsigned` gecastet
7. Ansonsten müssen beide Operanden vom Typ `int` sein und dies ist dann auch das Ergebnis der Operation

Innerhalb der IML sind diese Konvertierungen nicht direkt abgebildet (siehe Kapitel 4.4.1.1). Nur der Typ des Knotens, der eine Berechnung zweier unterschiedlicher Datentypen repräsentiert, gibt Aufschluß darüber, welchen Typ das Ergebnis haben soll. Anhand diesen Typs, kann dann die Konvertierung, die für eine Berechnung notwendig ist, durchgeführt werden.

3.4.4 Reihenfolge der Auswertung

In welcher Reihenfolge verschiedene Operatoren in einem Ausdruck ausgewertet werden, bestimmt die IML. Das bedeutet, daß ein Ausdruck wie zum Beispiel `3 * (4 + 6)` im IML-Graphen eine strikte Auswertungsreihenfolge vorgibt. So wird zu erst `(4 + 6)` ausgewertet und anschließend die Multiplikation aufgelöst.

Nicht vorgegeben in der IML ist die Reihenfolge, in der die einzelnen Operanden einer Operation aufgelöst werden. Stößt der Interpreter auf einen Ausdruck wie zum Beispiel `a[3] + b[6]`, ist nicht definiert ob zuerst `a[3]` oder `b[6]` aufgelöst werden soll. In vielen Sprachen wird wie in der Mathematik üblich von links nach rechts aufgelöst, so daß in diesem Fall zuerst `a[3]` dereferenziert wird. Dies mag hier keinen Unterschied machen, da jedoch beliebig komplexe Operationen Teil eines Operanden sein können, in denen sogar Funktionsaufrufe und Zuweisungen vorkommen, ist die Reihenfolge nicht mehr irrelevant.

Ein solches Beispiel ist dem Listing 3.5 zu entnehmen. Das Ergebnis der Multiplikation in der zweiten Zeile ist von der verwendeten Programmiersprache und dem Compiler abhängig. Während Java strikt von links nach rechts auflöst, ist die Reihenfolge für C und C++ nicht definiert (siehe [13, Seite 86]) und entsprechend unterscheiden sich die Ergebnisse.

Auch in dem zweiten Beispiel (Listing 3.6) könnte am Ende entweder `v[1]` oder auch `v[2]` modifiziert werden, je nachdem ob die Inkrementierung des `++`-Operators vor oder nach der Zuweisung ausgeführt wird.

Listing 3.5: Auswertungsreihenfolge: Beispiel 1

```
1 int k = 5;
2 k *= k++ + k;
```

Listing 3.6: Auswertungsreihenfolge: Beispiel 2

```
1 int i = 1;
2 v[i] = i++;
```

Um diese Problematik zu umgehen, sollte beim Programmieren darauf geachtet werden, in einer Anweisung eine Variable nur ein einziges mal zu verwenden. Der Interpreter hält sich dabei an die mathematische Konvention von links nach rechts aufzulösen.

3.4.5 Behandlung von Exceptions

Etwas abweichend von der normalen Behandlung von Knoten ist der Umgang mit Exceptions. Tritt eine Exception auf, so wird diese angelegt, indem der entsprechende Konstruktor aufgerufen und ausgeführt wird. Das resultierende Objekt wird wie gewohnt in den Speicher geschrieben, das dazugehörige Symbol aber in keiner Symboltabelle abgelegt. Stattdessen wird ein Pointer auf dieses Objekt erstellt und separat von allen Symboltabellen als statisches Objekt in der *FunctionCall*-Klasse gespeichert.

Eine Variable repräsentiert den Status der *Exception*, anhand der bestimmt werden kann, ob sie bereits komplett abgefangen wurde oder ob der Interpreter gerade dabei ist, die *Exception* zu behandeln. Dies ist notwendig, da nach dem Auftreten einer *Exception* alle folgenden Knoten wie gewohnt aufgerufen werden, aufgrund des Status der aufgetretenen *Exception* aber ignoriert werden, bis der Interpreter auf einen *Catch*-Block stößt.

Ist der gefundene Catch-Block für die geworfene Exception zuständig (was anhand einer Überprüfung des Typs der Exception festgestellt wird (siehe A.9.7)), so wird die Statusvariable

der Exception auf "wird behandelt" gesetzt, so daß alle folgenden Anweisungen im Catch-Block nicht auch übersprungen werden. Wurde die Exception erfolgreich behandelt, wird die Exception wieder aus dem Speicher entfernt und alle globalen Variablen zurückgesetzt.

Unabhängig davon, ob eine Exception aufgetreten ist oder nicht, wird anschließend an die Ausführung des Try-Catch-Blocks der Finally-Block aufgerufen, sollte ein solcher vorhanden sein. Dies ist nur in Java der Fall.

3.4.6 Unterstützung von *Threads*

In C und C++ existieren keine Standardfunktionen zur Behandlung von Threads, hier müssen Bibliotheken verwendet werden, die sich für jede verwendete Plattform unterscheiden. Da der Interpreter nicht in der Lage ist, Drittbibliotheken zu interpretieren oder über seine Schnittstellen auf diese zuzugreifen, ist die Verwendung von Threads innerhalb des Interpreters nicht möglich.

In Java sind die entsprechenden Klassen zwar in den Standardbibliotheken vorhanden, da aber keine Schnittstellen von Ada zu Java existieren, ist eine Verwendung von Threads auch bei aus Java generieter IML nicht möglich.

3.5 Verwendung des Interpreters

In den folgenden Kapiteln wird die Installation und Verwendung des Interpreters beschrieben. Wie die zu interpretierenden IML-Dateien erstellt werden, ist dem Kapitel 3.5.2 zu entnehmen.

3.5.1 Installation des Interpreters

Zur Installation des Interpreters sind verschiedene Schritte notwendig. Neben einer funktionierenden Installation von Bauhaus wird auch die Ausführung bestimmter Hilfsprogramme vorausgesetzt, die Teile des Sourcecodes des Interpreters generieren.

Als Teil des Bauhaus-Projektes liegen die Quelldateien des Interpreters im `tools`-Verzeichnis des `projects`-Ordners. Der relative Pfad zu den Sourcen lautet `projects/tools/imlvm/src/`, wobei die zur Generierung notwendigen Projektdateien in `projects/tools/imlvm/` liegen.

Der Interpreter wird zusammen mit Bauhaus kompiliert, wenn dessen Build-Prozess durch den Aufruf von `make` gestartet wird. Das `imlvm`-Binary liegt nach dem erfolgreichen Kompilieren im `bauhaus-tools/bin/`-Verzeichnis des Bauhaus-Projektes.

3.5.1.1 Benötigte Bibliotheken

Welche Bibliotheken für das Kompilieren des Interpreters benötigt werden, ist in Tabelle 3.2 aufgeführt. Kompatibilität mit anderen Versionen als den angegeben ist nicht gewährleistet aber möglich.

3.5.1.2 Hilfsprogramme

Die hier aufgeführten Programme erstellen Teile des Interpreter-Quellcodes und ermöglichen es, den Interpreter zu konfigurieren und letztendlich zu kompilieren.

Tabelle 3.2: Versionen der verwendeten Bibliotheken

Name	Version
Bauhaus	22050[13]
gnat-pro	5.04a1
gtk+	2.8.18
GtkAda	2.8.1
xmlada	2.1
ASIS-for-GNAT	5.04a1
AUnit	1.05
g++	4.1.2
gcc	3.4.6

`create_scanf_files.c` Dieses Programm dient zur Generierung der ADA-Spezifikation und der -Quelldatei für das Aufrufen der `scanf`-Funktionen über das Interface. Als einzigen Parameter akzeptiert das Programm einen Integer, der bestimmt wieviele zusätzliche Parameter die Funktionen maximal unterstützen sollen. Aufgerufen werden muss das Programm innerhalb des Quellcodeverzeichnisses des Interpreters[14].

```
./create_scanf_files <NUM>
```

Das Programm bindet die entsprechenden C-Funktionen ein und generiert eine Auswahl der Funktionen anhand der vom Interpreter angegebenen Parameterzahl des Aufrufs. Die erstellten Dateien heißen `scanfs.ads` und `scanfs.adb` und werden benötigt um den Interpreter zu kompilieren.

`create_sscanf_files.c` Dieses Programm erstellt die Sourcen, die benötigt werden, um vom Interpreter aus auf die `sscanf`-Funktion von C zuzugreifen. Es verhält sich analog zu dem Programm `create_scanf_files`, das im vorherigen Kapitel beschrieben ist. Die resultierenden Dateien heißen `sscanfs.ads` und `sscanfs.adb`.

`create_fscanf_files.c` Dieses Programm erstellt die Sourcen, die benötigt werden, um vom Interpreter aus auf die `fscanf`-Funktion von C zuzugreifen. Es verhält sich analog zu dem Programm `create_scanf_files`, das in Kapitel 3.5.1.2 beschrieben ist. Die resultierenden Dateien heißen `fscanfs.ads` und `fscanfs.adb`.

3.5.2 Erstellung eines IML-Files

Im Folgenden wird die Erstellung von IML-Files aus Quellcode beschrieben. Die erstellten IML-Dateien lassen sich wie in Kapitel 3.5.4 beschrieben ausführen. Vorausgesetzt wird eine valide Bauhaus-Installation und daß im Pfad (der Umgebungsvariable `PATH`) das Verzeichnis der einzelnen Bauhaus-Tools eingetragen ist.

C-Sourcedateien Zur Erstellung einer IML-Datei aus C-Quelldateien werden die Programme `cafeCC` und `cafe` benötigt. Die Eingabeparameter ähneln stark denen vom `gcc` und die Minimalkonfiguration sieht wie folgt aus:

```
cafeCC SOURCEFILE.c -o OUTPUT.iml
```

[14] `projects/tools/imlvm/src/`

C++-Sourcedateien Zur Erstellung einer IML-Datei aus C++-Quelldateien werden die Programme `cafeCC` und `cafe++` benötigt. Die Eingabeparameter ähneln stark denen vom `g++` und die Minimalkonfiguration sieht wie folgt aus:

```
cafeCC SOURCEFILE.cc -o OUTPUT.iml
```

3.5.3 Konfiguration des Interpreters

Es existieren keine Konfigurationsdateien für den Interpreter. Sämtliche Einstellungen müssen über die Kommandozeile erfolgen. Welche Parameter existieren, ist dem folgenden Kapitel zu entnehmen.

3.5.4 Aufruf des Interpreters

Für die Verwendung des Interpreters wird eine gültige Bauhaus-Lizenz benötigt. Ist diese nicht vorhanden, ist ein Aufruf des Interpreters nicht möglich.

Der Interpreter läßt sich über die Kommandozeile wie folgt aufrufen:

`imlvm IMLFILE [-mem <number>] [-verbose|-quiet] [-lines] [-param <string>]`

Folgende Parameter sind möglich, relevant ist dabei nur der erste, der angibt welche IML-Datei interpretiert werden soll.

`IMLFILE` Dies ist der Name des zu interpretierenden IML-Graphen und der einzige Pflichtparameter. Handelt es sich bei dem angegebenen IML-Graphen um die Repräsentation einer binären Datei, beziehungsweise um den Quellcode eines ausführbaren Programms, so wird die Interpretation sofort gestartet. Handelt es sich um eine Bibliothek oder um eine Java-Klasse, so werden alle gültigen Einstiegsmethoden vom Interpreter aufgelistet und der Benutzer kann eine von diesen auswählen. Mit dieser Methode beginnt dann die Interpretation des Graphen.

`-param <string>` Mittels dieses Parameters lassen sich Parameter angeben, die der Hauptfunktion des zu emulierenden Programms übergeben werden sollen. Dieser String sollte bei mehreren Parametern mit Anführungszeichen umschlossen und einzelne Parameter mit Leerzeichen getrennt werden.

Soll einer der Parameter ein Leerzeichen enthalten, so sollte dieser mit maskierten Anführungszeichen umgeben werden (`p1 \"p2 p2\" p3`). Parameter die ein Anführungszeichen enthalten sollen, denen muss ein maskierter Schrägstrich vorgesetzt werden (`p1 \\\"p2`).

Handelt es sich bei dem interpretierten Quellcode um C oder C++, so ist der erste Parameter, unabhängig von den restlichen, immer gleich dem Namen des IML-Files und vorgestelltem `./` und die restlichen werden entsprechend angehängt.

`-mem <number>` Dieser Wert gibt an, wieviel Speicher in Byte der Interpreter für den auszuführenden Graphen anlegen soll. Der Standardwert hierfür entspricht 10.485.760, also zehn Megabyte. Sollte dem Interpreter zur Laufzeit der Speicher ausgehen, so wird die Ausführung abgebrochen und eine entsprechende Fehlermeldung ausgegeben (siehe Kapitel 3.5.5.1).

`-quiet` Diese Option unterdrückt sämtliche Ausgaben des Interpreters die nicht im IML-Graphen selbst vorkommen. Ausgenommen sind also sämtliche Funktionen die über

I/O-Bibliotheken realisiert sind. Auch sämtliche Fehlermeldungen des Interpreters werden hiermit unterdrückt.

`-verbose` Wird diese Option angegeben, werden sämtliche Ausgaben des Interpreters aktiviert. Dies beinhaltet sowohl Debugging- als auch reine Laufzeitinformationen, die nützlich sind um sowohl Fehler in der IML als auch dem Interpreter selbst zu finden.

Diese Option überschreibt die `-quiet`-Option.

`-lines` Mit dieser Option werden vor die Ausgaben eines jeden Execute-Knotens der IML die Quelldatei und die Zeile des entsprechenden Knotens angegeben. Standardmäßig wird diese Ausgabe unterdrückt.

3.5.5 Fehlermeldungen des Interpreters

Stößt der Interpreter auf einen Fehler, so gibt dieser eine entsprechende Fehlermeldung aus oder erzeugt eine Exception. Sowohl Fehlermeldungen als auch die vorkommenden Exceptions werden im Folgenden näher erläutert.

Tritt eine Exception auf, so wird ebenfalls ein Stack-Trace der IML ausgegeben, der jeweils die übergeordneten IML-Knoten und die dazugehörige Datei und Zeile des Quellcodes ausgibt.

3.5.5.1 Exceptions des Interpreters

Sollten während der Ausführung der IML unerwartete Probleme auftreten, so wird eine Exception mit detaillierteren Fehlerbeschreibungen geworfen. Dies kann sowohl bei Fehlern innerhalb der IML als auch im Interpreter auftreten. Im Folgenden sind die Exceptions aufgelistet die auftreten können und nicht vom Interpreter abgefangen werden. Sie beenden daher die Ausführung des IML-Graphen.

Not_Implemented_Exception Diese Exception tritt auf, wenn der Interpreter eine Operation ausführen möchte, die noch nicht implementiert wurde. Hierbei handelt es sich dann um eine Funktion die an sich vorhanden sein müsste aber aus Zeitgründen noch nicht behandelt wurde. Gerade bei der Fülle an möglichen Kombinationen von Operatoren und Datentypen kann es vorkommen, daß auch einzelne mögliche Konstellationen übersehen wurden und so eine Exception geworfen wird.

Eine weitere Situation in der solch eine Exception auftreten kann ist, wenn eine Funktion wie `scanf` mit mehr Parametern aufgerufen wird, als momentan unterstützt wird. Die Anzahl an unterstützten Parametern wird zur Kompilierzeit des Interpreters festgelegt (siehe Kapitel 3.5.1.2).

Out_Of_Memory_Exception Reicht der vorhandene Speicher nicht aus, um die vom Interpreter benötigten Objekte anzulegen, so wird diese Exception geworfen und die Interpretation des IML-Graphen abgebrochen. Entweder handelt es sich hierbei um ein Speicherleck oder die Erhöhung des zur Verfügung stehenden Speichers beseitigt das Problem (siehe Kapitel 3.5.4).

Invalid_Pointer_Exception Wird versucht einen reservierten Speicherbereich zu befreien der nicht oder nur teilweise belegt ist, so wird diese Exception geworfen. Auch wenn ein Speicherbereich angesprochen wird, der außerhalb der definierten Grenzen liegt, wird diese Exception geworfen.

Unknown_Function_Called Wird eine externe Funktion aufgerufen, die dem Interpreter unbekannt ist, so wird diese Exception geworfen. Dies kann nur auftreten, wenn externe Funktionen der repräsentierten Sprache theoretisch über ein Interface angesprochen werden können. Existiert kein solches Interface, wird die Ausführung schon vorher abgebrochen.

No_External_Function_Support Falls der IML-Graph eine Sprache repräsentiert, deren externe Funktionen von dem Interpreter nicht unterstützt werden und eine solche aufgerufen wird, wird diese Exception geworfen.

Reference_To_Unknown_Object Diese Exception wird vom Garbage Collector geworfen, wenn die Anzahl an verweisenden *Pointern* für einen Speicherbereich verändert werden soll, der nicht von einer Klasseninstanz belegt ist.

Null_Pointer_Exception Versucht der Interpreter auf einen Wert mit der Speicheradresse Null zuzugreifen, wird diese Exception geworfen. Dies ist meist der Fall, wenn ein *Pointer* dereferenziert werden soll, der noch nicht initialisiert wurde.

KAPITEL 4

Ergebnisse der Arbeit

In diesem Kapitel werden die Ergebnisse der Arbeit beschrieben. Zuerst wird auf die Korrektheit des Interpreters eingangen und diese anhand von Beispielen belegt. Es werden die Ergebnisse der Interpretation eines IML-Graphen mit denen der Ausführung des jeweiligen Binaries, dem kompilierten Quellcode, verglichen.

Im anschließenden Kapitel wird die Performanz des Interpreters mit dem des Binaries verglichen und die Ergebnisse erläutert.

Zu welchen Ergebnissen diese Arbeit geführt hat, wird daraufhin in Kapitel 4.3 zusammengefasst.

In Kapitel 4.4 werden alle noch offenen Punkte aufgeführt. Hierzu gehören sowohl erweiternde Funktionen, die dem Interpreter noch hinzugefügt werden können, sowie Anpassungen der IML, die durchgeführt werden müssen, um mehr Eigenschaften der jeweiligen Sprachen zu unterstützen. Auch Möglichkeiten für die Integration von Laufzeitanalysen werden dort beschrieben.

4.1 Korrektheit

In diesem Abschnitt werden die Ergebnisse der Korrektheitsprüfung des Interpreters aufgeführt. Diese Tests umfassen einzelne kleine Programme, die bestimmte Funktionen der jeweiligen Sprache verwenden und so die korrekte Interpretation prüfen. Verglichen wird das Ergebnis des Interpreters mit denen des kompilierten Quellcodes. Die Performanz des Interpreters wird in diesem Kapitel nicht berücksichtigt, auf sie wird in Kapitel 4.2 eingangen.

Da nur die IML von C-Programmen vom Interpreter nahezu vollständig unterstützt wird, werden hier die meisten Eigenschaften der Sprache getestet werden. Die Unterstützung von C++ beschränkt sich auf die neuen Datentypen und fällt entsprechend kürzer aus. Auf das Testen von IML die aus Java und Ada generiert wurde, wurde aufgrund von mangelnder Unterstützung verzichtet.

Bei dem Vergleich der Ausgaben der Programme gilt zu berücksichtigen das Ada-Programme, wie der Interpreter eines ist, an jede Ausgabe die mindestens ein Zeichen beinhaltet, am Ende einen Zeilenumbruch anhängt. Daher wird sich die Ausgabe des interpretierten Programmes immer um mindestens ein Zeichen unterscheiden. Aus technischen Gründen ist dieser Zeilenumbruch in den in diesem Dokument dargestellten Ausgaben nicht enthalten.

4.1.1 C-Programme

Die einzelnen Tests der Eingenschaften von C sind in kleine Programme aufgeteilt, die jeweils einige der Funktionalitäten der Sprache benutzen.

Obwohl es sich bei diesen Programmen um reinen C-Code handelt, müssen sie teilweise mit dem `g++` übersetzt werden. Die Endung `.cc` ist notwendig, damit das `cafeCC`-Tool diese als C++-Quelldateien identifiziert. Ansonsten würden die C-Bibliotheken verwendet werden, die Makros für die Aufrufe aus dem `ctype`-Header verwenden und daher nicht interpretiert werden können.

operators.c Dieses Programm führt sechs unterschiedliche Berechnungen mit dem Parameter an, der dem Programm übergeben wird. Zuerst wird der Parameter in eine Integerzahl mittels der externen Funktion `atoi` umgewandelt und anschließend geteilt, multipliziert, potenziert, moduliert und auf Basis von Bits verundet und verodert. Die Ausgabe erfolgt mittels der `printf`-Funktion des `stdio.h`-Headers.

Listing 4.1: `operators.c` Binary

```
1 var /= 2 = 5
2 var *= 2 = 10
3 var ** 2 = 100
4 var %= 17 = 15
5 var &= 11 = 11
6 var |= 431 = 431
```

Listing 4.2: `operators.c` Interpreter

```
1 var /= 2 = 5
2 var *= 2 = 10
3 var ** 2 = 100
4 var %= 17 = 15
5 var &= 11 = 11
6 var |= 431 = 431
```

fak.c Dieses Programm berechnet rekursiv die Fakultät von Zehn. Verwendet werden die Datentypen `int` und `long`. Als einzige externe Funktion wird `printf` verwendet.

Listing 4.3: `fak.c` Binary

```
1 Fak(10) = 3628800
```

Listing 4.4: `fak.c` Interpreter

```
1 Fak(10) = 3628800
```

pointer.c Dieses Programm erstellt ein Integer-Array mit sechs Elementen und gibt diesen mit jeweils unterschiedlicher Interpretation aus. Es wird ein `int`-, ein `short`- und ein `long`-Pointer erzeugt, die durch das Array iterieren und den jeweiligen Wert, auf den sie verweisen, ausgeben.

Listing 4.5: `pointer.c` Binary

```
1 Array as int:
2 17460479 34920958 52381437 69841916
    87302395 104762874
3 Array as short:
4 27903 266 -9730 532 18173 799 -19460
    1065 8443 1332 -29190 1598
5 Array as long:
6 149984372572450047 299968745162360573
    449953117752271099
```

Listing 4.6: `pointer.c` Interpreter

```
1 Array as int:
2 17460479 34920958 52381437 69841916
    87302395 104762874
3 Array as short:
4 27903 266 -9730 532 18173 799 -19460
    1065 8443 1332 -29190 1598
5 Array as long:
6 149984372572450047 299968745162360573
    449953117752271099
```

count.cc Dieses Programm erwartet einen Parameter, den Namen einer Textdatei, die im gleichen Verzeichnis liegt. Es liest die Datei Zeichen für Zeichen ein und prüft welcher Art diese sind und gibt am Ende das prozentuale Vorkommen aller Zeichentypen aus.

Als Parameter wurde der Quelltext des Programms selbst (`count.cc`) angegeben.

Dieses Programm verwendet sowohl Unterfunktionen, Programmparameter, Arrays, Strings, Schleifen und die externen Funktionen `memset`, `printf`, `fread`, `isalnum` (und sämtliche Äquivalente aus dem Header `ctype.h`) und `fclose`.

Listing 4.7: `count.cc` Binary

```
Total number of characters: 1443
  809          alphanumeric =  56.06%
  760           alphabetic =  52.67%
   67     control character =   4.64%
   49               digit =   3.40%
 1122   graphical character =  77.75%
  739   lowercase character =  51.21%
 1376    printing character =  95.36%
  313           punctuation =  21.69%
  321       space character =  22.25%
   21   uppercase character =   1.46%
  304 hexidecimal character =  21.07%
```

Listing 4.8: `count.cc` Interpreter

```
Total number of characters: 1443
  809          alphanumeric =  56.06%
  760           alphabetic =  52.67%
   67     control character =   4.64%
   49               digit =   3.40%
 1122   graphical character =  77.75%
  739   lowercase character =  51.21%
 1376    printing character =  95.36%
  313           punctuation =  21.69%
  321       space character =  22.25%
   21   uppercase character =   1.46%
  304 hexidecimal character =  21.07%
```

4.1.2 C++-Programme

Da *Templates* und somit die *Standard Template Library* von C++ nicht unterstützt werden, werden im Folgenden nur die C++-Datentypen geprüft.

classes.cc Dieser Quellcode beinhaltet die Definition einer Klasse mit einem Konstruktor und einem Destruktor, die jeweils ausgeben, wieviele Instanzen der Klasse existieren. Dies wird anhand einer statischen Variable innerhalb der Klasse gezählt. Eine globale Variable bestimmt, wieviele Instanzen angelegt werden sollen, in diesem Fall Zehn.

Erstellt werden die Instanzen mit dem `new`-Operator und der resultierende Pointer wird in einem Array gespeichert. Gelöscht werden die Instanzen entsprechend explizit mit dem `delete`-Operator.

Listing 4.9: `classes.c` Binary

```
Constructor called, 1 of 10 available!
Constructor called, 2 of 10 available!
Constructor called, 3 of 10 available!
Constructor called, 4 of 10 available!
Constructor called, 5 of 10 available!
Constructor called, 6 of 10 available!
Constructor called, 7 of 10 available!
Constructor called, 8 of 10 available!
Constructor called, 9 of 10 available!
Constructor called, 10 of 10 available!
Destructor called, 9 of 10 left!
Destructor called, 8 of 10 left!
Destructor called, 7 of 10 left!
Destructor called, 6 of 10 left!
Destructor called, 5 of 10 left!
Destructor called, 4 of 10 left!
Destructor called, 3 of 10 left!
Destructor called, 2 of 10 left!
Destructor called, 1 of 10 left!
Destructor called, 0 of 10 left!
```

Listing 4.10: `classes.c` Interpreter

```
Constructor called, 1 of 10 available!
Constructor called, 2 of 10 available!
Constructor called, 3 of 10 available!
Constructor called, 4 of 10 available!
Constructor called, 5 of 10 available!
Constructor called, 6 of 10 available!
Constructor called, 7 of 10 available!
Constructor called, 8 of 10 available!
Constructor called, 9 of 10 available!
Constructor called, 10 of 10 available!
Destructor called, 9 of 10 left!
Destructor called, 8 of 10 left!
Destructor called, 7 of 10 left!
Destructor called, 6 of 10 left!
Destructor called, 5 of 10 left!
Destructor called, 4 of 10 left!
Destructor called, 3 of 10 left!
Destructor called, 2 of 10 left!
Destructor called, 1 of 10 left!
Destructor called, 0 of 10 left!
```

4.2 Performanz

In diesem Abschnitt wird die Performanz des Interpreters ermittelt[1]. Es werden unterschiedliche Operationen eine Million mal durchgeführt und die erforderliche Zeit gemessen.

[1]Der für den Test verwendete Rechner besitzt einen Intel Core 2 Duo E6600-Prozessor und verfügt über zwei Gigabyte Arbeitsspeicher.

Die unterschiedlichen Tests werden im Folgenden beschrieben und anschließend die Ergebnisse erläutert und graphisch dargestellt.

Die Iterationen der einzelnen Tests werden mittels einer `for`-Schleife realisiert, die Teil des Programms ist. Diese wird auch interpretiert, weswegen deren Aufwand ebenfalls in das Testergebnis einfließt. Teil dieser Schleifen sind eine Initialisierung und je Iteration eine *Postfix*-Operation und eine *Equal*-Abfrage.

Dateiausgabe In diesem Testdurchlauf wird der String "Test" eine Million mal mittels der Funktion `fprintf` in eine Datei geschrieben. Getestet wird mit diesem Durchlauf der Aufwand für den Aufruf von externen Funktionen über Schnittstellen. Die Ausführung der externen Funktion selbst, das Schreiben in eine Datei, ist vom Aufwand her zu vernachlässigen.

Arithmetik In diesem Testdurchlauf wird eine einfache Addition eine Million mal durchgeführt. Getestet wird mit diesem Durchlauf der Aufwand für einfache Operationen.

Konstruktor Eine Million Instanzen einer Klasse werden mittels des `new`-Operators erzeugt und der Speicher nicht wieder freigegeben. Getestet wird der Aufruf von Konstruktoren und das Belegen von Speicher.

Destruktor In diesem Durchlauf wird gegenüber dem vorherigen die erstellte Instanz der Klasse wieder zerstört und der dazugehörige Speicherbereich wieder freigegeben. Innerhalb des Dekonstruktors erfolgt keine Ausführung. Getestet wird mit diesem Durchlauf das Freigeben von Speicherbereichen.

Rekursion Dieser Testdurchlauf berechnet eine Million mal rekursiv die Fakultät von Zehn. Dies entspricht in etwa 25 Millionen Operationen.

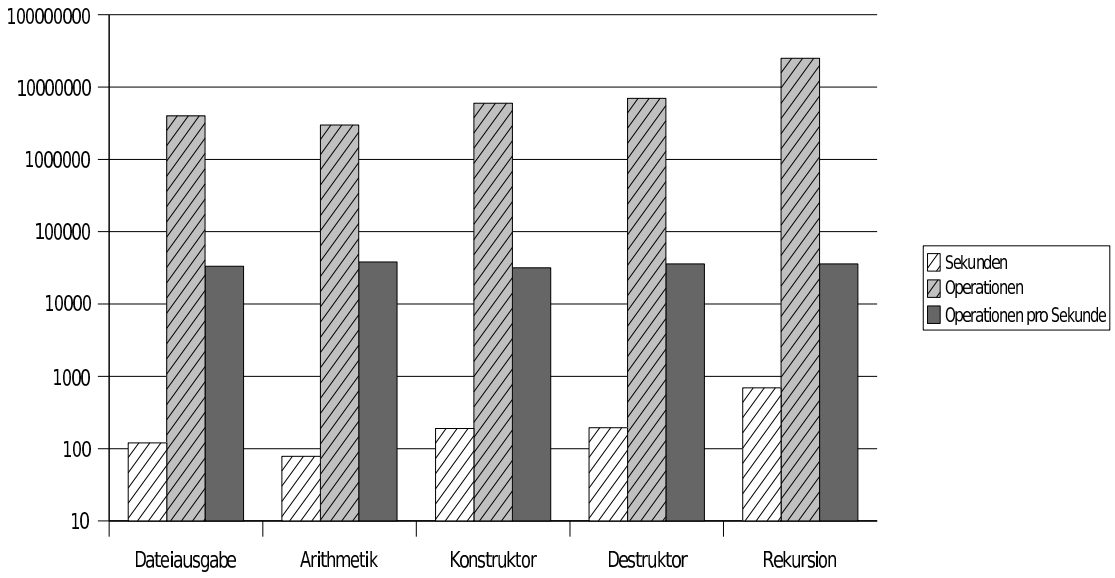

Abbildung 4.1: Dieses Diagramm zeigt die Ergebnisse der Performanztests.

Wie der 4.1 entnehmen kann, liegt die durchschnittliche Anzahl an Operationen pro Sekunde bei 35.000. Dabei ist die Art der durchgeführten Operation irrelevant. Im Vergleich zu dem kompilierten Programm ist der Interpreter um den Faktor 30.000 langsamer. Wie die Performanz verbessert werden kann, ist dem Kapitel 4.4.7 zu entnehmen.

4.3 Ergebnisse

Wie diese Arbeit zeigt, lassen sich IML-Graphen interpretieren und ausführen. Die im Graphen enthaltenden Informationen sind im Grunde bis auf wenige Ausnahmen ausreichend, um eine Ausführung zu ermöglichen.

Die Schnittstellen zu den von der IML unterstützten Sprachen stellen das größte Hindernis bei der Interpretation dar. Denn zu den meisten der von der IML unterstützten Sprachen existieren keine Schnittstellen von Ada aus, so daß auf diese Funktionen nicht direkt zugegriffen werden kann.

Die Graphen für die einzelnen Sprachen unterscheiden sich teilweise grundlegend, da diese sich im Aufbau stark unterscheiden. So gibt es in Ada zum Beispiel ausführbare Units ohne Main-Funktion und in Java kann jede eingebundene Klasse eine Main-Methode besitzen, während in C und C++ nur eine erlaubt ist. Dies erschwert die Interpretation, da ein gewisser Aufbau des Graphen an einigen Stellen vorausgesetzt wird, da die vorliegenden Informationen nicht immer ausreichen. Mit einer Abfrage über die interpretierte Sprache und einer enstprechenden Behandlung kann dies zwar umgangen werden, was aber gegen den eigentlichen Gedanken, unabhängig von der repräsentierten Sprache zu sein, verstößt.

Möglich ist mit dem Interpreter die Ausführung von IML die aus C-Quellcode generiert wurde. Die Interpretation von C++-Quellcode ist nur eingeschränkt möglich, da die *Standard Template Library* nicht angesprochen werden kann. Die meisten der ungefährt 150 Standardfunktionen von C können vom Interpreter aufgerufen werden, auch wenn deren Funktionalität teilweise eingeschränkt ist.

Änderungen an der IML oder dessen Generierung wurden während der Erstellung des Interpreters keine vorgenommen. Der Interpreter selbst wurde abgesehen von den kleinen Hilfsprogramm zur Quellcodegenerierung komplett in Ada geschrieben und ist ungefähr 56.000 Zeilen lang. Es wurden, abgesehen von der *Standard Library* von C, keine weiteren Bibliotheken verwendet, die nicht Teil von Bauhaus sind.

Die Einarbeitung in das Bauhaus-Projekt und die IML dauerte länger als anfangs angenommen, da keinerlei Vorkenntnisse vorlagen. Auch die verwendete Programmiersprache war vorher nicht bekannt, weswegen auch hier zahlreiche Probleme auftraten. Daher musste die Zeitplanung mehrfach angepasst werden und auf die Implementierung einiger Optionen verzichtet werden. Die erste lauffähige Version wurde nach einem Monat bei einem Zwischenvortrag vorgestellt. Diese verwendete das abstrakte Speichermodell und war in seiner Funktionalität stark eingeschränkt. Nach diesem Vortrag und Rücksprache mit den Entwicklern des Bauhaus-Projektes war eine Neuimplementierung des Speichermodells notwendig. Insgesamt waren die sechs Monate Zeit, die für diese Arbeit benötigt wurden, ausreichend, um die gegebenen Anforderungen zu erfüllen.

4.4 Offene Punkte

Im Folgenden wird auf alle Punkte eingegangen die bei der Fertigstellung dieser Diplomarbeit noch offen sind. Dazu gehören externe Funktionen aus Standardbibliotheken die aus Zeitgründen nicht eingebunden wurden und auch Funktionen die aus technischen Gründen nicht implementiert werden konnten. Zu den Gründen gehören sowohl fehlende oder unvollständige Schnittstellen als auch inkorrekte oder unvollständige Repräsentationen des Quelltextes seitens der IML.

4.4.1 Anpassung der IML

Die Repräsentation des Quelltextes anhand der IML beinhaltet fehlerhafte Abbildungen und einzelne Knoten, die zur Interpretation benötigt werden, fehlen an einigen Stellen. Die Erweiterungen beziehungsweise Veränderungen, die an den Programmen zur Generierung der IML-Graphen vorgenommen werden müssen, um eine fehlerfreie Interpretation zu gewährleisten, sind im Folgenden aufgeführt.

4.4.1.1 Konvertierungsknoten

An vielen Stellen innerhalb der IML fehlen Knoten zur Repräsentierung von Konvertierungen von Datenwerten. In den meisten Fällen sind diese innerhalb der IML dargestellt, indem ein Knoten einen anderen Typ hat als sein Kindknoten. Es wird also kein *Implicit_Conversion*-Knoten verwendet, obwohl dies ein valider IML-Knoten ist.

Zwar kann der Interpreter anhand der Unterschiede der Typen der jeweiligen Knoten die Konvertierung durchführen, bei Pointern treten dann aber Probleme auf. Denn diese werden in der IML gar nicht konvertiert, wenn dies innerhalb eines Ausdrucks geschieht.

Auch arithmetische Konvertierungen (siehe Kapitel 3.4.3) werden in der IML nicht über spezielle Knoten abgebildet. Diese müssten vom Interpreter speziell in den jeweiligen arithmetischen Knoten, falls nötig, durchgeführt werden.

4.4.1.2 Indexed_Dereference

Die Repräsentation dieses Knotens in der IML ist fehlerhaft. Das Codebeispiel aus Listing 4.11 würde in der IML mit einem *Indexed_Dereference*-Knoten repräsentiert werden, bei dem der Index die `1` ist und die Basisadresse die des `doubles` `d`. Die Konvertierung des *Pointers* von `d` zum Typ `char*` wird bei der Generierung der IML komplett ignoriert, was bewirkt daß bei der Dereferenzierung nicht `sizeof(char)` sondern `sizeof(double)` Bytes zur Basisadresse hinzugerechnet werden. Dies führt zu falschen Ergebnissen bei der Dereferenzierung, sollte die Größe der beiden Typen sich unterscheiden.

Dieser Fehler ist teilweise zurückzuführen auf das Fehlen von Konvertierungsknoten, wie im vorherigen Abschnitt beschrieben.

Listing 4.11: Fehlerhafte IML-Darstellung bei Pointerkonvertierung

```
1  double d = 1234.1234;
2  char c;
3
4  c = (*(((char*)(&d))+1));
```

4.4.1.3 Initialisierung eines `char`-Arrays

Es gibt mehrere Methoden innerhalb von C mit `char`-Arrays zu arbeiten. Eine Möglichkeit ist es, sie mit einem `char*` zu referenzieren, eine andere sie direkt als Array zu behandeln.

Listing 4.12: `char`-Array-Initialisierng

```
1  char *t = "test";
2  char tt[5] = "test";
```

Die in Listing 4.12 in der ersten Zeile aufgeführte Initialisierung wird in der IML korrekt abgebildet. Es wird ein String erstellt und dieser zu einem *Pointer* konvertiert und in `t` gespeichert.

Die Initialisierung in der zweiten Zeile ist dagegen fehlerhaft in der IML repräsentiert. Wie in Abbildung 4.2 zu sehen ist, handelt es sich bei dem Zielknoten um einen *Dereference*-Knoten. Dessen *Its_Type*-Kante verweist auf den gleichen Typ-Knoten, wie das Objekt das dereferenziert werden soll. Dies stellt ein Problem dar, da laut dem Graphen der Dereferenzierungsknoten keine Funktion hat, außer den eigentlichen Wert weiterzureichen. Dies widerspricht allerdings der Definition des Knotens, laut der in dem Fall das erste Element des Arrays zurückgeben werden müsste. Wird das Beispiel weiterverfolgt, bedeutet dies, daß die Quelle der Initialisierung, in diesem Fall ein *Pointer*, vom Interpreter einem *char* zugewiesen werden würde. Anstelle des eigentlichen Strings würde am Ende im *char*-Arrays dessen Adresse stehen (nachdem diese in einen *char* gecastet wurde).

Der Interpreter fängt diesen Spezialfall ab und kopiert den String auf den der *Source-Pointer* verweist in das Array, das als *Target* dereferenziert wird.

4.4.2 C

Aufgrund fehlender Kompatibilität der Ada-Schnittstellen zu C in Hinsicht auf variable Parameterzahlen und -typen müssen die entsprechenden Standardfunktionen anderwertig eingebunden werden, so dies denn möglich ist. Da der Interpreter in der Lage ist, die Funktionalität von einem vollwertigem `printf`, `fprintf` und `sprintf` durch die Zerlegung des Aufrufs zu erreichen, müssen diese Methoden nicht weiter berücksichtigt werden. Das gleiche gilt für die Methoden `scanf`, `sscanf` und `fscanf`, die bis zu einer bestimmten Parameterzahl angesprochen werden können (siehe Kapitel 3.4.2.1.5).

Die Unterstützung von komplexen Zahlen und `wide chars` muss ebenfalls eingebunden werden, wenn eine Bauhaus-Version zur Verfügung steht, in der `cafeCC` diese Typen unterstützt.

Auch die Unterstützung von variablen Parameterlisten muss zu einem späteren Zeitpunkt implementiert werden, wenn die IML-Generatoren in der Lage sind, diese korrekt innerhalb der IML abzubilden anstatt diese gänzlich zu ignorieren (siehe Kapitel 3.2.1.1).

4.4.3 C++

Da momentan keine Schnittstellen von Ada zu C++ existieren, die Templates unterstützen, kann die Einbindung in den Interpreter erst zu einem späteren Zeitpunkt erfolgen, wenn diese existieren.

4.4.4 Java

Die momentan fehlenden Schnittstellen von Ada zu Java ermöglichen keine Einbindung der Java-Bibliotheken. Daher ist die Interpretation aus Java-Quellcode generierter IML sinnlos, denn ohne diese Bibliotheken verfügt das Programm über keinerlei Ausgabe und hat auch keine Möglichkeit auf irgend eine Weise mit anderen Prozessen zu kommunizieren.

Desweiteren muss entweder das Problem bei der Generierung der IML mittels `jafe` beseitigt werden, bei dem sämtliche Objektreferenzen eine Größe von Null haben oder aber der Interpreter müsste dieses Problem erkennen und in dem Fall eine Größe von vier Byte annehmen.

4.4.5 Ada

Um eine Unterstützung von Ada zu ermöglichen, die nur den Aufruf von Funktionen und grundlegende Arithmetik beinhaltet, muss die Initialisierung des Interpreters generischer gestaltet werden. Die aktuelle Implementierung setzt einen bestimmten Aufbau der Funktionsknoten voraus, die bei Ada nicht gegeben ist.

Die Repräsentation von Datentypen innerhalb des Interpreters müsste ebenfalls generischer gestaltet werden, so daß auch mit Wertegrenzen behaftete Datentypen möglich sind.

Eine Einbindung zumindest ausgewählter Standardfunktionen ist zwingend notwendig, um auch sinnvolle Programme interpretieren zu können.

4.4.6 64-Bit

Der aktuelle Interpreter geht von einem 32-Bit-System aus und behandelt die internen Systemadressen entsprechend. Dies bedeutet, daß falls Speicher angefordert wird, der nicht ein Multiplikatives von Vier ist, die Anforderung entsprechend erhöht wird. Dies verhindert, daß Speicherbereiche zur Verfügung gestellt werden, die nicht für das unterliegende System üblich sind. Bei einem 64-Bit-System müssten entsprechend die Adressen auf ein Multiplikatives von Acht erhöht werden, um das Problem zu umgehen. In den meisten Fällen ist dies nur bei der Reservierung für Speicher von Arrays von Chars und bei Zeichenketten der Fall.

Die einzelnen Datentypen müssten ebenfalls entsprechend angepasst werden. Ein 128-Bit Integer wird momentan vom Interpreter nicht unterstützt. Eine Erweiterung der Funktionalität, damit nahezu beliebig große Datentypen unterstützt werden, würde dieses Problem beseitigen.

4.4.7 Optimierungsmöglichkeiten

Der Interpreter verwendet in seiner ersten Version kein Datenregister für Zwischenergebnisse von Berechnungen. Dies bedeutet daß alle Ergebnisse im emulierten Hauptspeicher abgelegt werden, was einen Mehraufwand darstellt. Eine Implementierung eines Akkumulatorregisters würde die Anzahl der Zugriffe auf den Speicher und die Symboltabellen erheblich verringern. Dies würde auch die Performanz des Interpreters verbessern.

Temporäre Symbole, die momentan mit einem negativen Index versehen sind, müssten dann im Akkumulator abgelegt werden. Das Hochzählen der Indizes müsste nicht mehr für jede Funktion einzeln geschehen, sondern global, da ansonsten einzelne Funktionen die im Akkumulator liegenden Symbole der überliegenden Funktionen überschreiben würde.

Bei der Auswahl, welche externe Funktion aufgerufen werden soll, wird momentan der Name der Funktion, ein String, mit den Namen der verfügbaren Funktionen verglichen, bis die korrekte gefunden wurde. Dies bedeutet, daß wenn eine externe Funktion aufgerufen wird, im schlimmsten Fall über 120 Stringvergleiche durchgeführt werden müssen. Eine andere Form der Identifikation der aufzurufenden Funktion existiert innerhalb der IML nicht. Eine Schachtelung der Vergleiche anhand bestimmter Merkmale wie dem Anfangsbuchstaben des Namen, dem Rückgabetypen oder der Parameterzahl würde die Ausführung beschleunigen. Auch das Setzen der gebräuchlicheren Funktionen in den oberen Teil der Abfragen, würde eine, wenn auch geringe, Verbesserung der Performanz bewirken.

4.4.8 Konfigurationsdateien

Momentan wird der Interpreter über die Kommandozeile konfiguriert und die Möglichkeiten sind dabei begrenzt. Einige Einstellungen, wie die Größe der einzelnen Datentypen, sind nur durch Änderungen des Quellcodes möglich.

Die Einführung von Konfigurationsdateien würde die Flexibilität des Interpreters erhöhen und kein erneutes Kompilieren für unterschiedliche Plattformen benötigen.

4.4.9 Integration von IML-Analysen

Der Interpreter selbst beinhaltet keine Analysen der IML, er führt sie momentan nur aus. Eine Integration von Schnittstellen für analytische Zwecke ist jedoch ohne großen Aufwand möglich. Zu den möglichen Analysen gehören folgende:

- Wie häufig wurde ein bestimmter Knoten aufgerufen?
- Welche Werte hat ein *O_Node* maximal und/oder minimal angenommen?
- Wie oft wurde ein *O_Node* ausgelesen oder beschrieben?
- Wie hoch war die maximale rekursive Tiefe bestimmter Funktionsaufrufe?
- Wie sieht der Speicher zu einem bestimmten Zeitpunkt aus?

Zwar ist es möglich diese oder ähnliche Analysen mit anderen Programmen direkt auf der IML durchzuführen, diese wären aber nicht so dynamisch (wenn zum Beispiel Eingaben des Benutzers über die Konsole verarbeitet werden müssen) und würden über keine Schnittstelle zur Standardbibliothek der verwendeten Sprache verfügen. Zwar könnten alle in der IML abgebildeten Anweisungen analysiert werden, aber die externen Funktionen und deren Auswirkungen wären davon ausgeschlossen.

Auch die Implementierung von Haltepunkten wäre anhand der Indizes der IML-Knoten möglich. Stößt der Interpreter auf solch einen als Haltepunkt markierten Knoten, wäre das Anhalten der Interpretation möglich. Die Ausgabe der aktuellen Symboltabelle und des dazugehörigen Speicherbereichs wäre ebenfalls möglich. So könnte auch definiert werden, daß bei jeder Ausführung eines bestimmten Knotens der komplette benutzte Speicherbereich in eine Datei geschrieben wird.

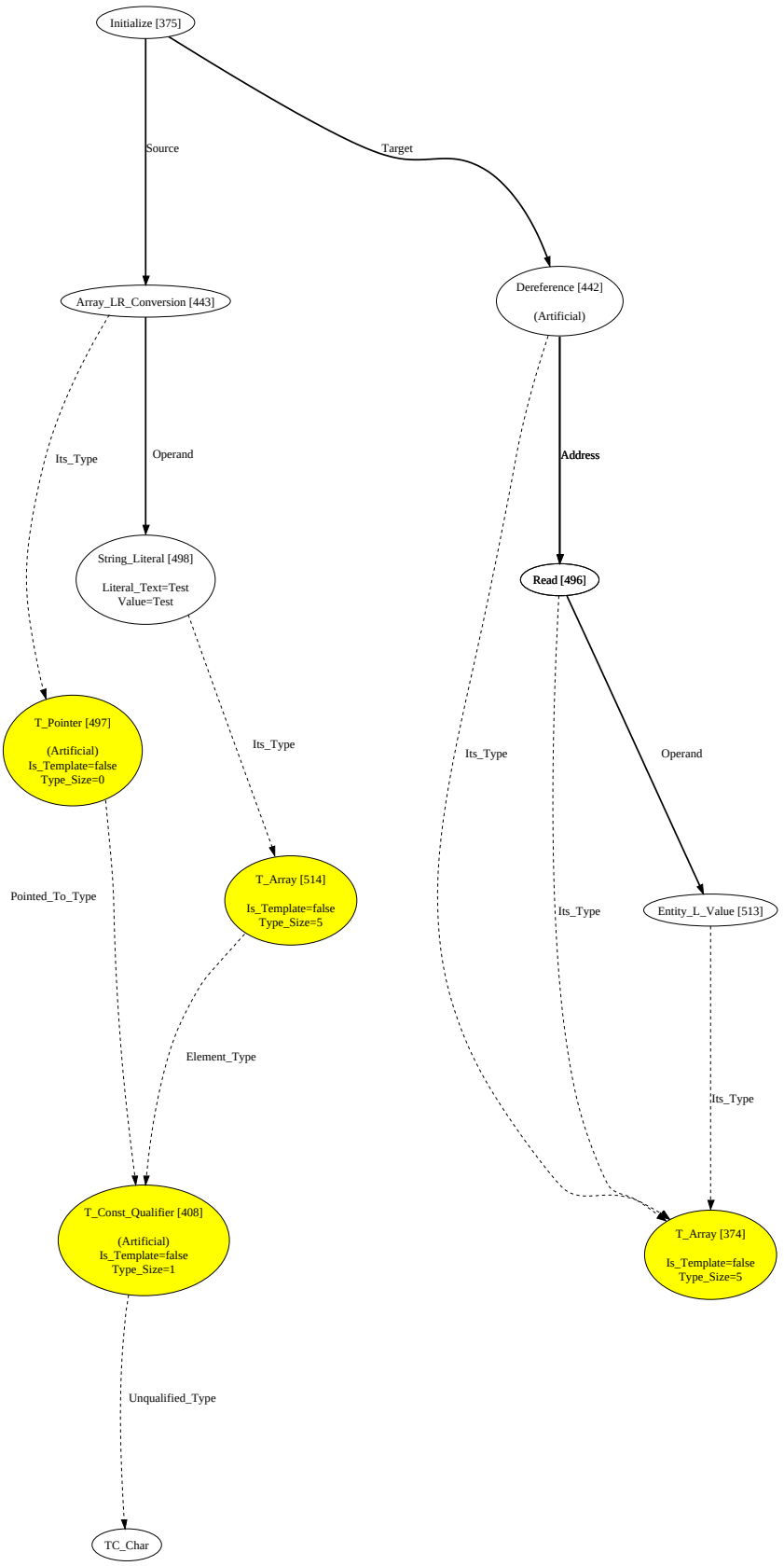

Abbildung 4.2: In diesem Beispiel soll einem `char`-Array ein String zugewiesen werden. Dieses entspricht `char t[5] = "Test";`

ANHANG A

IML-Knotenreferenz

Im Folgenden werden alle für den Interpreter zur Ausführung relevanten IML-Knoten näher erläutert. Knoten die von den Klassen *T_Node* und *O_Node* abgeleitet sind, werden hier nicht aufgeführt, da ihr Aufbau weder einer Ausführung benötigen, noch diese für die hier erwähnten Knoten relevant sind.

Neben der operationellen Semantik der einzelnen Knoten werden einige Beispiele aufgeführt und das Vorgehen des Interpreters bei den jeweiligen Knoten erläutert.

Spezielle Knoten die nicht vom Interpreter unterstützt werden, da eine Interpretation der Sprache die sie repräsentieren nicht möglich ist, werden hier nicht aufgeführt. Zu diesen Knoten gehören sowohl Ada- als auch Java-spezifische Repräsentationen.

A.1 Variablen

Die folgenden Knoten repräsentieren die Erzeugung, Zuweisung als auch den Zugriff auf Variablen. Jede Variable ist abgeleitet von einem *O_Node*, der sowohl den Namen der Variable als auch deren Typ beinhaltet.

A.1.1 Sichtbarkeit

Diese Knoten repräsentieren die Sichtbarkeitsdefinition von einzelnen Variablen innerhalb von Blöcken. Während der *Begin_Of_Lifetime*-Knoten nur am Anfang eines Blocks definiert ist, so wird der *End_Of_Lifetime*-Knoten nicht nur am Ende eines Blocks sondern auch an sämtlichen Punkten referenziert, an denen der Block verlassen werden kann.

A.1.1.1 Begin_Of_Lifetime

Dieser Knoten verweist auf jene Variablen, deren Sichtbarkeit ab hier beginnt. Sie werden in die Symboltabelle eingetragen und der benötigte Speicher reserviert, wenn auch nicht initialisiert. Das bedeutet, daß ab diesem Zeitpunkt zwar dem Interpreter die Existenz dieser Variable bekannt ist, aber weder Kontruktoren aufgerufen werden, noch irgend eine andere Form der Initialisierung stattfindet. Der Wert der Variable wird dann durch den vorherigen Zustand des zugewiesenen Speichers bestimmt.

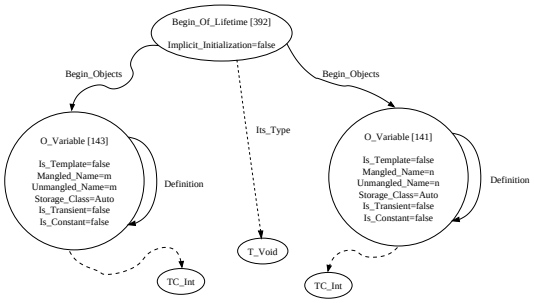

Abbildung A.1: Knoten Begin_Of_Lifetime

$$\langle n, \sigma \rangle \Rightarrow_C \sigma \cup n \tag{A.1}$$

A.1.1.2 End_Of_Lifetime

Dieser Knoten steht am Ende jeder Funktion und zeigt auf die Variablen, deren Sichtbarkeit endet und die somit gelöscht werden sollten. Handelt es sich dabei um Instanzen von Klassen, so wird der entsprechende Destruktor ausgeführt. Der zuvor reservierte Speicher wird anschließend freigegeben.

Wenn die Sichtbarkeit eines *Pointers* endet, so wird der Speicher des Pointers freigegeben und nicht der des Objektes auf das verwiesen wird. Während in C und C++ der Programmierer dafür zuständig ist nicht mehr referenzierte Objekte zu löschen, übernimmt in Java der *Garbage Collector* (siehe Kapitel 3.2.2.1) diese Aufgabe.

$$\langle n, \sigma \rangle \Rightarrow_C \sigma \cap n \tag{A.2}$$

A.1.2 Zuweisungen

Die folgenden Knoten repräsentieren Zuweisungen von Werten zu Variablen. Bei den Werten die zugewiesen werden, kann es sich sowohl um Literale als auch um komplexe Ausdrücke handeln, die zu Literalen aufgelöst werden können.

A.1.2.1 Assignment

Hierbei handelt es sich um eine simple Zuweisung eines *RValue* zu einem *LValue*. Der *RValue* kann hierbei ein beliebig komplexer Ausdruck sein, Bedingung ist nur die Auflösung zum selben Typen wie der des *LValues*. Wie diese Konvertierungen behandelt werden, ist näher in Kapitel 3.4.3 beschrieben.

$$\frac{\langle a, \sigma \rangle \Rightarrow_A n}{\langle X := a, \sigma \rangle \Rightarrow_C \sigma[X \leftarrow n]} \tag{A.3}$$

A.1.2.2 Initialize

Bei diesem Knoten handelt es sich um die Initialisierung einer Variable. Das Resultat einer Verwendung der Variable vor diesem Knoten ist undefiniert. Die eigentliche Erstellung der Variable, sprich die Reservierung des Speichers und das Eintragen in die Symboltabelle, erfolgt bereits vor diesem Knoten, wenn der Interpreter auf den entsprechenden *Begin_Of_Lifetime*-Knoten trifft.

Daher wird die `source`-Kante des Knoten aufgelöst, das Resultat entsprechend dem Typen des Knotens an der `target`-Kante konvertiert und anschließend in den bereits reservierten Speicher geschrieben.

$$\frac{\langle a, \sigma \rangle \Rightarrow_A n}{\langle X := a, \sigma \rangle \Rightarrow_C \sigma \cup (X \leftarrow n)} \tag{A.4}$$

A.1.2.3 Shortcut_Assignment

Bei der *Shortcut-Assignment* handelt es sich um eine kompakte Schreibweise einer Zuweisung, bei der dem *Target* ein Wert zugewiesen wird, der der Anwendung einer bestimmten

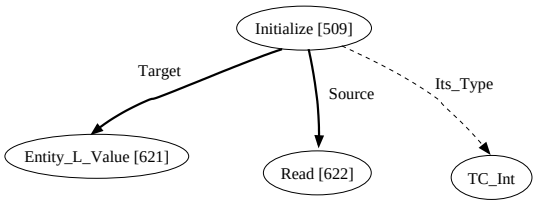

Abbildung A.2: Knoten Initialize

Operation auf dessen Ursprungswert entspricht. Zum Beispiel wird `x *= 5` aufgelöst zu `x = x * 5`.

$$\frac{\langle m_0, \sigma\rangle \Rightarrow_A m_1 \quad \langle n * m_1, \sigma\rangle \Rightarrow_A k}{\langle n* = m_0, \sigma\rangle \Rightarrow_C \sigma\,[n := k]} \tag{A.5}$$

$$\frac{\langle m_0, \sigma\rangle \Rightarrow_A m_1 \quad \langle n/m_1, \sigma\rangle \Rightarrow_A k}{\langle n/ = m_0, \sigma\rangle \Rightarrow_C \sigma\,[n := k]} \tag{A.6}$$

$$\frac{\langle m_0, \sigma\rangle \Rightarrow_A m_1 \quad \langle n + m_1, \sigma\rangle \Rightarrow_A k}{\langle n+ = m_0, \sigma\rangle \Rightarrow_C \sigma\,[n := k]} \tag{A.7}$$

$$\frac{\langle m_0, \sigma\rangle \Rightarrow_A m_1 \quad \langle n - m_1, \sigma\rangle \Rightarrow_A k}{\langle n- = m_0, \sigma\rangle \Rightarrow_C \sigma\,[n := k]} \tag{A.8}$$

$$\frac{\langle m_0, \sigma\rangle \Rightarrow_A m_1 \quad \langle n\%m_1, \sigma\rangle \Rightarrow_A k}{\langle n\% = m_0, \sigma\rangle \Rightarrow_C \sigma\,[n := k]} \tag{A.9}$$

A.1.2.4 Aggregate

Der *Aggregate*-Knoten steht für die Zuweisung eines *Arrays* anhand eines *Aggregats*. Dieses besteht aus einer Liste aus Elementen deren Typ dem entspricht, der im *Array* enthalten ist. Verwendung findet der Knoten bei der Initialisierung von *Arrays*, in denen der *Aggregate*-Knoten an der *Source*-Kante des entsprechenden *Initialize*-Knotens hängt.

Der Interpreter erzeugt ein *Array* dessen Größe und Typ anhand des *T_Node*-Knotens an der *Its_Type*-Kante des *Aggregats* ermittelt wird. Anschließend werden die einzelnen *Initializers* aufgelöst und in die entsprechenden Felder des *Arrays* eingetragen.

A.1.3 Read

Dieser Knoten steht für das Lesen einer Variable aus dem Speicher. Bei dem Operanden handelt es sich um einen *Entity_L_Value*-Knoten, der letztendlich auf den *O_Node* der auszulesenden Variable zeigt. Der Wert des Knotens wird durch den Typ der Variable bestimmt, auf den letztendlich verwiesen wird.

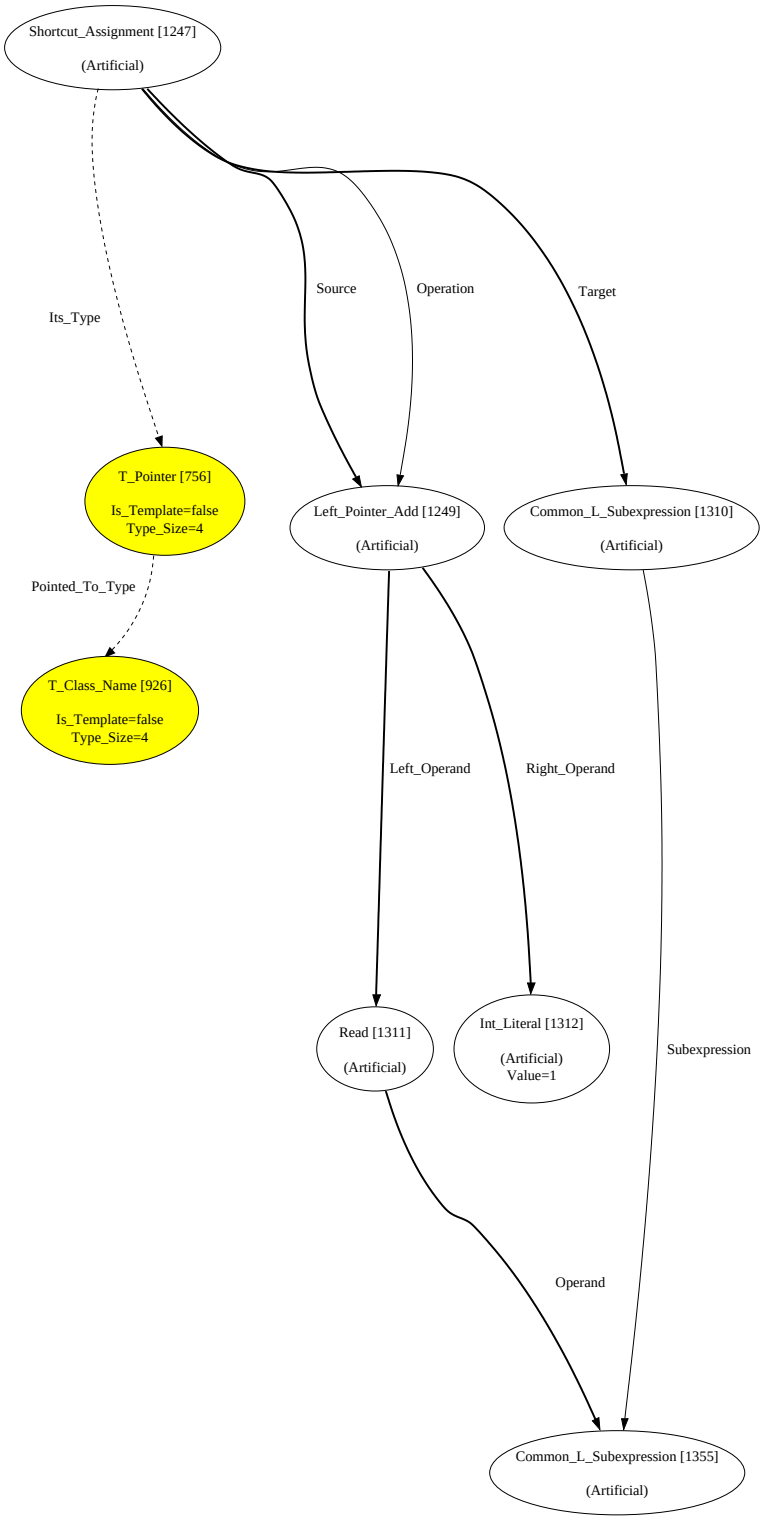

Abbildung A.3: Knoten Shortcut_Assignment, dargestellt ist hier die Shortcut-Zuweisung einer Pointer-Addition. Das Ziel der Operation ist sogleich ein Teil der Berechnung selbst, verwiesen mit der *Common_L_Subexpression*, die, was hier nicht dargestellt ist, zu dem ursprünglichen Pointer (auf eine Klasse) aufgelöst wird.

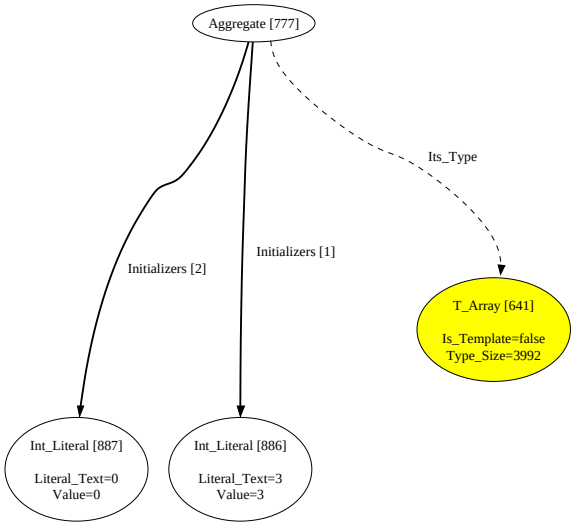

Abbildung A.4: Knoten Aggregate

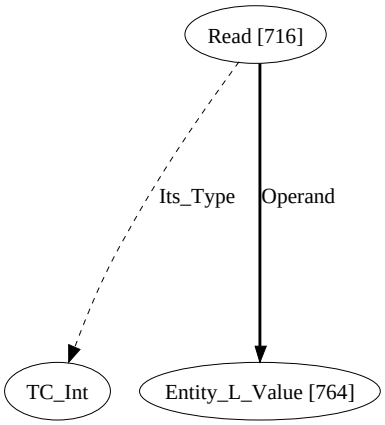

Abbildung A.5: Knoten Read

$$\langle X, \sigma \rangle \Rightarrow_A \sigma(X) \tag{A.10}$$

A.2 Konstanten

Die folgenden Knoten repräsentieren Konstanten die sich zu den entsprechenden Literalen auflösen lassen.

A.2.1 Integer_Constant

Der Knoten der einen konstanten Integerwert darstellt, verweist auf ein *Integer_Literal* mit dem jeweiligen konstanten Wert. Trifft der Interpreter auf eine Konstante, so wird aus dieser ein Symbol generiert und dieses an den überliegenden Knoten zurückgegeben.

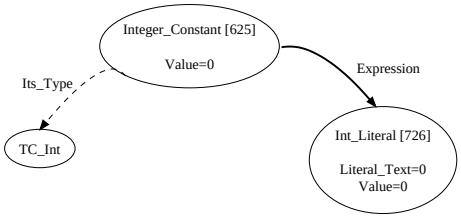

Abbildung A.6: Knoten Integer_Constant

Mit σ_* beliebig:

$$\langle n, \sigma_* \rangle \Rightarrow_A n \tag{A.11}$$

A.3 Funktionen

Die folgenden Knoten repräsentieren den Aufruf von Funktionen und die Behandlung von deren Parametern. Die die Rückgabewerte betreffenden Knoten sind hier ebenfalls aufgeführt.

A.3.1 Aufruf

Die hier aufgeführten Knoten repräsentieren den eigentlichen Funktionsaufruf, der die Erzeugung eines neuen *FunctionCall*-Objektes erzeugt, wodurch auch eine neue Symboltabelle angelegt wird. Wie Funktionen vom Interpreter behandelt werden, ist Kapitel 3.4 zu entnehmen.

A.3.1.1 Direct_Call

Bei diesem Knoten handelt es sich um die Repräsentation eines Funktionsaufrufs. Dieser verweist sowohl auf den entsprechenden Knoten der die aufzurufende Funktion repräsentiert und auf die *Pre_Call_Link*- und *Post_Call_Link*-Knoten, die jeweils vor beziehungsweise nach der Ausführung der eigentlichen Funktion ausgeführt werden müssen. Teil der Unterknoten

sind die *Copy_In*-Knoten, die das Kopieren der Funktionsparameter in dessen Symboltabelle darstellen.

Der Typ des Knotens ist gleich dem des Rückgabewertes der aufzurufenden Funktion. Die Funktion selbst ist über die *Routine_Expression*-Kante referenziert.

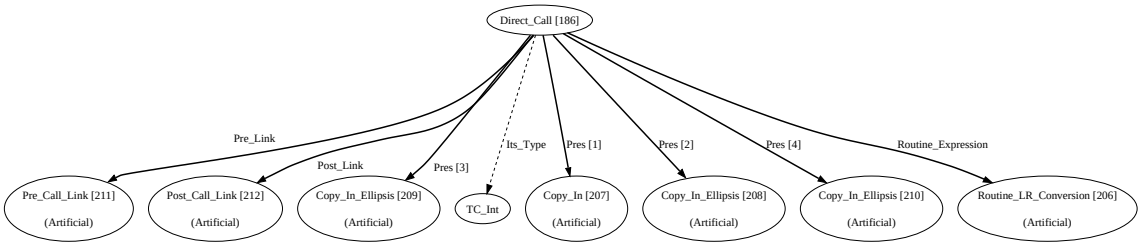

Abbildung A.7: Knoten Direct_Call

A.3.1.2 Indirect_Call

Dieser Knoten repräsentiert den Aufruf von Funktionen anhand von Funktionspointern, wie sie in Kapitel 3.3.3 beschrieben sind. Anstelle eines *O_Routine*-Knotens, wird über die *Routine_Expression*-Kante ein *Read*-Knoten referenziert, der auf den Funktionspointer verweist.

A.3.2 Parameter

Die folgenden Knoten repräsentieren das Kopieren von Parametern in Funktionsaufrufe. Diese finden sowohl bei internen (siehe Kapitel 3.4.1) als auch bei externen (siehe Kapitel 3.4.2) Funktionsaufrufen Verwendung.

A.3.2.1 Copy_In

Dieser Knoten steht für die Übergabe einer Variable als Kopie an eine Funktion. Es wird eine Kopie des Speicherbereichs angelegt, auf den der Knoten verweist. Keine der eventuellen Modifikationen des so erzeugten Parameters beeinflusst den Ausgangsparameter. Bei der Ausgangsvariable darf es sich sowohl um einen *LValue* als auch um einen *RValue* handeln.

Wird ein Parameter als Referenz übergeben, so wird er intern als *Pointer* behandelt. Es wird zwar eine Kopie des *Pointers* angelegt, der Speicherbereich auf den diese Kopie verweist ist aber mit dem des Originals identisch. So läßt sich der Wert der übergebenen Parameter zwar nicht ändern, jedoch der Wert der Variable auf die verwiesen wird.

Jeder dieser Knoten verweist auf einen *O_Parameter*-Knoten, der die relevanten Informationen über den Parameter beinhaltet. Hierzu gehört sowohl dessen Datentyp als auch dessen Name.

Dieser Parameter wird in die Symboltabelle des Funktionsaufrufes kopiert, wobei der Interpreter den Index des *O_Parameter*-Knotens als Schlüssel für die Symbol-Hashmap (siehe Kapitel 3.1.2) verwendet.

Bei externen Funktionsaufrufen unterscheidet sich das Verfahren etwas, denn dort werden die *O_Parameter*-Knoten größtenteils ignoriert. Jeder Parameter erhält einen fortlaufenden Index, angefangen bei Eins. Dies erleichtert den direkten Zugriff auf die Parameter, deren Wertigkeiten über die Schnittstellen an die jeweiligen Funktionen übergeben werden.

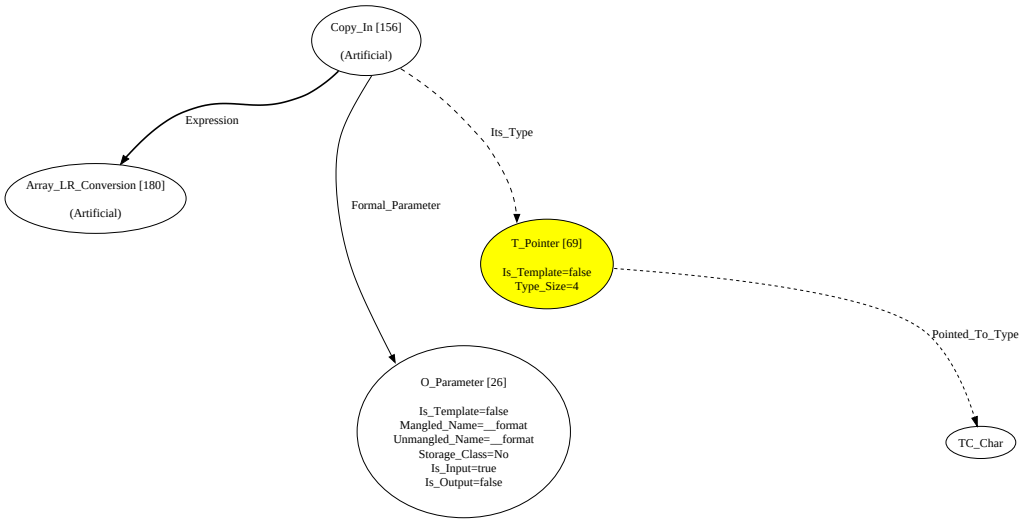

Abbildung A.8: Knoten Copy_In, dargestellt ist hier der erste Parameter der Funktion `printf`, einem Pointer auf ein `char`-Array.

A.3.2.2 Copy_In_Ellipsis

Dieser Knoten steht für die Kopie eines Parameters in einen Funktionsaufruf. Im Gegensatz zum normalen *Copy_In*-Knoten handelt es sich hierbei allerdings um einen Parameter aus einer Parameterliste, wie sie in C und C++ existieren. Dies bedeutet die Funktion akzeptiert eine beliebige Anzahl an zusätzlichen Parametern und daher existiert auch kein Verweis auf einen Parameterknoten vom Typ *O_Parameter*. Die Reihenfolge, in der diese Knoten an einem *Direct_Call*-Knoten, dem Funktionsaufruf, hängen, bestimmt, den wievielten zusätzlichen Parameter dieser Knoten darstellt.

Da ohne einen Verweis auf einen *O_Parameter*-Knoten dieser Parameter keinen eindeutigen Index hat, über den die aufzurufende Funktion auf ihn zugreifen könnte, wird ein fortlaufender Index verwendet. So wird dem ersten optionalen Parameter der Wert Eins zugeordnet, dem zweiten der Wert Zwei und so weiter.

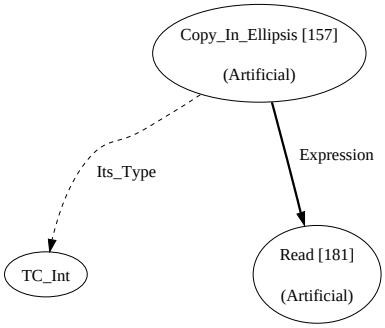

Abbildung A.9: Knoten Copy_In_Ellipsis

A.3.2.3 Copy_This_In

Bei diesem Knoten handelt es sich nicht um die Repräsentation eines normalen Parameters einer Funktion, sondern um die Anweisung einen Verweis auf ein bestimmtes Objekt als `this`-Pointer in die aufgerufene Funktion zu kopieren.

Dieser Knoten wird also dann verwendet, wenn eine nicht statische Memberfunktion eines Objekts aufgerufen wird und beinhaltet dann einen Verweis auf dieses Objekt. So lassen sich innerhalb der Memberfunktion die Membervariablen des Objekts ansprechen.

A.3.3 Rückgabe

Diese beiden Knoten repräsentieren die Rückgabe einer Funktion, jeweils mit und ohne einen Wert an die übergeordnete Funktion zurückzugeben.

A.3.3.1 Return_With_Value

Wird dieser Knoten erreicht, wird der Wert des Operanden ermittelt und die Ausführung der aktuellen Funktion beendet. Der ermittelte Wert wird an die aufrufende Funktion zurückgegeben und gegebenenfalls dort in eine Variable gespeichert oder bei weiteren Berechnungen verwendet.

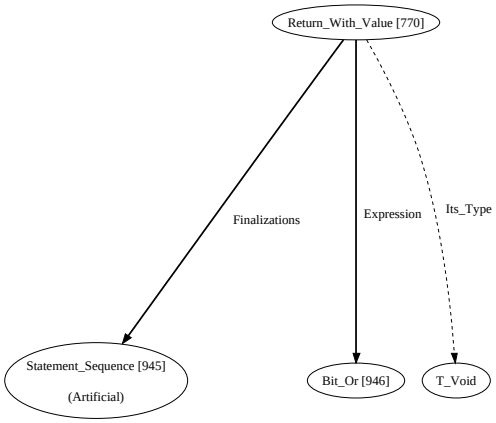

Abbildung A.10: Knoten Return_With_Value

$$\frac{\langle s_1, \sigma\rangle \Rightarrow_C \sigma_1 \quad \langle n, \sigma_1\rangle \Rightarrow_A m}{\langle X := f(s_1; return\ n), \sigma\rangle \Rightarrow_C \sigma_1\,[X \leftarrow m]} \tag{A.12}$$

A.3.3.2 Return_Without_Value

Wird dieser Knoten erreicht, so wird die Ausführung der aktuellen Funktion sofort beendet. Alle folgenden Knoten werden ignoriert, bis der Interpreter zurück kommt, zu dem eigentlichen Funktionsaufruf. Dies geschieht, indem die boolsche Variable *Return_From_Function* des aktuellen Funktions-Objekts auf `true` gesetzt wird. Diese Variable wird vor der Ausführung jeden *Statements* geprüft und diese gegebenenfalls dann ignoriert.

$$\langle f(return; s_1); s_2, \sigma\rangle \Rightarrow_C \langle s_2, \sigma\rangle \tag{A.13}$$

A.3.3.3 Copy_Out

Dieser Knoten repräsentiert Parameter wie sie in Ada existieren, die es erlauben, daß diese von der aufgerufenen Funktion manipuliert werden. Es handelt sich im Grunde um die Übergabe

einer Referenz auf das Objekt, anstelle einer Kopie des eigentlichen Wertes.

Die Übergabe von Werten als Referenz, wie es in C++ möglich ist, ist mit diesem Knoten nicht repräsentiert. Diese Referenzen sind einfache *Pointer*, die wie normale Datentypen in die Funktion kopiert werden.

A.4 Labels

Bei den *Labels* handelt es sich um Sprungadressen, daher erfolgt keine Ausführung. Die *Labels* werden vom Interpreter nicht gespeichert, sondern so lange ignoriert, bis sie benötigt werden, da die Knoten, die auf diese *Labels* angewiesen sind, immer einen Verweis auf diese beinhalten.

Stößt der Interpreter auf ein gesuchtes *Label*, wird geprüft an welcher Stelle des überliegenden Knoten sich dieses befindet. Anschließend werden die nachfolgenden Knoten ausgeführt.

A.4.1 Valued_Label

Diese Knoten stehen für die einzelnen *case*-Felder in einem *Switch_Case*-Block. Wie diese behandelt werden, ist dem Kapitel A.15.5 zu entnehmen.

A.4.2 Named_Label

Bei diesem Knoten handelt es sich um das Ziel einer `goto`-Anweisung und der Name dient zur Identifikation. Es erfolgt keine Ausführung, der Knoten wird, wenn der Interpreter auf ihn stößt, ignoriert.

A.4.3 Anonymous_Label

Innerhalb eines Switch-Blocks steht das anonyme Label für den *default*-Fall. Dieser wird ausgelöst, wenn sonst kein `case`-Fall erfüllt ist. Eine direkte Ausführung des Knotens erfolgt nicht.

A.5 Literale

Bei den folgenden Knoten handelt es sich um Repräsentationen von Literalen. Der Wert des Literals ist direkt in dem Knoten gespeichert. Dieser wird vom Interpreter in ein entsprechendes Symbol umgewandelt und der Speicher sowohl reserviert als auch beschrieben.

A.5.1 Boolean_Literal

Bei diesem Literal handelt es sich um eine Repräsentation eines booleschen Werts, der entweder *wahr* oder *falsch* sein kann. Dieser Wert läßt sich in andere primitive Datentypen konvertieren und entspricht dann dem integralen Wert Eins beziehungsweise Null.

$$\langle t, \sigma \rangle \Rightarrow_A t \tag{A.14}$$

$$\frac{\langle t, \sigma \rangle \Rightarrow_B true}{\langle t, \sigma \rangle \Rightarrow_A 1} \tag{A.15}$$

$$\frac{\langle t, \sigma \rangle \Rightarrow_B false}{\langle t, \sigma \rangle \Rightarrow_A 0} \tag{A.16}$$

A.5.2 Char_Literal

Bei dem *Char_Literal* handelt es sich um die Repräsentation eines einzelnen Zeichens, das normalerweise acht Bit im Speicher einnimmt. Hiermit lassen sich also alle ASCII-Zeichen abbilden. Da es sich bei einem Character im Grunde um einen acht Bit Integer handelt, sind entsprechende Konvertierungen möglich.

$$\langle n, \sigma \rangle \Rightarrow_A n \tag{A.17}$$

$$\frac{\langle n, \sigma \rangle \Rightarrow_A 0}{\langle n, \sigma \rangle \Rightarrow_B false} \tag{A.18}$$

$$\langle n, \sigma \rangle \Rightarrow_B true \tag{A.19}$$

A.5.3 Int_Literal

Bei diesem Literalknoten handelt es sich um die Repräsentation von integralen Zahlenwerten, deren Größe über den *T_Node* an der *Its_Type*-Kante hängt. Daher können sowohl normale 32-Bit Integer als auch jene mit 64 oder gar mehr Bits mit diesem Knoten abgebildet werden.

$$\langle n, \sigma \rangle \Rightarrow_A n \tag{A.20}$$

$$\frac{\langle n, \sigma \rangle \Rightarrow_A 0}{\langle n, \sigma \rangle \Rightarrow_B false} \tag{A.21}$$

$$\langle n, \sigma \rangle \Rightarrow_B true \tag{A.22}$$

A.5.4 Floating_Point_Literal

Dieser Knoten steht für Fließkommazahlen nahezu beliebiger Bitlänge. Genau wie beim *Int_Literal* wird die Größe des benötigten Speichers durch einen anhängenden *T_Node* bestimmt. Sowohl Werte vom Typ `float` als auch `double` werden hiermit repräsentiert.

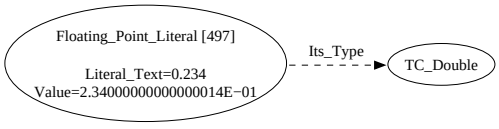

Abbildung A.11: Knoten Floating_Point_Literal

$$\langle n, \sigma \rangle \Rightarrow_A n \tag{A.23}$$

$$\frac{\langle n, \sigma \rangle \Rightarrow_A 0}{\langle n, \sigma \rangle \Rightarrow_B false} \tag{A.24}$$

$$\langle n, \sigma \rangle \Rightarrow_B true \tag{A.25}$$

A.5.5 String_Literal

Ein String-Literal wird durch diesen Knoten repräsentiert. Dabei verfügt der Knoten über zwei Werte, die den jeweiligen String unterschiedlich darstellen, den *Value* und den *Literal_Text*. Der *Literal_Text* stellt dabei genau das dar, was im Quelltext an seiner Stelle steht. Das bedeutet, daß weder Sonderzeichen wie \n in ihr ASCII-Äquivalent umgewandelt wurden, noch daß das in C und C++ benötigte Nullbyte (\0) angehängt wird. *Value* hingegen beinhaltet diese Konvertierungen und wird daher vom Interpreter verwendet.

Problematisch ist allerdings die Darstellung von Strings mit 4-Byte-Zeichen, wie sie in C++ möglich sind. Dort liefern sowohl *Value* als auch *Literal_Text* fehlerhafte Werte, wie in Kapitel 3.3.4 genauer beschrieben wird.

Im Gegensatz zu den anderen Literalen, werden String-Literale immer in der globalen Symboltabelle gespeichert, so sie dort noch nicht existieren. Dies bedeutet, daß am Ende der Ausführung eines Programms alle String-Literale die während der Interpretation verwendet wurden, im Speicher einmalig abgelegt werden.

$$\langle n, \sigma \rangle \Rightarrow_A n \tag{A.26}$$

$$\frac{\langle n, \sigma \rangle \Rightarrow_A null}{\langle n, \sigma \rangle \Rightarrow_B false} \tag{A.27}$$

$$\langle n, \sigma \rangle \Rightarrow_B true \tag{A.28}$$

A.6 Subexpressions

Die folgenden Knoten stellen Unterknoten komplexerer Operationen dar. Sie repräsentieren die Zwischenspeicherung von Ergebnissen verschiedener Operationen.

A.6.1 Common_L_Subexpression

Dieser Knoten ist Teil komplexerer Operationen, wie zum Beispiel dem *Prefix_Operator* oder dem *Shortcut_Assignment*-Knoten, wo er auf existente *LValues* verweist. Der Aufbau eines solchen Knotens ist den Abbildungen A.3 und A.14 zu entnehmen.

$$\langle L, \sigma \rangle \Rightarrow_A \sigma(L) \tag{A.29}$$

A.6.2 Common_R_Subexpression

Im Gegensatz zum *Common_L_Subexpression*-Knoten gibt dieser keinen *LValue*, sondern einen *RValue* zurück und symbolisiert die Zwischenspeicherung von Ergebnissen einzelner Operationen. Verwendet wird dieser Knoten beispielsweise bei dem *Postfix_Operator*, wie in der Abbildung A.15 dargestellt ist.

A.7 Arithmetik

Die folgenden Knoten repräsentieren die grundlegende Mathematik, wie sie von den Programmiersprachen unterstützt werden. Zu diesen Operationen gehören sowohl die Addition, die

Multiplikation als auch die Berechnung des Modulo.

Überladene Operatoren, wie sie in C++ möglich sind, werden durch diese Knoten nicht abgedeckt. Dieser werden anhand von normalen Funktionsaufrufen repräsentiert (siehe Kapitel A.3.1.1).

A.7.1 Arithmetic_Add

Arithmetic_Add stellt die mathematische Addition zweier nahezu beliebiger Datenwerte dar. Das Ergebnis der Addition kann per *Its_Type*-Kante ermittelt werden und ist abhängig von den beiden Operatoren, die mit *Left_Operand* und *Right_Operand* ausgewiesen sind. Bei den Operanden kann es sich um beliebig komplexe Ausdrücke handeln, die sich zu primitiven Datentypen auflösen lassen.

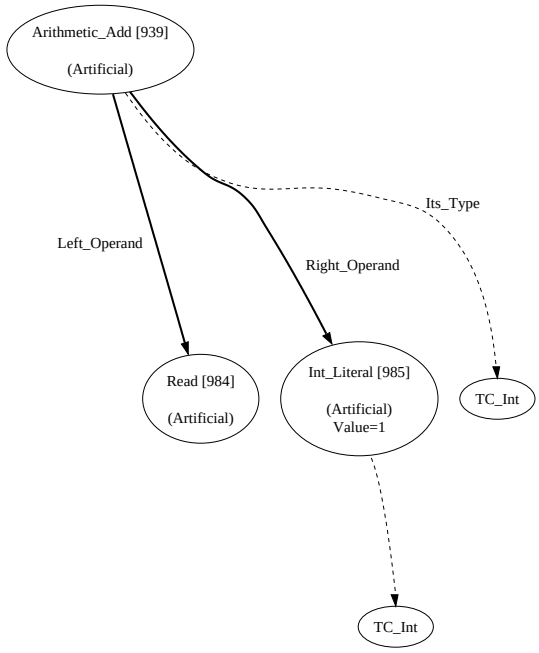

Abbildung A.12: Knoten Arithmetic_Add

$$\frac{\langle a_1, \sigma\rangle \Rightarrow_A n_1 \quad \langle a_2, \sigma\rangle \Rightarrow_A n_2}{\langle a_1 + a_2, \sigma\rangle \Rightarrow_A n} \tag{A.30}$$

für n gleich Summe von n_1 und n_2.

A.7.2 Multiply

Der *Multiply*-Knoten steht für eine mathematische Multiplikation zweier Werte und steht dabei generisch für beliebige Datentypen. Der Typ vom Ergebnis ist per *Its_Type*-Kante referenziert, während die beiden Operanten mit *Left_Operand* und *Right_Operand* angesprochen werden können. Bei diesen muss es sich nicht, wie im Beispiel (Abbildung A.13), um Literale handeln, sondern um wesentlich komplexere Ausdrücke die sich dafür in Literale auflösen lassen.

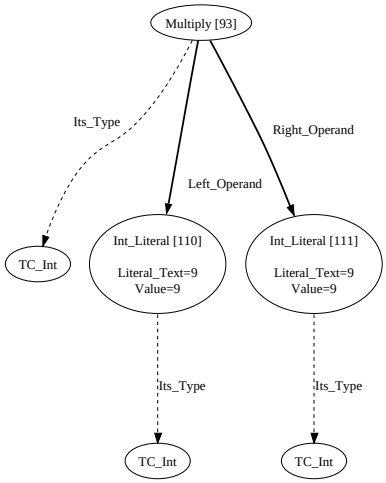

Abbildung A.13: Knoten Multiply

$$\frac{\langle a_1, \sigma\rangle \Rightarrow_A n_1 \quad \langle a_2, \sigma\rangle \Rightarrow_A n_2}{\langle a_1 * a_2, \sigma\rangle \Rightarrow_A n} \tag{A.31}$$

$$\tag{A.32}$$

für n gleich Produkt von n_1 und n_2.

A.7.3 Divide

Der Divide-Knoten steht für die generische mathematische Division zweier Datenwerte. Der Typ des Quotienten kann mit *Its_Type* und der Typ des Dividenden und Divisors mit *Left_Operand* beziehungsweise *Right_Operand* ermittelt werden. Bei beiden kann es sich um beliebig komplexe Ausdrücke handeln, die sich auf einen unterstützten Datenwert reduzieren lassen müssen.

$$\frac{\langle a_1, \sigma\rangle \Rightarrow_A n_1 \quad \langle a_2, \sigma\rangle \Rightarrow_A n_2}{\langle a_1 / a_2, \sigma\rangle \Rightarrow_A n} \tag{A.33}$$

für n gleich Quotient von n_1 und n_2.

A.7.4 Arithmetic_Subtract

Bei diesem Knoten handelt es sich um die Repräsentation einer arithmetischen Subtraktion. Die beiden Operanden werden zuerst zu dem höchstwertigsten der beiden Datentypen konvertiert und dann von einander substrahiert. Der Typ des Ergebnisses ist gleich dem des höherwertigen Datentyps.

$$\frac{\langle n_0, \sigma\rangle \Rightarrow_A a_0 \quad \langle n_1, \sigma\rangle \Rightarrow_A a_1}{\langle n_0 - n_1, \sigma\rangle \Rightarrow_A a_2} \tag{A.34}$$

wobei a_2 gleich $a_0 - a_1$ ist.

A.7.5 Modulo

Dieser Knoten steht für die Modulo-Rechnung, bei der das Ergebnis der Rest der Ganzzahldivision der beiden Operanden entspricht.

Mit a_2 gleich $a_0 \% a_1$:

$$\frac{\langle n_0, \sigma\rangle \Rightarrow_A a_0 \quad \langle n_1, \sigma\rangle \Rightarrow_A a_1}{\langle n_0 \% n_1, \sigma\rangle \Rightarrow_A a_2} \tag{A.35}$$

A.7.6 Prefix_Operator

Dieser Operator inkrementiert (oder dekrementiert, je nach Operator) einen Integerwert um Eins und gibt den neuen Wert zurück.

Innerhalb der IML wird diese Berechnung fehlerhaft abgebildet, wenn diese Operation allein steht, ohne daß der Wert einer Variable zugewiesen oder in einer anderen Operation verwendet wird. In diesem Fall haben die Operanden der Operation und der Operator selbst den Typ *T_Node* anstelle des eigentlichen Typs, zum Beispiel *TC_Int*. Dies führt zu Fehlern während der Berechnung und somit zum Abbruch der Interpretation.

$$\frac{\langle n := n + 1, \sigma_0\rangle \Rightarrow_C \sigma_1}{\langle ++n, \sigma_0\rangle \Rightarrow_A \sigma_1(n)} \tag{A.36}$$

A.7.7 Postfix_Operator

Dieser Operator gibt den ursprünglichen Wert des Operanden zurück, inkrementiert (oder dekrementiert, je nach Operator) ihn vorher um Eins. Bei dem Rückgabewert handelt es sich um keinen LValue.

Die *Common_R_Subexpression* darf hierbei nicht nur aufgelöst werden, sondern es muss eine komplette Kopie des Symbols inklusive des dazugehörigen Speicherbereichs angelegt werden. Ansonsten würde die Durchführung der arithmetischen Operation den zurückgegebenen Datenwert verändern.

$$\frac{\langle n + 1, \sigma_0\rangle \Rightarrow_A m \quad \langle n := m, \sigma_0\rangle \Rightarrow_C \sigma_1}{\langle n++, \sigma_0\rangle \Rightarrow_A \langle n, \sigma_1\rangle} \tag{A.37}$$

A.8 Bit-Operatoren

Die folgenden Operatoren arbeiten nicht mit den eigentlichen Werten der Variablen, sondern direkt auf deren binären Repräsentationen im Speicher. Daher sind nur Integer-Werte als Parameter zulässig.

A.8.1 Bit_And

Dieser Knoten gibt die binäre Verundung der beiden Operanden wieder. Hierbei ist ein Bit genau dann gesetzt, wenn es bei beiden Operanden ebenfalls gesetzt war.

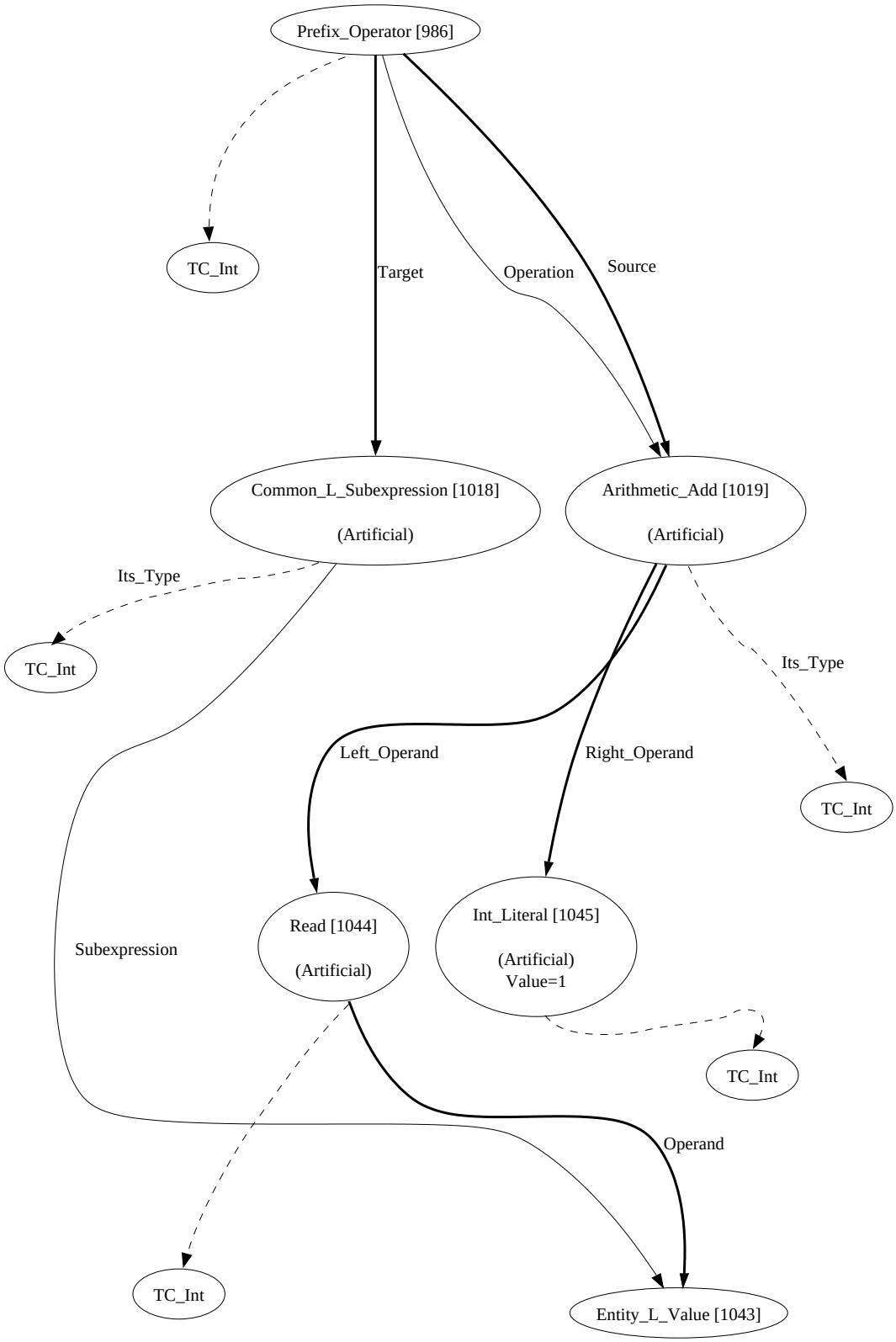

Abbildung A.14: Knoten Prefix_Operator

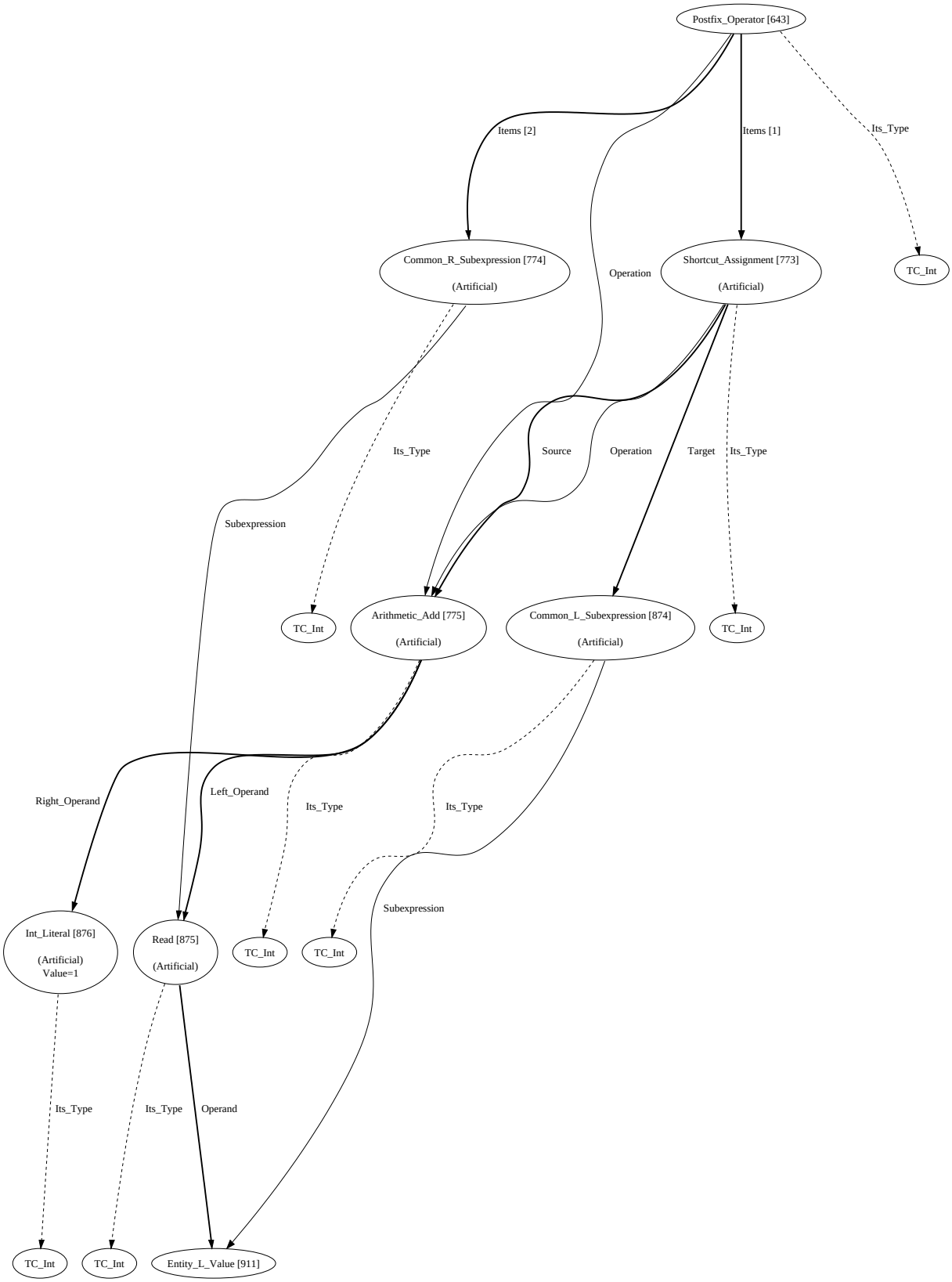

Abbildung A.15: Knoten Postfix_Operator

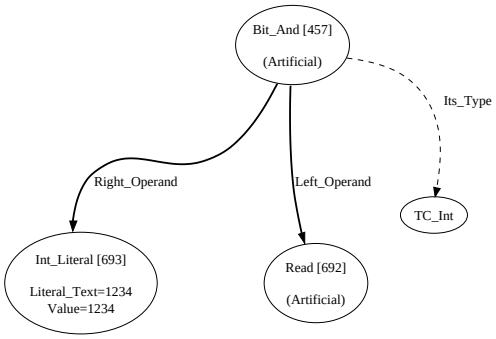

Abbildung A.16: Knoten Bit_And

$$\frac{\langle a_1, \sigma\rangle \Rightarrow_A n_1 \quad \langle a_2, \sigma\rangle \Rightarrow_A n_2}{\langle a_1 \& a_2, \sigma\rangle \Rightarrow_A n_3} \tag{A.38}$$

für n_3 gleich $n_1 \& n_2$.

A.8.2 Bit_Or

Dieser Knoten gibt die binäre Veroderung der beiden Operanden wieder. Hierbei wird ein Bit genau dann gesetzt, wenn es bei mindestens einem der beiden Operanden ebenfalls gesetzt war.

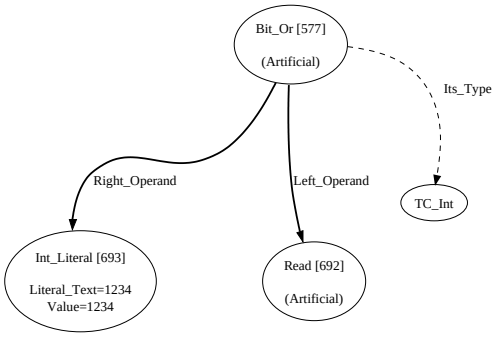

Abbildung A.17: Knoten Bit_Or

$$\frac{\langle a_1, \sigma\rangle \Rightarrow_A n_1 \quad \langle a_2, \sigma\rangle \Rightarrow_A n_2}{\langle a_1 | a_2, \sigma\rangle \Rightarrow_A n_3} \tag{A.39}$$

für n_3 gleich $n_1 | n_2$.

A.8.3 Bit_Xor

Dieser Knoten gibt die binäre exklusive Veroderung der beiden Operanden wieder. Hierbei wird ein Bit genau dann gesetzt, wenn es bei genau einem der beiden Operanden ebenfalls gesetzt war.

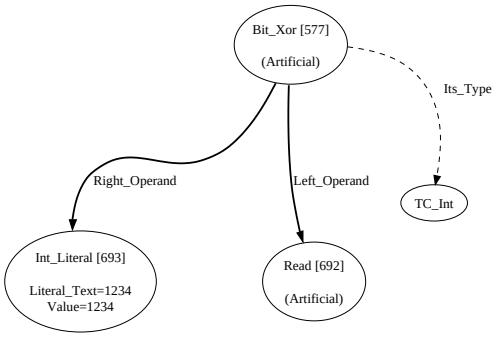

Abbildung A.18: Knoten Bit_Xor

$$\frac{\langle a_1, \sigma\rangle \Rightarrow_A n_1 \quad \langle a_2, \sigma\rangle \Rightarrow_A n_2}{\langle a_1 \ XOR \ a_2, \sigma\rangle \Rightarrow_A n_3} \tag{A.40}$$

für n_3 gleich $n_1 \ XOR \ n_2$.

A.8.4 Bit_Not

Hierbei handelt es sich um die Repräsentation der binären Negierung. Da diese Operation unabhängig vom Typ des Operanden ist, ist sie nur für Integer-Typen definiert. Die einzelnen Bits des Typen werden hierbei einzeln und unabhängig voneinander negiert.

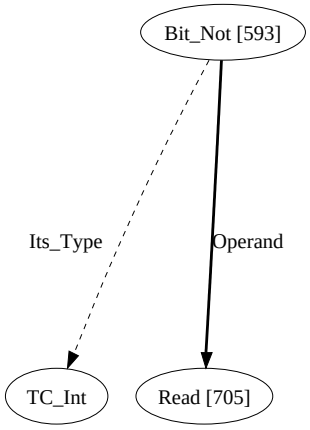

Abbildung A.19: Knoten Bit_Not

$$\langle \neg n, \sigma\rangle \Rightarrow_A m \tag{A.41}$$

wobei m die binäre Negation von n ist.

A.9 Boolesche Ausdrücke

Die folgenden Knoten werden vom Interpreter zu einem boolschen Datentyp aufgelöst, der den Wert `true` oder `false` annehmen kann.

A.9.1 Equal

Dieser Knoten überprüft die beiden Operanden auf ihre Gleichheit. Unterscheiden sich die Datentypen, so werden sie entsprechend der Konventionen der jeweiligen Programmiersprache konvertiert (siehe Kapitel 3.4.3).

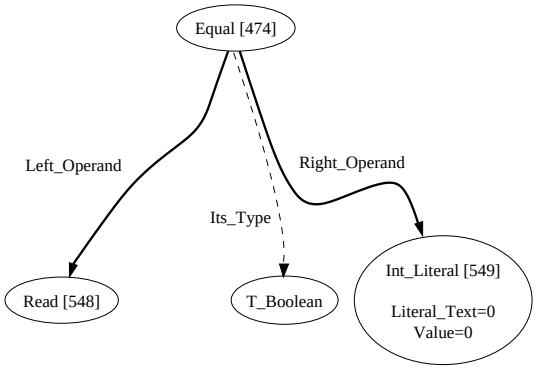

Abbildung A.20: Knoten Equal

$$\frac{\langle a_1, \sigma\rangle \Rightarrow_A n_1 \quad \langle a_2, \sigma\rangle \Rightarrow_A n_2}{\langle a_1 = a_2, \sigma\rangle \Rightarrow_B true} \tag{A.42}$$

falls n_1 und n_2 gleich

$$\frac{\langle a_1, \sigma\rangle \Rightarrow_A n_1 \quad \langle a_2, \sigma\rangle \Rightarrow_A n_2}{\langle a_1 = a_2, \sigma\rangle \Rightarrow_B false} \tag{A.43}$$

falls n_1 und n_2 nicht gleich

A.9.2 Unequal

Dieser Knoten überprüft, ob die beiden Operanden sich unterscheiden. Im Prinzip handelt es sich dabei um die Negation des Equal-Knoten.

$$\frac{\langle a_1, \sigma\rangle \Rightarrow_A n_1 \quad \langle a_2, \sigma\rangle \Rightarrow_A n_2}{\langle a_1 \neq a_2, \sigma\rangle \Rightarrow_B true} \tag{A.44}$$

falls n_1 und n_2 nicht gleich

$$\frac{\langle a_1, \sigma\rangle \Rightarrow_A n_1 \quad \langle a_2, \sigma\rangle \Rightarrow_A n_2}{\langle a_1 \neq a_2, \sigma\rangle \Rightarrow_B false} \tag{A.45}$$

falls n_1 und n_2 gleich

A.9.3 Greater_Or_Equal

Dieser Knoten steht für die Überprüfung ob der linke Operand größer oder gleich dem rechten Operanden ist. Der Typ dieses Knotens ist `boolean`.

$$\frac{\langle a_1, \sigma\rangle \Rightarrow_A n_1 \quad \langle a_2, \sigma\rangle \Rightarrow_A n_2}{\langle a_1 \geq a_2, \sigma\rangle \Rightarrow_B true} \tag{A.46}$$

falls n_1 gleich n_2 oder größer

$$\frac{\langle a_1, \sigma\rangle \Rightarrow_A n_1 \quad \langle a_2, \sigma\rangle \Rightarrow_A n_2}{\langle a_1 \geq a_2, \sigma\rangle \Rightarrow_B false} \tag{A.47}$$

falls n_1 kleiner als n_2

A.9.4 Greater_Than

Dieser Knoten repräsentiert den Vergleich, ob der Wert, der mit der *Left_Operand*-Kante verwiesen wird, größer ist als der, der an der *Right_Operand*-Kante hängt.

$$\frac{\langle a_1, \sigma\rangle \Rightarrow_A n_1 \quad \langle a_2, \sigma\rangle \Rightarrow_A n_2}{\langle a_1 > a_2, \sigma\rangle \Rightarrow_B true} \tag{A.48}$$

falls n_1 größer als n_2

$$\frac{\langle a_1, \sigma\rangle \Rightarrow_A n_1 \quad \langle a_2, \sigma\rangle \Rightarrow_A n_2}{\langle a_1 > a_2, \sigma\rangle \Rightarrow_B false} \tag{A.49}$$

falls n_1 kleiner als n_2 oder gleich

A.9.5 Less_Or_Equal

Bei diesem Knoten handelt es sich um den arithmetischen Vergleich der beiden Operanden. Ist der linke Operand kleiner oder gleich dem rechten Operanden, so wird `true` zurückgegeben, ansonsten `false`.

$$\frac{\langle a_1, \sigma\rangle \Rightarrow_A n_1 \quad \langle a_2, \sigma\rangle \Rightarrow_A n_2}{\langle a_1 \leq a_2, \sigma\rangle \Rightarrow_B true} \tag{A.50}$$

falls n_1 kleiner als n_2 oder gleich

$$\frac{\langle a_1, \sigma\rangle \Rightarrow_A n_1 \quad \langle a_2, \sigma\rangle \Rightarrow_A n_2}{\langle a_1 \leq a_2, \sigma\rangle \Rightarrow_B false} \tag{A.51}$$

falls n_1 größer als n_2

A.9.6 Less_Than

Bei diesem Knoten handelt es sich um den arithmetischen Vergleich der beiden Operanden. Ist der linke Operand kleiner dem rechten Operanden, so wird `true` zurückgegeben, ansonsten `false`.

$$\frac{\langle a_1, \sigma\rangle \Rightarrow_A n_1 \quad \langle a_2, \sigma\rangle \Rightarrow_A n_2}{\langle a_1 < a_2, \sigma\rangle \Rightarrow_B true} \tag{A.52}$$

falls n_1 kleiner als n_2

$$\frac{\langle a_1, \sigma\rangle \Rightarrow_A n_1 \quad \langle a_2, \sigma\rangle \Rightarrow_A n_2}{\langle a_1 < a_2, \sigma\rangle \Rightarrow_B false} \tag{A.53}$$

falls n_1 größer als n_2 oder gleich

A.9.7 Instanceof_Operator

Dieser Knoten repräsentiert den Test in Ada und Java, ob ein Objekt eine Instanz einer bestimmten Klasse ist.

Obwohl dieser Operand nicht in *C++* existiert, wird der Knoten verwendet, um bei dem Werfen einer Exception zu überprüfen, ob der Typ des jeweiligen Catch-Blocks eine Instanz des selben Typs der geworfenen Exception ist.

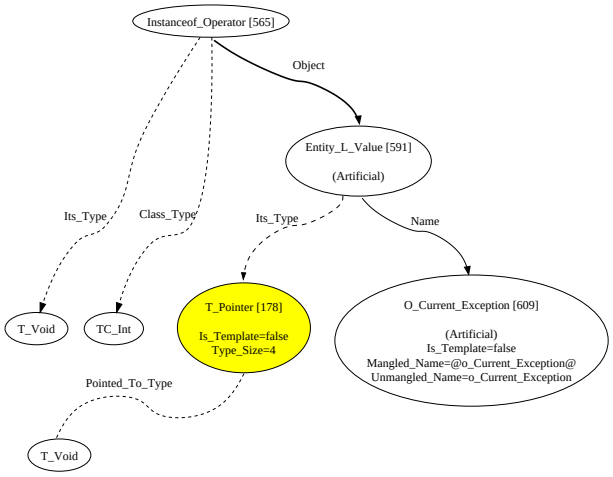

Abbildung A.21: Knoten Instanceof_Operator

$$\langle instanceOf\ x\ y, \sigma\rangle \Rightarrow_A True \tag{A.54}$$

wobei x vom selben oder einem abgeleitetem Typ des Typs von y ist.

$$\langle instanceOf\ x\ y, \sigma\rangle \Rightarrow_A False \tag{A.55}$$

wobei x nicht vom selben oder einem abgeleitetem Typ des Typs von y ist.

A.9.8 Logical_Not

Bei diesem Knoten handelt es sich um die Repräsentation der logischen Negation zur Verwendung in Bedingungen und Kontrollstrukturen. Zur Bestimmung des Werts dieses Knotens wird der Operand falls nötig in einen boolschen Ausdruck aufgelöst und anschließend negiert.

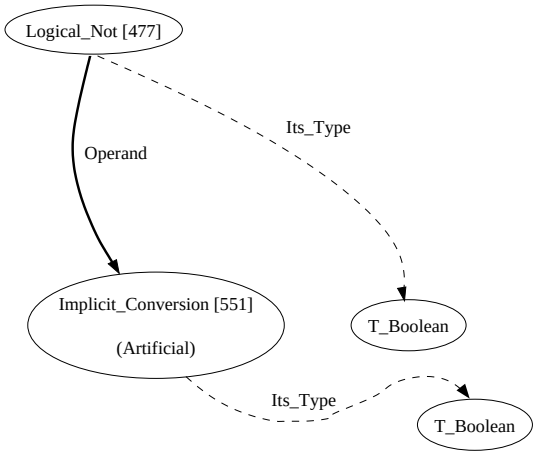

Abbildung A.22: Knoten Logical_Not

$$\frac{\langle b, \sigma\rangle \Rightarrow_B true}{\langle \neg b, \sigma\rangle \Rightarrow_B false} \tag{A.56}$$

$$\frac{\langle b, \sigma\rangle \Rightarrow_B false}{\langle \neg b, \sigma\rangle \Rightarrow_B true} \tag{A.57}$$

A.10 Shift-Operatoren

Die Shift-Operatoren, die durch die folgenden Knoten repräsentiert werden, manipulieren direkt den vom Objekt verwendeten Speicher. Dabei ist es belanglos, was der in dem Speicherbereich liegende Wert darstellt. Definiert sind diese Operationen nur für Integer-Werte, bei denen jedes Bit entsprechend der Operation verschoben wird. Mit den nötigen Cast-Operationen sind diese Shift-Operatoren auf allen Datentypen möglich, auch wenn das Ergebnis nicht definiert ist.

A.10.1 Shift_Left

Dieser Knoten repräsentiert einen Links-Shift. Dies bedeutet, daß alle Bits des Operanden um Stellen entsprechend dem rechten Operanden nach links verschoben werden. Aufgefüllt wird dabei von rechts mit Nullen.

$$\langle n \ll m, \sigma\rangle \Rightarrow_A k \tag{A.58}$$

mit k gleich um m Bit nach links geshiftetem n. Es wird von rechts mit Null-Bits aufgefüllt. Dies entspricht in etwa `n * 2 ** m` ohne Überlauf.

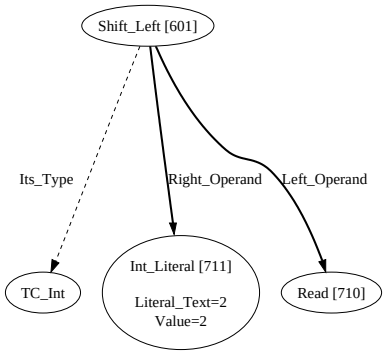

Abbildung A.23: Knoten Shift_Left

A.10.2 Fill_Zero_Shift_Right

Dieser Knoten repräsentiert den logischen Rechts-Shift, bei dem die Bits des Operanden um die gegebene Anzahl an Bits nach rechts verschoben werden. Dabei werden die linken Bits mit Nullen aufgefüllt.

$$\langle n \gg m, \sigma \rangle \Rightarrow_A k \tag{A.59}$$

mit k gleich um m Bit nach rechts geshifteten und mit Nullbits aufgefülltem n. Die Besonderheit des *Sign-Bits* wird hierbei ignoriert und es wird wie alle anderen auch geshiftet.

A.10.3 Shift_Right

Bei diesem Knoten handelt es sich um die Repräsentation des arithmetischen Rechts-Shifts. Im Gegensatz zum logischen Shift wird hier eine Kopie des ersten Bits (das Vorzeichenbit) von links nachgeschoben.

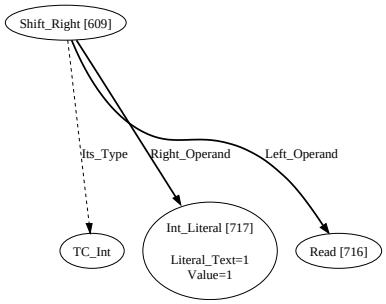

Abbildung A.24: Knoten Shift_Right

$$\langle n \gg m, \sigma \rangle \Rightarrow_A k \tag{A.60}$$

mit k gleich um m Bit nach rechts geshiftetem n. Die auf der linken Seite hinzugekommenen Bits sind dabei Kopien des *Sign-Bits*.

A.11 Pointer

Die folgenden Knoten repräsentieren die Verwendung von *Pointern*. An sich handelt es sich bei *Pointern* um Integer-Werte, die die Adresse des referenzierten Speicherbereichs darstellen. Durch die Verwendung des *Pointer*-Typs sind jedoch die folgenden Operationen möglich.

A.11.1 Dereferenzierung

Die Dereferenzierungsknoten repräsentieren die Auflösung eines Pointers zu seinem eigentlichen Wert. Der Typ der Knoten entspricht dabei dem des Objekts auf das der Pointer verweist.

A.11.1.1 Dereference

Dieser Knoten löst eine Referenz, etwa einen *Pointer*, auf (siehe Kapitel A.11.2.1) und gibt das Element zurück, auf das verwiesen wird. Bei einem *Array* handelt es sich um dessen erstes Element.

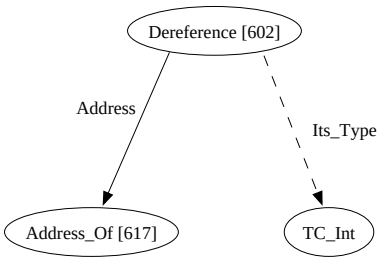

Abbildung A.25: Knoten Dereference

A.11.1.2 Indexed_Dereference

Im Gegensatz zum Dereference-Knoten wird hier nicht das erste Element zurückgegeben, auf das verwiesen wird, sondern das `n`-te Element, wobei `n` der gegebene Index ist. Hauptsächlich findet dies bei dem Verweis auf Elemente innerhalb eines Arrays Verwendung.

Der Interpreter berechnet die Adresse des zu dereferenzierten Elements anhand der Knoten an den Kanten *Base_Address* und *Index*. Hierbei können allerdings Probleme auftreten, wenn unterhalb dieses Knotens Pointerkonvertierungen durchgeführt werden (siehe Kapitel 4.4.1.2).

A.11.2 Referenzierung

Im Gegensatz zu den Dereferenzierungsknoten werden die hier dargestellten Referenzierungsknoten zu den Adressen der jeweiligen Objekte aufgelöst.

A.11.2.1 Address_Of

Dieser Knoten repräsentiert die Ermittlung der Speicheradresse einer Variable. Der Interpreter gibt hier nicht die Systemadresse zurück, bei der die Variable abgelegt ist, sondern den Index des virtuellen Speicherbereichs. So kann leichter kontrolliert werden, ob das Programm innerhalb der Grenzen des eigenen Speicherbereichs arbeitet. Nur beim Zugriff auf

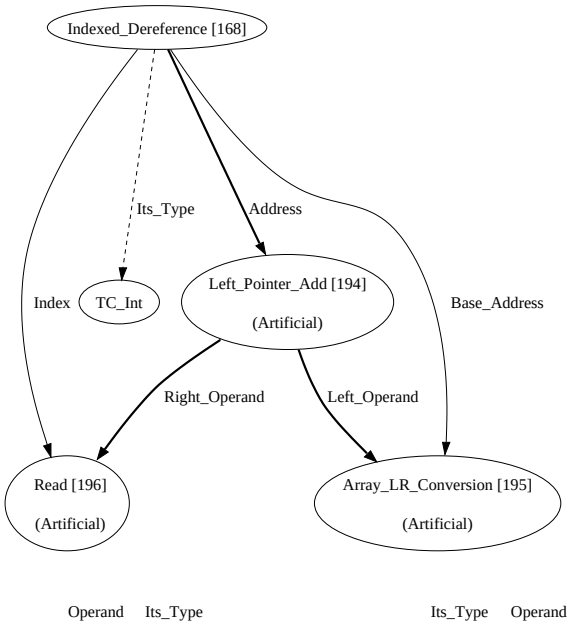

Abbildung A.26: Knoten Indexed_Dereference

externe Funktionen, die Speicheradressen als Parameter erwarten oder zurückgeben, ist eine Konvertierung zwischen internem Speicherindex und Systemadressen nötig.

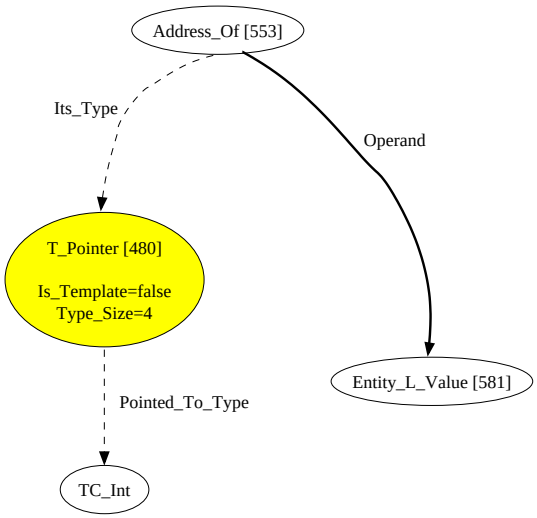

Abbildung A.27: Knoten Address_Of

$$\langle \&n, \sigma \rangle \Rightarrow_A m \tag{A.61}$$

mit m gleich der Speicheradresse von n.

A.11.2.2 Array_LR_Conversion

Dieser Knoten repräsentiert die Erstellung eines *Pointers* auf ein *Array*. Der so erlangte *Pointer* zeigt auf das erste Element des *Arrays*.

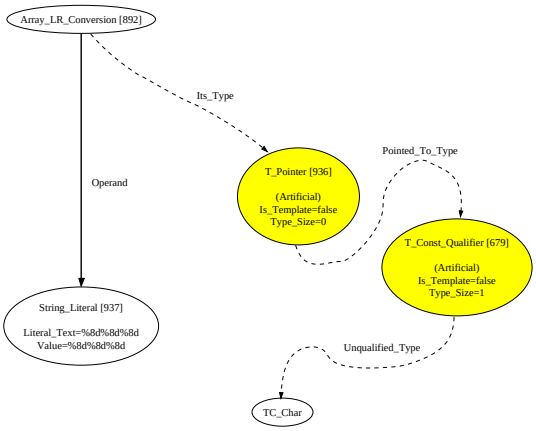

Abbildung A.28: Knoten Array_LR_Conversion

A.11.2.3 Routine_LR_Conversion

Dieser Knoten repräsentiert die Auflösung einer Funktion zu einem *Pointer*. Dieser so erlangte Funktions*pointer* kann daraufhin genutzt werden, um die verwiesene Methode aufzurufen. Wie Funktions*pointer* vom Interpreter behandelt werden, ist dem Kapitel 3.3.3 zu entnehmen.

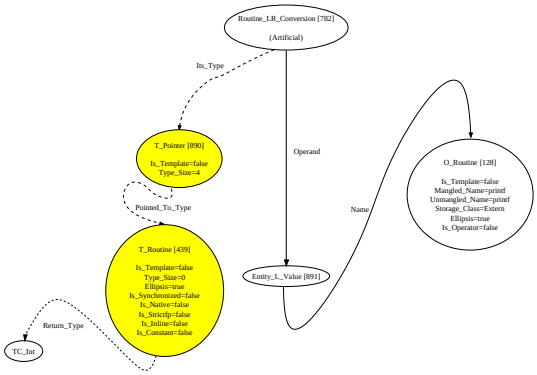

Abbildung A.29: Knoten Routine_LR_Conversion

A.11.3 Arithmetik

Die folgenden Knoten repräsentieren die möglichen Operationen auf Pointerwerte. Hierzu gehören Addition, Subtraktion und das Bilden der Differenz zweier *Pointer*.

A.11.3.1 Pointer_Subtract

Dieser Knoten steht für die Subtraktion von Integern von Systemadressen. Um wieviele Bytes der resultierende *Pointer* von der Ursprungsadresse abweicht, ist abhängig von dem Typ, auf den der *Pointer* verweist. Wenn `n` von dem *Pointer* vom Typ `t` abgezogen wird, liegt der resultierende *Pointer* um `n * sizeof(t)` Byte weiter links im Speicher.

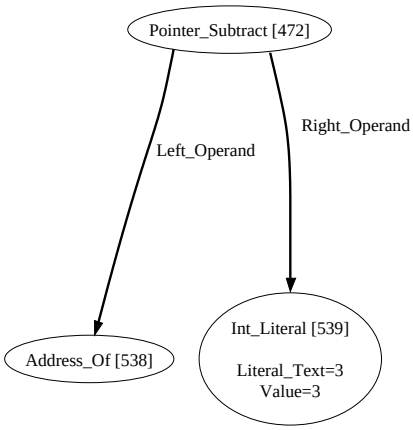

Abbildung A.30: Knoten Pointer_Subtract

A.11.3.2 Left_Pointer_Add

Bei diesem Knoten handelt es sich um die Repräsentation für das Verschieben eines *Pointers*. Die Basisadresse wird anhand des linken Operanden an der *Left_Operand*-Kante bestimmt, während der Index, der bestimmt um wieviel der Pointer verschoben werden soll, über die *Right_Operand*-Kante verwiesen wird.

Der Index wird mit der Größe des Typen multipliziert, auf den der *Pointer* verweist, und entsprechend verschoben. Konvertierungen des *Pointer*typen die eigentlich unterhalb dieses Knotens geschehen müssten, werden bei der Generierung der IML ignoriert, was zu kritischen Fehlern führen kann (siehe Kapitel 4.4.1.2).

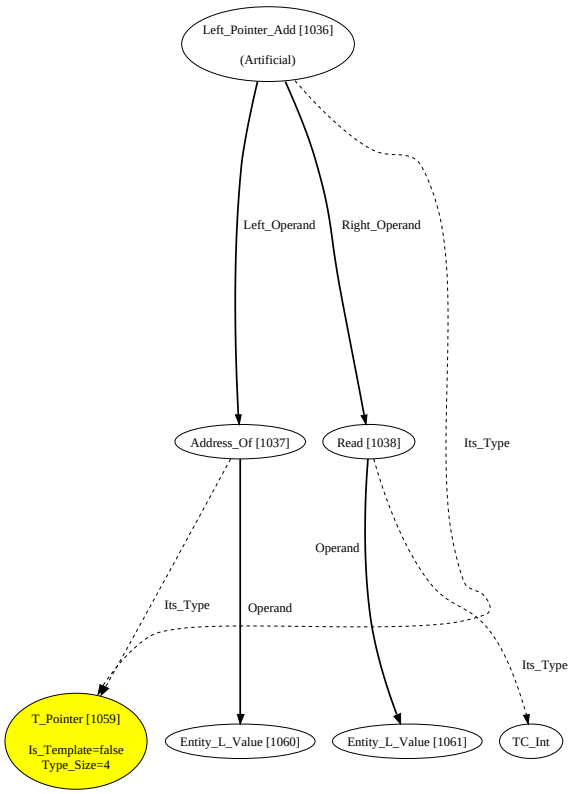

Abbildung A.31: Knoten Left_Pointer_Add

A.11.3.3 Right_Pointer_Add

Dieser Knoten verhält sich äquivalent zu dem Knoten *Left_Pointer_Add*, nur die Operanden sind entsprechend vertauscht.

A.11.3.4 Pointer_Difference

Dieser Knoten ermittelt die Differenz zwischen zwei *Pointern*. Diese müssen auf den gleichen Datentyp verweisen. Das Ergebnis ist der Abstand der beiden Speicheradressen in Bytes, geteilt durch die Größe des Typen auf den verwiesen wird. So läßt sich der Abstand zweier Elemente eines Arrays anhand ihrer *Pointer* ermitteln.

Zum Zeitpunkt der Implementierung ignorierte die mit `cafeCC` generierte IML jegliche Konvertierung der Operanden. Wurden im Quelltext beide Operanden zu einem anderen Typ *Pointer* umgewandelt, so wurde dies ignoriert und der ursprüngliche Typ beibehalten. Dies verfälschte das Ergebnis solcher Berechnungen. Für den Interpreter gibt es keine Möglichkeit dies auszugleichen, daher bleibt auf eine Behebung des Fehlers seitens der IML zu hoffen.

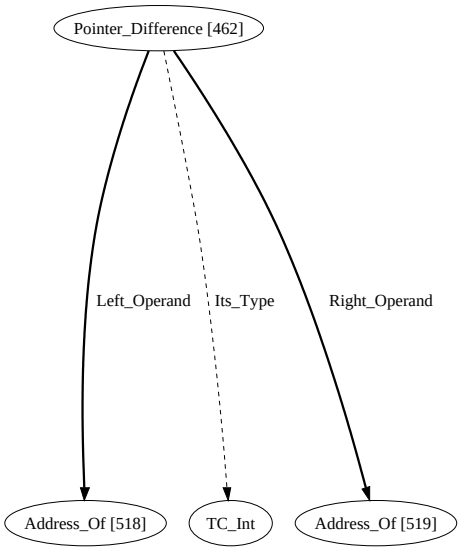

Abbildung A.32: Knoten Pointer_Difference

A.12 Conversion und Casting

Die folgenden Knoten repräsentieren das Konvertieren und Casten von Variablen eines Datentyps in einen anderen.

A.12.1 C_Cast

Bei diesem Knoten handelt es sich um das Konvertieren eines Wertes in eine andere Repräsentation eines anderen Datentyps. Der Wert selbst kann dabei verändert werden.

$$\frac{\langle m_0, \sigma\rangle \Rightarrow_C m_1}{\langle (t)m_0, \sigma\rangle \Rightarrow_A m_2} \tag{A.62}$$

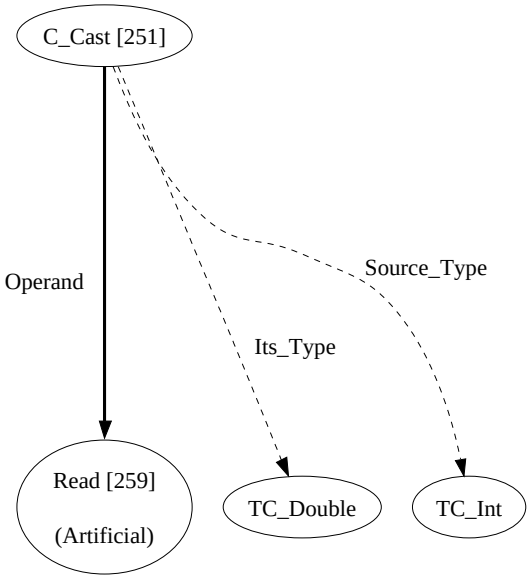

Abbildung A.33: Knoten C_Cast

mit m_2 vom Typ t.

A.12.2 Dynamic_Cast

Knoten vom Typ *Dynamic_Cast* repräsentieren das Casten mittels `dynamic_cast`, wie es in C++ möglich ist. Dieser spezielle Aufruf wird bei der Umwandlung vom Quellcode in IML jedoch ignoriert und als ein simpler *C_Cast*-Knoten abgebildet. Eine Interpretation dieses Knotens erfolgt daher nicht.

A.12.3 Static_Cast

Für diesen Knoten gilt das gleiche wie für den *Dynamic_Cast*-Knoten, eine Abbildung innerhalb der IML erfolgt nicht. Daher ist eine Interpretation des Knotens nicht nötig.

A.12.4 Reinterpret_Cast

Für diesen Knoten gilt das gleiche wie für den *Dynamic_Cast*-Knoten, eine Abbildung innerhalb der IML erfolgt nicht. Daher ist eine Interpretation des Knotens nicht nötig.

A.12.5 Const_Cast

Für diesen Knoten gilt das gleiche wie für den *Dynamic_Cast*-Knoten, eine Abbildung innerhalb der IML erfolgt nicht. Daher ist eine Interpretation des Knotens nicht nötig.

A.12.6 Implicit_Conversion

Dieser Knoten repräsentiert Konvertierungen von Datentypen, die nicht explizit im Quellcode angegeben sind und durchgeführt werden müssen, wenn eine Operationen mit zwei

unterschiedlichen Datentypen durchgeführt werden muss, die für nur einen bestimmten Typen definiert ist. Nach welchen Regeln diese Konvertierungen durchgeführt werden, ist dem Kapitel 3.4.3 zu entnehmen.

Innerhalb der IML werden diese Konvertierungen zum Großteil nicht abgebildet, weswegen dies der Interpreter anhand der jeweiligen Typkanten der einzelnen Knoten durchführen muss. Genauer ist dieses Problem in Kapitel 4.4.1.1 beschrieben.

A.12.7 Explicit_Conversion

Bei diesem Knoten handelt es sich um eine explizit im Quellcode angegebene Konvertierung. Auch diese Knoten werden innerhalb der IML nicht dargestellt.

A.13 Unäre Operatoren

Unter unären Operatoren sind jene zu verstehen, die nur einen Operanden besitzen. Im Folgenden sind die beiden für primitive Datentypen vordefinierten Operatoren aufgeführt. Mittels Operatorenüberladung, wie sie in C und C++ möglich ist, lassen sich weitere Operatoren für komplerexe Datentypen wie Klassen definieren. Diese werden dann durch die entsprechenden Funktions-Knoten (siehe Kapitel A.3) und nicht mehr durch unäre Operatoren repräsentiert.

A.13.1 Unary_Plus

Dieser Operator gibt eine Kopie des Operanden zurück, die den exakt gleichen Wert und Typ wie das Original besitzt. Relevant ist dieser Operator hauptsächlich, wenn er von einer Klasse überladen wird.

$$\langle +n, \sigma \rangle \Rightarrow_A m \tag{A.63}$$

wobei m gleich n ist.

A.13.2 Unary_Minus

Dieser Operator gibt eine Kopie des Operanden zurück, die den negierten Wert des Originals besitzt und dessen Typ dabei beibehält. Vorzeichenlose Typen werden dabei von ihrem maximalen Wert abgezogen. `-u_int8` entspricht also `(2^8-1) - u_int8`.

$$\frac{\langle n * -1, \sigma \rangle \Rightarrow_A m}{\langle -n, \sigma \rangle \Rightarrow_A m} \tag{A.64}$$

A.14 Speicherverwaltung

Die folgenden Knoten repräsentieren die manuelle Speicherverwaltung, wie sie zum Beispiel in C++ möglich ist.

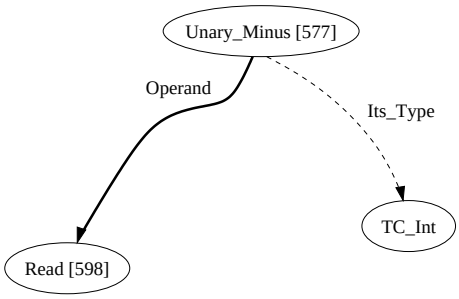

Abbildung A.34: Knoten Unary_Minus

A.14.1 New_Operator

Der `new`-Operator erzeugt eine neue Instanz einer Klasse oder eines primitiven Datentyps und gibt einen *Pointer* auf den entsprechenden Speicherbereich zurück. Handelt es sich bei dem zu erstellenden Objekt um eine Klasse, so wird der dazugehörige Konstruktor aufgerufen.

Die Reservierung des Speichers selbst ist in der IML nicht abgebildet, daher muss die benötigte Größe aus dem Typknoten ausgelesen und der entsprechende Speicherbereich angelegt werden. Erst nach diesem Schritt, kann der Konstruktor auf das neue Objekt angewendet werden.

Soll ein Array neu angelegt werden, so wird dies in der IML in einer `for`-Schleife abgebildet, in der die einzelnen Elemente des *Arrays* initialisiert werden. Handelt es sich bei diesen Elementen um Klassen, so werden die jeweiligen Standardkonstruktoren aufgerufen.

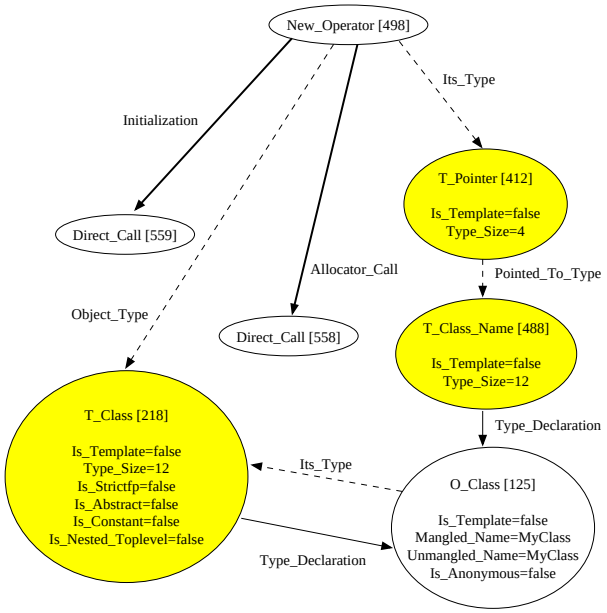

Abbildung A.35: Knoten New_Operator

$$\langle new\ Y(), \sigma \rangle \Rightarrow_C \langle n.Y(), \sigma \cup n \rangle \tag{A.65}$$

mit n vom Typ Y.

A.14.2 Delete_Operator

Dieser Knoten steht für die `delete`-Anweisung, wie sie in C++ existiert. Der Knoten weist den Interpreter an, die Variable, auf den der übergebene *Pointer* verweist, aus dem Speicher zu entfernen bzw. diesen wieder freizugeben. Handelt es sich bei dem zu löschenden Objekt um die Instanz einer Klasse, so wird dessen Destruktor aufgerufen, bevor es gelöscht wird.

A.15 Kontrollstrukturen

Die folgenden Knoten repräsentieren die bedingte Ausführung von Anweisungen. Grundsätzlich bestehen diese Knoten aus einer *Condition*, die zu einem booleschen Wert aufgelöst wird und entsprechend die Anweisung ausgeführt wird, auf die mit dem *Then_Branch* beziehungsweise *Else_Branch* verwiesen wird.

A.15.1 Conditional_Operator

Der *Conditional_Operator* wird durch diesen Knoten repräsentiert. Wird die *Condition* vom Interpreter zu `true` aufgelöst, so ist der Wert des Knoten gleich dem des *Then_Branch*, ansonsten dem des *Else_Branch*. Handelt es sich bei den Branches nicht um Literale, so werden diese Knoten entsprechend aufgelöst, was auch weitere Ausführungen zur Folge haben kann.

Der Typ des Knoten entspricht dabei dem arithmetisch höherwertigen Typ (siehe Kapitel 3.4.3) der beiden Branches. Ist zum Beispiel einer der Branches ein `int` und der andere ein `float`, so ist das Ergebnis des Operators in jedem Fall vom Typ `float`.

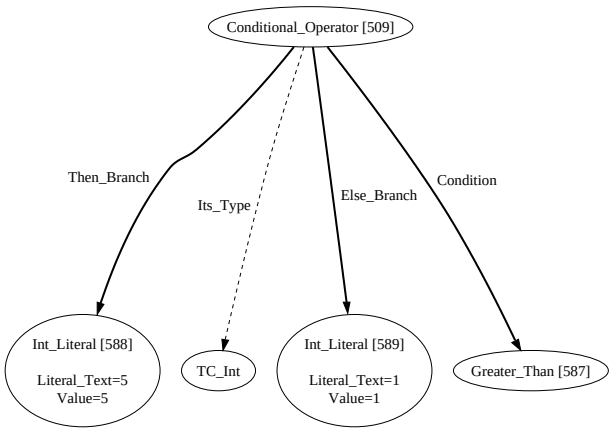

Abbildung A.36: Knoten Conditional_Operator

$$\langle (b\ ?\ c_0\ :\ c_1), \sigma \rangle \Rightarrow_C \langle if\ b\ then\ c_0\ else\ c_1, \sigma \rangle \tag{A.66}$$

A.15.2 If_Statement

Bei diesem Knoten handelt es sich um die Repräsentation der Kontrollstruktur `if`. Der Interpreter löst zuerst die *Condition* auf und führt entsprechend dem Ergebnis entweder den *Then_Branch* oder den *Else_Branch* aus. Beinhaltet der *Else_Branch* eine weitere If-Abfrage

(siehe Listing A.1), so wird in der IML ein weiterer *If_Statement*-Knoten an den *Statement_Sequence*-Knoten unter dem *Else_Branch* gehängt. Entsprechend wird solch ein Fall vom Interpreter bearbeitet.

Listing A.1: Codebeispiel für den *If_Statement*-Knoten

```
1  if (...){
2    ...
3  }else if (...){
4    ...
5  }else{
6    ...
7  }
```

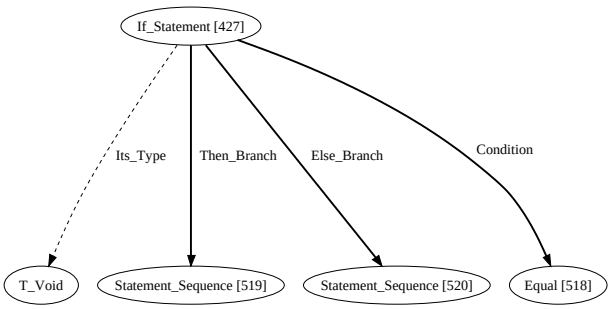

Abbildung A.37: Knoten If_Statement

$$\frac{\langle b, \sigma_0\rangle \Rightarrow_B true \quad \langle c_1, \sigma_0\rangle \Rightarrow_C \sigma_1}{\langle if\ b\ then\ c_1\ else\ c_2, \sigma_0\rangle \Rightarrow_C \sigma_1} \tag{A.67}$$

$$\frac{\langle b, \sigma_0\rangle \Rightarrow_B false \quad \langle c_2, \sigma_0\rangle \Rightarrow_C \sigma_1}{\langle if\ b\ then\ c_1\ else\ c_2, \sigma_0\rangle \Rightarrow_C \sigma_1} \tag{A.68}$$

A.15.3 And_Then

Dieser Knoten repräsentiert die Verknüpfung von Bedingungen, bei der die zweite Bedingung nur dann geprüft wird, wenn die erste erfüllt ist. In C, C++ und Java entspricht dies `a && b`, in Ada `a and then b`.

A.15.4 Or_Else

Dieser Knoten repräsentiert die bedingte Veroderung, bei der die zweite Bedingung nur geprüft wird, wenn die erste unwahr ist. In C, C++ und Java entspricht dies `a || b`, in Ada `a or else b`.

A.15.5 C_Switch_Statement

Das Switch-Case-Konstrukt bietet die Möglichkeit einen Ausdruck einmalig auszuwerten und diesen mit verschiedenen Werten zu vergleichen und entsprechende Anweisungen auszuführen. Als Besonderheit im Vergleich zu Switch-Anweisungen in anderen Sprachen ist in C zu nennen, daß falls ein Fall eintritt, alle unterliegenden Anweisungen ausgeführt werden, bis auf ein `break` gestoßen wird. Bei anderen Sprachen, zum Beispiel Java und Ada, werden die Anweisungen unterliegender `case`-Branches ignoriert.

Der Interpreter wertet zuerst den *Selector* des *C_Switch_Statement*-Knotens aus und speichert das Ergebnis zwischen. Danach werden die einzelnen *Case_Branch*-Knoten, die an der *Dispatch*-Kante hängen, ausgewertet. Ist die Bedingung einer dieser Knoten erfüllt, so wird das *Target* des entsprechenden *Case_Goto* ausgewertet und daraufhin dieses in den einzelnen Statements der Sequenz gesucht, die an der *Switch_Body*-Kante des Switch-Knotens hängt. Alle auf dieses anonyme Label folgenden Knoten werden wie gewohnt ausgeführt.

Stößt der Interpreter während der Ausführung des *Switch_Body*s auf einen Knoten vom Typ *Exit_Switch* (was in C++ einem `break` gleicht), so werden alle folgenden Knoten ignoriert, bis der Interpreter wieder zu dem ursprünglichen *C_Switch_Statement*-Knoten zurückkehrt.

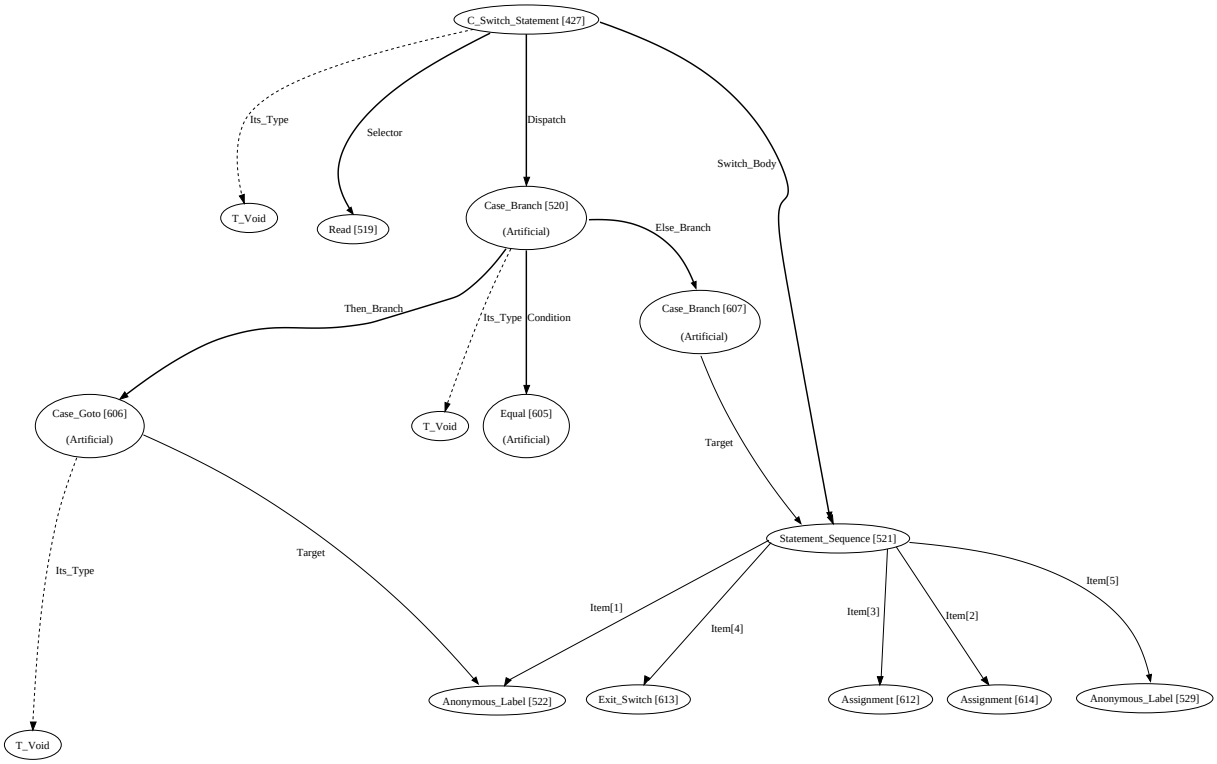

Abbildung A.38: Knoten C_Switch_Statement

A.15.6 Case_Branch

Bei diesen Knoten handelt es sich um einen Unterknoten des *C_Switch_Statement*s. Wie dieser Knoten behandelt wird, ist dem vorherigen Kapitel zu entnehmen.

A.15.7 Case_Goto

Dieser Knoten steht für einen Verweis innerhalb einer Switch-Anweisung zu einem `case`-Zweig. Der Aufbau einer Switch-Anweisung in C ist Kapitel A.15.5 zu entnehmen. Ist eine case-Bedingung erfüllt, so folgt in der Ausführung dieser Knoten, der auf ein *Valued_Label* oder *Anonymous_Label* innerhalb des *switch*-Bodys verweist.

A.15.8 Assert

Bei Assertion handelt es sich um Zusicherungen an das Programm zur Laufzeit. Ist die in der Assertion angegebene Bedingung nicht wahr, so wird die Ausführung des Programms mit entsprechendem Hinweis terminiert.

In aus C generierter IML entspricht dies dem Aufruf der C-Funktion `__assert_fail` und dem Kopieren der jeweiligen Parameter. Zu diesen gehört sowohl der Name der Funktion, in der die Assertion auftrat, der Name der Quelldatei und die jeweilige Zeilennummer.

$$\langle assert(b), \sigma\rangle \Rightarrow_C \langle if\ (b)\ then\ null\ else\ exit(), \sigma\rangle \tag{A.69}$$

A.16 Schleifen

Die Schleifenkonstrukte gleichen sich zum größten Teil. Sie bestehen aus einer Laufbedingung, einem Schleifenrumpf, der bei jeder Iteration ausgeführt wird und eventuell um eine Aktion, die nach jedem erfolgreichen Durchlauf ausgeführt wird. Nur was zu welchem Zeitpunkt geschieht, unterscheidet diese Schleifen und ob anfangs noch Variablen deklariert werden, die nur innerhalb der Schleife selbst gültig sind.

A.16.1 C_For_Loop

Dieser Knoten repräsentiert die For-Schleife, wie sie in C und C++ zu finden ist. Bei der Ausführung durch den Interpreter wird zunächst die Initialisierung durchgeführt, indem der Knoten ausgeführt wird, der an der *Initialization*-Kante hängt. Anschließend wird die Sequenz an der *Loop_Kernel*-Kante so lange ausgeführt, bis auf einen *Exit_Loop*-Knoten gestoßen wird.

Die einzelnen Elemente der Schleife, können über die Kanten *Loop_Body* (der eigentliche Inhalt der Schleife), *Continue_Expression* (die Anweisung die nach jedem Durchlauf ausgeführt wird) und *Initialization* (die Anweisung, die nur beim ersten Aufruf der Schleife ausgeführt wird) und *Condition* (die Ausführungsbedingung) angesprochen werden.

Da alle relevanten Anweisungen, die bei jedem Durchlauf ausgeführt werden müssen, ebenfalls im *Loop_Kernel* zu finden sind, muss dieser ausgeführt werden. Der erste Knoten unter der *Statement_Sequenz* ist die Ausführungsbedingung, gefolgt von dem Körper der Schleife und letztendlich der *Continue_Expression*, was in den meisten Fällen die Inkrementierung eines Zählers ist.

Trift der Interpreter innerhalb einer `for`-Schleife auf eine `continue`-Anweisung, werden die restlichen Knoten innerhalb des Schleifenkörpers übersprungen. Anschließend wird die *Continue_Expression* explizit aufgerufen, da sie sonst ebenfalls übersprungen wird.

$$\frac{\langle c_1, \sigma_0\rangle \Rightarrow_C \sigma_1 \quad \langle b, \sigma_1\rangle \Rightarrow_B true \quad \langle c_3; c_2, \sigma_1\rangle \Rightarrow_C \sigma_2}{\langle for(c_1; b; c_2)c_3, \sigma_0\rangle \Rightarrow_C \langle for(null; b; c_2)c_3, \sigma_2\rangle} \tag{A.70}$$

$$\frac{\langle c_1, \sigma_0\rangle \Rightarrow_C \sigma_1 \quad \langle b, \sigma_1\rangle \Rightarrow_B false}{\langle for(c_1; b; c_2)c_3, \sigma_0\rangle \Rightarrow_C \sigma_1} \tag{A.71}$$

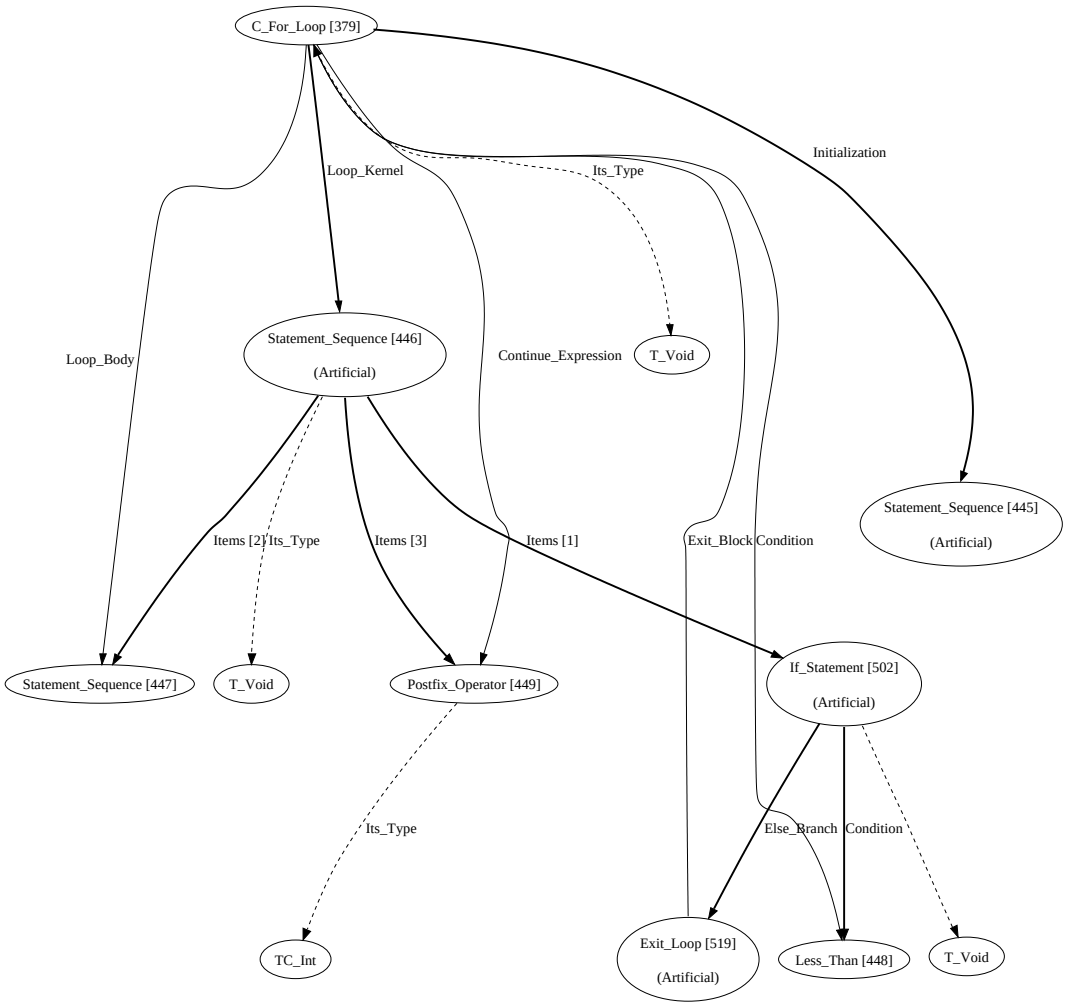

Abbildung A.39: Knoten C_For_Loop

A.16.2 Do_While_Loop

Dieser Knoten repräsentiert das `Do...while(...)`-Konstrukt, das in vielen Sprachen vertreten ist. Die Schleife wird solange durchlaufen, solange die angegebene Bedingung erfüllt ist. Die Schleife wird dabei mindestens einmal ausgeführt.

Die *Loop_Kernel*-Kante der Schleife verweist auf eine *Statement_Sequence*, die sowohl den Rumpf der Schleife als auch die Bedingung selbst beinhaltet. So muss der Interpreter diesen *Kernel* solange ausführen, bis er auf einen *Exit_Loop*-Knoten stößt, wie er beim *Else_Branch* der *Condition* verlinkt ist.

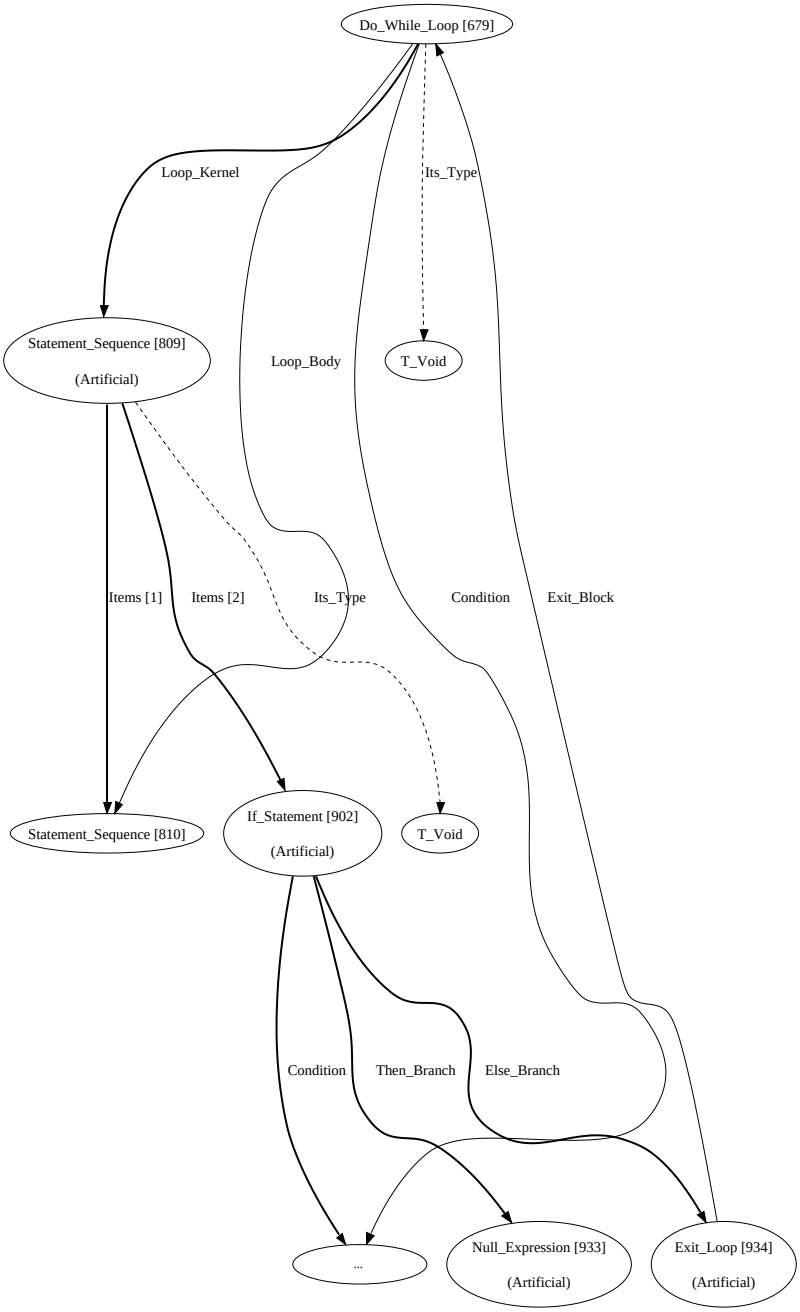

Abbildung A.40: Knoten Do_While_Loop

$$\frac{\langle c, \sigma_0\rangle \Rightarrow_C \sigma_1 \quad \langle while\ b\ do\ c, \sigma_1\rangle \Rightarrow_C \sigma_2}{\langle do\ c\ while\ b, \sigma_0\rangle \Rightarrow_C \sigma_2} \tag{A.72}$$

A.16.3 While_Loop

Bei diesem Knoten handelt es sich um die Repräsentation einer While-Schleife. Ihr Rumpf wird solange und nur dann ausgeführt, solange die gegebene Bedingung erfüllt ist. Dies bedeutet daß wenn die Bedingung beim ersten Aufruf nicht erfüllt ist, keine Ausführung des Rumpfes erfolgt.

Der Aufbau des Knotens ist im Grunde identisch mit dem der *Do_While*-Schleife, abgesehen davon das die Bedingung für die weitere Ausführung der erste Unterknoten des *Loop_Kernel*s ist. Daher wird diese Bedingung vor jeder Ausführung des *Loop_Body*s geprüft. Nach dem Erreichen eines *Exit_Loop*-Knotens wird jede weitere Ausführung übersprungen, bis der Interpreter zurück zu dem *While_Loop*-Knoten springt.

$$\frac{\langle b, \sigma\rangle \Rightarrow_B false}{\langle while\ b\ do\ c, \sigma\rangle \Rightarrow_C \sigma} \tag{A.73}$$

$$\frac{\langle b, \sigma_0\rangle \Rightarrow_B true \quad \langle c, \sigma_0\rangle \Rightarrow_C \sigma_1 \quad \langle while\ b\ do\ c, \sigma_1\rangle \Rightarrow_C \sigma_2}{\langle while\ b\ do\ c, \sigma_0\rangle \Rightarrow_C \sigma_2} \tag{A.74}$$

A.16.4 Continue_Goto

Dieser Knoten steht für die `continue`-Anweisung, wie man sie in Java, C und C++ innerhalb von Schleifen findet. Stößt der Interpreter auf diesen Knoten, so wird die Ausführung der weiteren Knoten der Schleife unterbunden und mit der *Continue_Expression* fortgefahren.

A.16.5 Exit_Loop

Dieser Knoten steht für das Verlassen einer Schleife. Dieser wird sowohl verwendet, wenn ein `break` innerhalb einer Schleife steht, als auch wenn die Kontrollbedingung nicht mehr erfüllt ist.

Trifft der Interpreter auf solch einen Knoten, so werden alle folgenden Knoten ignoriert, bis er zu dem ursprünglichen Schleifen-Knoten zurückkehrt. Ein direkter Sprung zu dem Ende der Schleife innerhalb des Graphen ist nicht möglich, da der Interpreter rekursiv durch die Knoten läuft.

Abgesehen von der Markierung dieses Zustandes erfolgt keine weitere Ausführung.

A.17 Exceptionbehandlung

Die folgenden Knoten dienen zur Behandlung von Exceptions, wie sie in C++, Java und Ada auftreten können. Wie Exceptions vom Interpreter behandelt werden, steht in Kapitel 3.4.5 beschrieben.

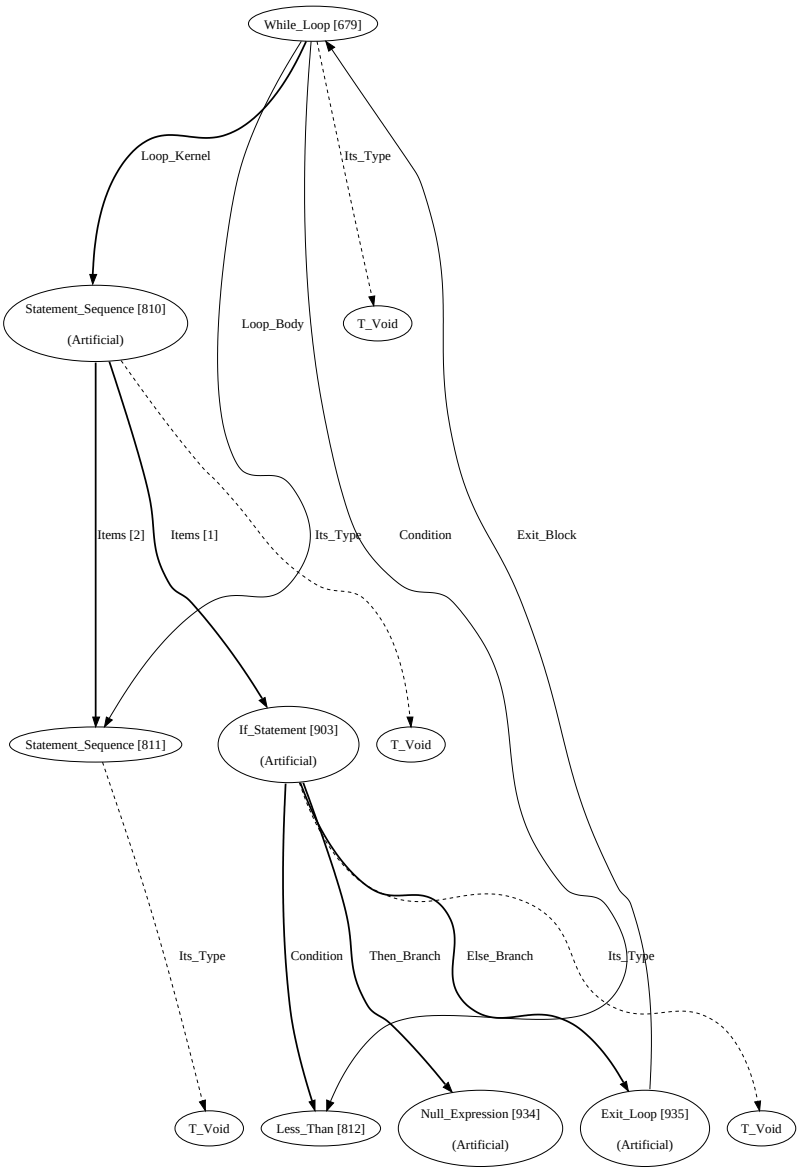

Abbildung A.41: Knoten While_Loop

A.17.1 Try_Catch_Finally_Statement

Dieser Knoten steht für die Behandlung von Ausnahmen, wie sie sowohl in Java als auch in C++ zu finden sind, wobei der *Finally*-Teil nur in Java verwendet werden kann.

Wird eine Exception von einer der Anweisungen innerhalb des Try-Blocks geworfen, so wird versucht diese mit einer der Catch-Blöcke zu fangen. Sollte dies nicht gelingen, so wird die Exception weitergereicht an den nächsten umgebenden Block. Existiert kein umgebender `try/catch`-Block, so wird die Ausführung des Programms beendet.

Unabhängig davon, ob die Exception abgefangen wird oder nicht, werden noch die Anweisungen im jeweiligen *Finally*-Block ausgeführt.

In Java bezieht sich der Test, ob eine Exception von einem der Catch-Blöcke abgefangen wird immer auf den Typ der Exception. Jede Exception in Java ist abgeleitet von der Basisklasse *Exception*. Anders ist es in C++, wo jeder Datentyp als Exception dienen kann und entsprechend auf diesen geprüft wird. So können sowohl `int`s als auch `float`s geworfen werden und auf die entsprechenden Typen in den Bedingungen der Catch-Blöcke getestet werden. Das Beispiel in Abbildung A.42 hat als Datentyp für die abgefangene Exception *TC_Int* und würde somit jeden geworfenen Integer abfangen.

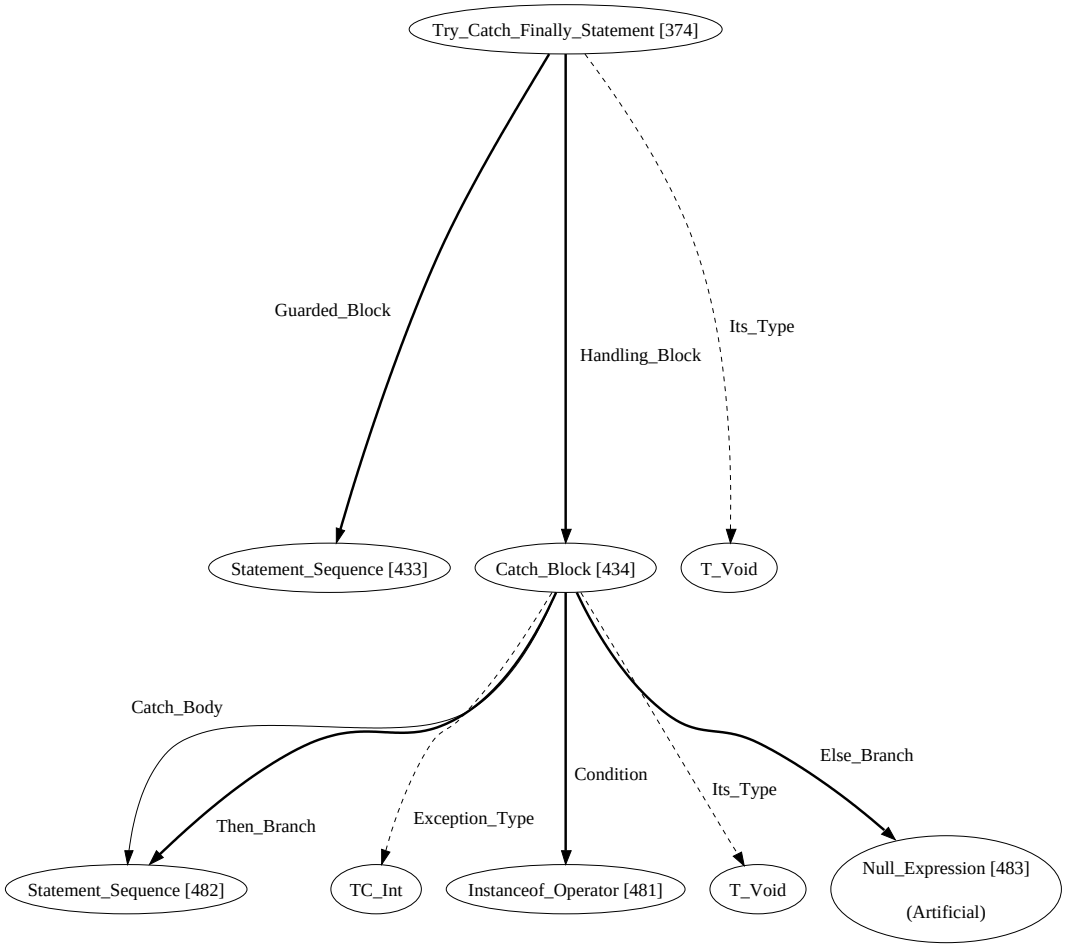

Abbildung A.42: Knoten Try_Catch_Finally_Statement

A.17.2 Catch_Block

Der Catch-Block ist ein Unterknoten des *Try_Catch_Finally*-Knotens und besteht aus einer Bedingung und den jeweiligen Zweigen die in den entsprechenden Fällen ausgeführt werden. Die Bedingung ist die Überprüfung ob eine Exception geworfen wurde beziehungsweise ob diese von diesem Block behandelt wird. Dies geschieht mit Hilfe des *Instanceof_Operator*-Knotens, der zwei Typen vergleicht. Der Aufbau des Knotens ist der Abbildung A.42 zu entnehmen.

A.17.3 Throw_Statement

Dieser Knoten steht für das Werfen einer Exception. Trifft der Interpreter auf einen Knoten dieses Typs, so wird nach dem Erstellen der entsprechenden Exception die Ausführung der folgenden Anweisungen übersprungen, bis er auf einen Catch-Block trifft, der die geworfene Exception abfängt. Wird solch ein Catch-Block nicht gefunden, wird die Ausführung des Programms abgebrochen.

Die Beschaffenheit von Exceptions und wie sie in den einzelnen Sprachen behandelt werden, erfordert eine besondere Behandlung bezüglich der Abbildung im Speicher. Wie der Interpreter genau mit Exceptions verfährt, ist in Kapitel 3.4.5 aufgeführt.

A.17.4 Cpp_Throw_Statement

Dieser Knoten repräsentiert den Wurf einer Exception in C++. Im Gegensatz zu Java gibt es bei C++ keine speziellen Exception-Klassen, geworfen werden kann jeder beliebige Datentyp. Das neu erstellte Objekt, beziehungsweise der Pointer darauf, wird im Objekt *O_Current_Exception* abgelegt. Wie die Exceptions vom Interpreter behandelt werden, ist in Kapitel 3.4.5 aufgeführt.

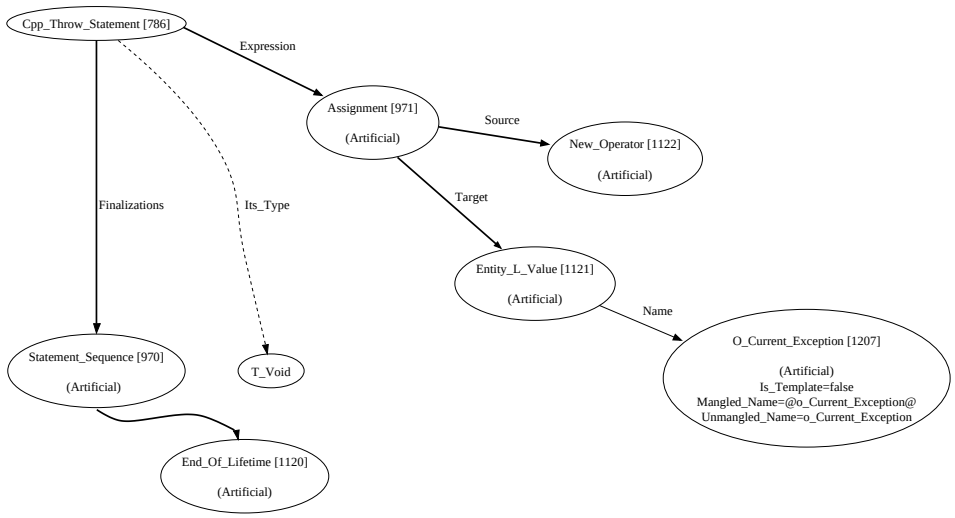

Abbildung A.43: Knoten Cpp_Throw_Statement

A.18 Sequenzen

Die folgenden Knoten repräsentieren unterschiedliche Varianten von Knoten-Sequenzen. Sie werden nacheinander ausgeführt, solange bis die Sequenz endet oder bis zum ersten Abbruchknoten. Zu solchen gehören unter anderem die `return`-Statements der umgebenden Funktion sowie auch die `continue`-Anweisungen innerhalb von Schleifen.

A.18.1 Statement_Sequence

Bei diesem Knoten handelt es sich um eine Kette von Anweisungen, die zu einer Sequenz zusammengefasst wurden. Der erste Knoten unterhalb dieser Sequenz ist ein *Begin_Of_Lifetime*-Knoten, während der letzte entsprechend ein *End_Of_Lifetime*-Knoten ist, die für die Initialisierung und Entfernung von Variablen zuständig sind, deren Sichtbarkeit in dieser Sequenz beginnt beziehungsweise danach endet.

Solch eine Sequenz ist Bestandteil einer jeden Funktion, bei denen sie den eigentlichen Funktionsrumpf darstellt, und auch von Schleifen und Blöcken. Die Namen der Variablen die nur innerhalb dieser Blöcke gültig sind, können Variablennamen aus dem umgebenden Bereich überdecken. Dies stellt im Interpreter kein Problem dar, da die Variablen nicht anhand ihres Namen, sondern anhand des eindeutigen IML-Indizes identifiziert werden.

Die einzelnen Unterknoten der Sequenz werden nacheinander ausgeführt. Vor jeder Ausführung eines Knoten wird geprüft, ob eine Abbruchbedingung erfüllt wurde, in welchem Fall er übersprungen wird. Zu diesen Abbruchbedingungen gehört das Treffen auf eine *Continue_Expression* innerhalb einer Schleife oder einem `abort`. Auch wenn während der Ausführung der Sequenz auf ein `return`-Statement getroffen wird, werden die restlichen Knoten übersprungen, so daß der Interpreter zu der aufrufenden Funktion zurückkehren kann.

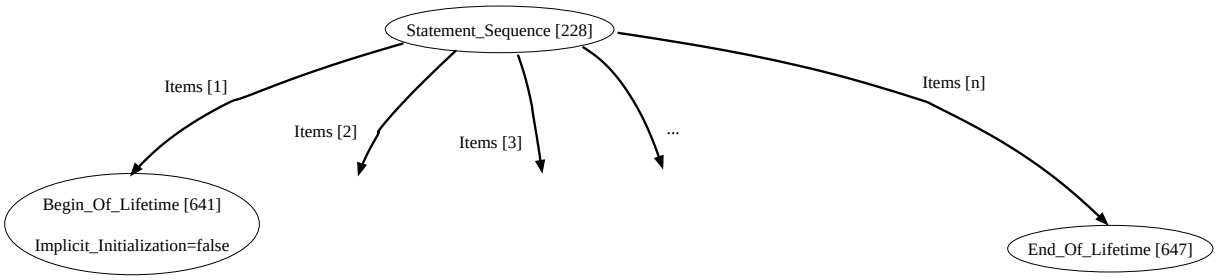

Abbildung A.44: Knoten Statement_Sequence

$$\frac{\langle c_1, \sigma_0\rangle \Rightarrow_C \sigma_1 \quad \langle c_2, \sigma_1\rangle \Rightarrow_C \sigma_2}{\langle c_1; c_2, \sigma_0\rangle \Rightarrow_C \sigma_2} \tag{A.75}$$

A.18.2 Synchronized_Sequence

Dieser Knoten repräsentiert einen `synchronize`-Block, wie er in Java existiert. Dieser stellt sicher, daß nur ein Thread zur Zeit auf ein bestimmtes Objekt zugreifen kann. Da weder Java noch Threads vom Interpreter unterstützt werden, wird dieser Knoten nicht interpretiert.

A.18.3 C_Comma_Sequence

Bei dieser Sequenz werden die aufgeführten Operationen hintereinander ausgeführt. Der Wert und der Typ zu dem die Sequenz selbst aufgelöst wird, entspricht dem der letzten Anweisung in der Sequenz.

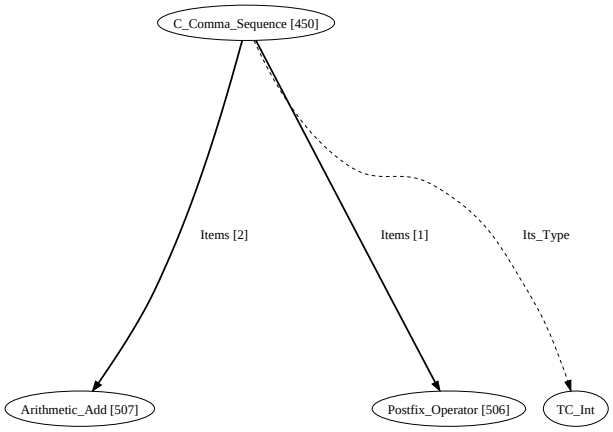

Abbildung A.45: Knoten C_Comma_Sequence

$$\frac{\langle c_1, \sigma_0\rangle \Rightarrow_C \sigma_1}{\langle (c_1, c_2), \sigma_0\rangle \Rightarrow_A \langle c_2, \sigma_1\rangle} \tag{A.76}$$

A.19 Sonstige Knoten

Die folgenden Knoten passen in keine der anderen Kategorien und sind daher in diesem Abschnitt separat aufgeführt.

A.19.1 Expression_Sizeof

Dieser Knoten ermittelt die Größe eines *LValues* oder auch *RValues* in Byte und gibt diesen Wert zurück. In C entspricht dies zum Beispiel `sizeof(4)` und in Ada `My_Int'Size`.

A.19.2 Type_Sizeof

Dieser Knoten ermittelt die Größe eines Datentyps in Byte und gibt diesen Wert zurück. In C entspricht dies zum Beispiel `sizeof(int)` und in Ada `Integer'Size`.

A.19.3 Goto_Statement

Ein *Goto_Statement* besteht aus zwei Verbindungen zu weiteren Knoten innerhalb des Graphen. Die erste ist ein Link zu der Finalisierung, dem *End_Of_Lifetime*-Knoten, der alle Variablen entfernt, deren Sichtbarkeit durch die Goto-Anweisung verlassen wird. Die zweite Verbindung verweist auf das entsprechende *Named_Label*, das an beliebiger Stelle im Code stehen kann.

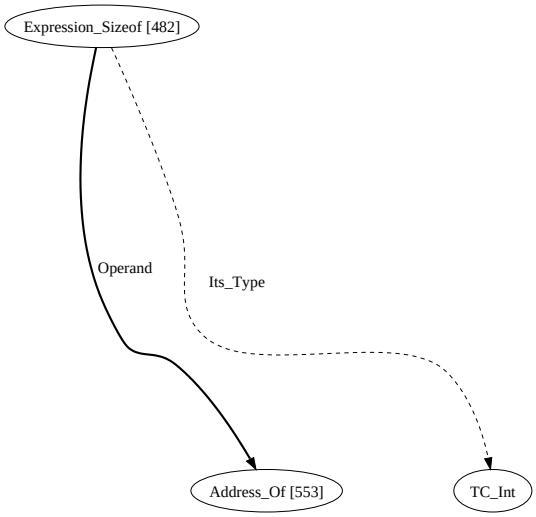

Abbildung A.46: Knoten Expression_Sizeof

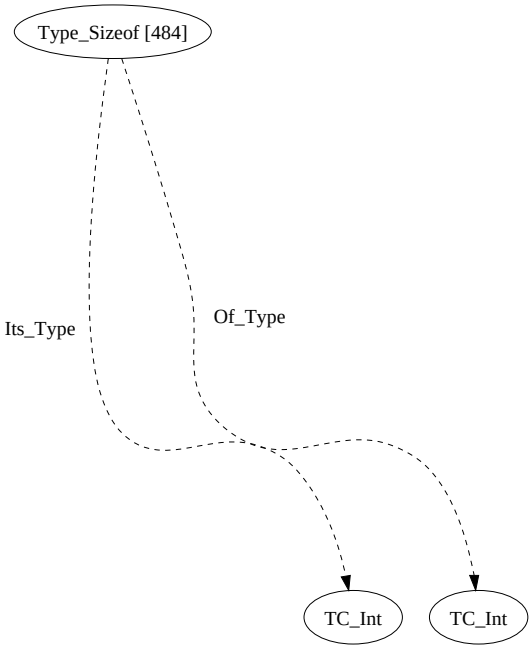

Abbildung A.47: Knoten Type_Sizeof

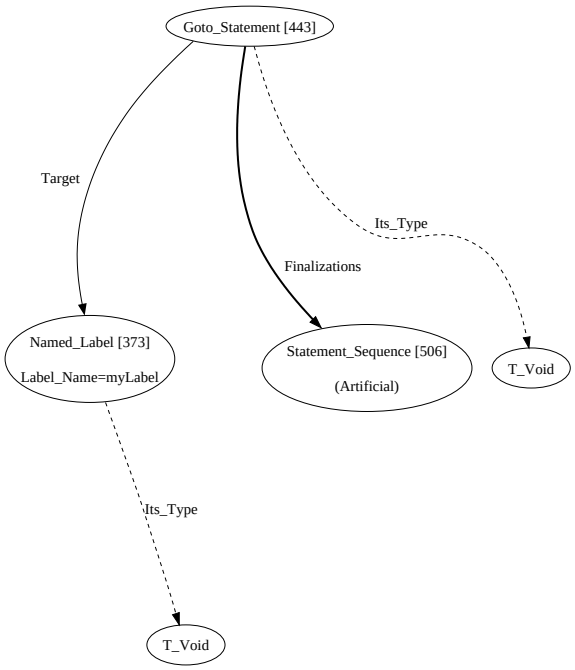

Abbildung A.48: Knoten Goto_Statement

Um dies zu realisieren sucht der Interpreter, wenn er auf ein Goto stößt, das entsprechende Label und die über diesem stehende *Statement_Sequence* heraus. Dann führt er alle Anweisungen der Sequenz, die nach dem Label folgen, normal aus. Wurden alle Anweisungen ausgeführt, das Programm also an sich beendet, springt der Interpreter zurück zu der letzten *Goto*-Anweisungen und markiert das Programm als beendet. So werden alle folgenden Anweisungen im IML-Graphen übersprungen. Dies ist notwendig, da der Graph rekursiv vom Interpreter durchlaufen wird und das Versetzen der aktuellen Position im Graphen daher nicht möglich ist.

$$\langle goto\ l_1; s_1;\ l_1 :\ s_2;, \sigma\rangle \Rightarrow_C \langle s_2, \sigma\rangle \tag{A.77}$$

A.19.4 Null_Expression

Dieser Knoten steht für den `Null`-Ausdruck, der in Sprachen wie zum Beispiel Ada verwendet wird, um leere Blöcke zu markieren. Es handelt sich hierbei also um eine leere Anweisung, die in keinster Weise behandelt werden muss. In C und C++ entspricht dieser Knoten dem alleinstehenden Semikolon.

In der IML findet sie Anwendung zum Beispiel als einzige Anweisung eines leeren *Else_Branch* oder ähnlicher optionaler Zweige.

$$\langle null, \sigma\rangle \Rightarrow_C \sigma \tag{A.78}$$

A.19.5 Field_Selection

Dieser Knoten repräsentiert den Zugriff auf ein Feld einer Klasse. Die *Selector*-Kante verweist dabei auf das aufzulösende Feld, während der *Operand* auf das Objekt verweist, dessen

Element verwendet werden soll. Das Beispiel in Kapitel A.49 entspricht dabei `this.x`, wobei `x` eine Member-Variable der Klasse ist, die mit `this` angesprochen wird.

Das Feld auf das verwiesen wird, das vom Typ *O_Field* ist, beinhaltet einen *Offset*. Anhand dieses Offsets und der eigentlichen Speicheradresse des dazugehörigen Objekts, läßt sich die absolute Speicheradresse des Feldes ermitteln.

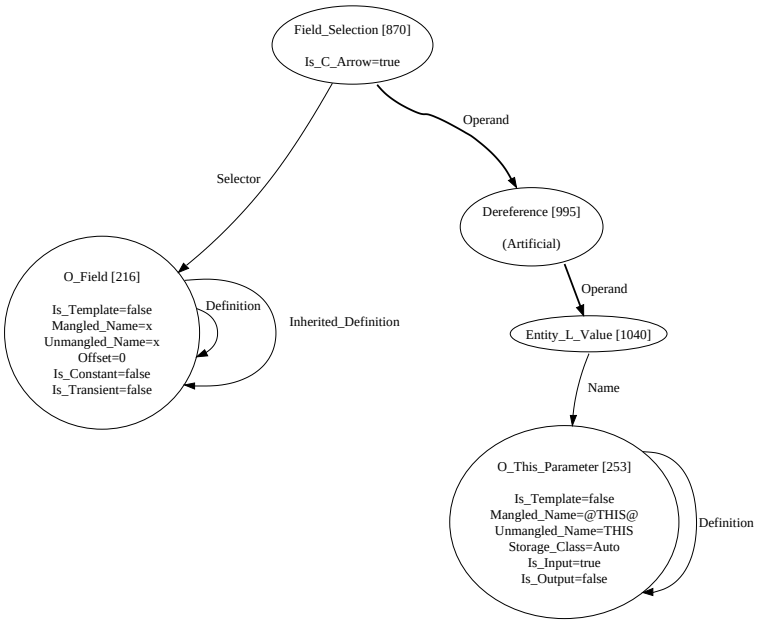

Abbildung A.49: Knoten Field_Selection

ANHANG B

C-Standardfunktionen

Im folgenden sind die unterstützten Funktionen der C-Standard-Library aufgeführt. Die einzelnen Tabellen stehen für die jeweiligen Header, die im C-Standard definiert sind. Nicht unterstützte Funktionen sind nicht in diesen Tabellen aufgeführt, sondern werden gesondert in Kapitel 3.2.1 erläutert.

Tabelle B.1: Funktionen aus `stdlib.h`

Signatur	Status	Beschreibung
`int abs(int num)`	Interface	Gibt den absoluten Wert eines Integers zurück
`void abort()`	Interface	Bricht die Ausführung des aktuellen Programms ab.
`double atof(const char *str)`	Interface	Konvertiert einen String in eine Fließkommazahl der Größe `double`
`int atoi(const char *str)`	Interface	Konvertiert einen String in einen Integer
`long atol(const char *str)`	Interface	Konvertiert einen String in einen `long`
`void exit(int exit_code)`	Emulation	Beendet das Programm und gibt den übergebenen Code zurück
`long labs(long num)`	Interface	Gibt den absoluten Wert des `long` zurück
`ldiv_t ldiv(long numerator, long denominator)`	Interface	Gibt den Quotienten und den Rest der Division in Form eines Structs zurück.
`int rand(void)`	Interface	Gibt eine Pseudozufallszahl zwischen 0 und `RAND_MAX` zurück
`void srand(unsigned seed)`	Interface	Initialisiert den Zufallsgenerator, wobei `seed` als Saat dient
`int system(char *command)`	Interface	Führt einen System Call durch und gibt das Ergebnis zurück
`void* calloc(size_t num, size_t size)`	Emulation	Reserviert Speicher für `num` Objekte der Größe `size`
`void free(void* ptr)`	Emulation	Befreit Speicher, der vorher mittels `malloc` oder `calloc` reserviert wurde
`void *malloc(size_t size)`	Emulation	Reserviert Speicher der Größe `size`
`div_t div(int numerator, int denominator)`	Interface	Gibt Quotienten und Rest der Division beider Parameter in Form eines `div_t`-Structs zurück (siehe Kapitel 3.4.2.1.3).
`double strtod(char* str, char** end)`	Interface	Liest einen Double aus einem String und speichert einen Verweis auf das erste nicht mehr interpretierte Zeichen in `end`
`long strtol(char* str, char** end, int base)`	Interface	Liest einen Long zur Basis `base` aus einem String und speichert einen Verweis auf das erste nicht mehr interpretierte Zeichen in `end`
`char *getenv(const char *name)`	Interface	Gibt falls vorhanden den Wert der Umgebungsvariable `name` zurück.

`long strtoul(char* str, char** end, int base)`	Interface	Liest einen vorzeichenlosen Long zur Basis `base` aus einem String und speichert einen Verweis auf das erste nicht mehr interpretierte Zeichen in `end`

Tabelle B.2: Funktionen aus `time.h`

Signatur	Status	Beschreibung
`time_t time(time_t *time)`	Interface	Gibt die aktuelle Systemzeit zurück oder schreibt sie in die Variable auf die über den Parameter verwiesen wird, so dieser denn nicht `null` ist.
`size_t strftime(char *str, size_t maxsize, const char *fmt, struct tm *time)`	Interface	Wandelt die Zeit aus `tm` gemäßt dem Format `fmt` um und schreibt maximal `maxsize` Zeichen vom Ergebnis in den Buffer `str`.
`clock_t clock(void)`	Interface	Gibt zurück, wie lange das Programm bereits ausgeführt wird. Dieser Wert ist in Prozessorzeit.
`char *asctime(const struct tm *ptr)`	Interface	Wandelt das referenzierte Konstrukt in einen String um, der die entsprechende Zeit darstellt.
`time_t mktime(const tm *time)`	Interface	Wandelt die Zeitangabe aus `time` in den integralen Typ `time_t` um.
`char *ctime(const time_t *time)`	Interface	Wandelt die Zeit `time` in ein lokales Format.
`double difftime(time_t time2, time_t time1)`	Interface	Gibt die Differenz der beiden Zeiten in Sekunden zurück.

Tabelle B.3: Funktionen aus `math.h`

Signatur	Status	Beschreibung
`double acos(double arg)`	Interface	Berechnet den Arkuskosinus von `arg`.
`double asin(double arg)`	Interface	Berechnet den Arkussinus von `arg`.
`double atan(double arg)`	Interface	Berechnet den Arkustangenz von `arg`.
`double atan2(double y, double x)`	Interface	Berechnet den Arkustangenz von `y/x` unter Verwendung der Vorzeichen der Operanden zur Auflösung des Quadranten.
`double ceil(double num)`	Interface	Gibt den kleinsten Integer zurück, der nicht kleiner als `num` ist.
`double cos(double arg)`	Interface	Berechnet den Kosinus von `arg`.
`double cosh(double arg)`	Interface	Berechnet den hyperbolischen Kosinus von `arg`.
`double exp(double arg)`	Interface	Berechnet den Exponenten von `arg`.
`double fabs(double arg)`	Interface	Gibt den absoluten Wert von `arg` zurück.
`double floor(double arg)`	Interface	Gibt den größten Integer-Wert zurück, der nicht größer als `arg` ist.
`double fmod(double x, double y)`	Interface	Gibt den Rest der Division `x/y` zurück.
`double frexp(double num, int* exp)`	Interface	Zerlegt `num` in eine Mantisse zwischen 0.5 und 1 und einen Exponenten, der in `exp` abgelegt wird.
`double ldexp(double num, int exp)`	Interface	Gibt `num * (2êxp)` zurück.
`double log(double num)`	Interface	Gibt den natürlichen Logarithmus von `num` zur Basis e zurück.
`double log10(double num)`	Interface	Gibt den Logarithmus von `num` zur Basis 10 zurück.
`double modf(double num, double *i)`	Interface	Zerlegt `num` in seinen Integer und seinen Bruchteil.
`double pow(double base, double exp)`	Interface	Gibt `baseêxp` zurück.
`double sin(double arg)`	Interface	Gibt den Sinus von `arg` zurück.
`double sinh(double arg)`	Interface	Gibt den hyperbolischen Sinus von `arg` zurück.
`double sqrt(double num)`	Interface	Berechnet die Wurzel von `num`.
`double tan(double arg)`	Interface	Berechnet die Tangente von `arg`.
`double tanh(double arg)`	Interface	Berechnet die hyperbolische Tangente von `arg`.

Tabelle B.4: Funktionen aus `stdio.h`

Signatur	Status	Beschreibung
`int printf(const char *format, ...)`	Interface	Gibt die zusätzlichen Parameter gemäß dem angegebenen Format aus. Der Rückgabewert entspricht der Anzahl an geschriebenen Zeichen (siehe Kapitel 3.4.2.1.4).
`int sprintf(char *buffer, const char *format, ...)`	Interface	Schreibt die zusätzlichen Parameter gemäß dem angegebenen Format in den Buffer. Der Rückgabewert entspricht der Anzahl an geschriebenen Zeichen (siehe Kapitel 3.4.2.1.4).
`int fprintf(FILE *stream, const char *format, ...)`	Interface	Schreibt die zusätzlichen Parameter gemäß dem angegebenen Format in die angebene Datei. Der Rückgabewert entspricht der Anzahl an geschriebenen Zeichen (siehe Kapitel 3.4.2.1.4).
`int vprintf(char *format, va_list arg_ptr)`	Interface	Diese Funktionen gibt die in der Parameterliste angegebenen Variablen gemäß dem Formatstring aus (siehe Kapitel 3.2.1.1).
`int vsprintf(char *buffer, char *format, va_list arg_ptr)`	Interface	Diese Funktionen schreibt die in der Parameterliste angegebenen Variablen gemäß dem Formatstring in den Buffer (siehe Kapitel 3.2.1.1).
`int vfprintf(FILE *stream, char *format, va_list arg_ptr)`	Interface	Diese Funktionen schreibt die in der Parameterliste angegebenen Variablen gemäß dem Formatstring in den Stream (siehe Kapitel 3.2.1.1).
`int scanf(const char *format, ...)`	Interface	Liest eine Zeile von `stdin` und versucht diese auf das Format zu matchen und in die gegebenen zusätzlichen Parameter einzutragen. Über das Interface werden momentan bis zu 100 optionale Parameter unterstützt (siehe Kapitel 3.4.2.1.5).
`int sscanf(const char *str, const char *format, ...)`	Interface	Versucht `str` auf das Format zu matchen und in die gegebenen zusätzlichen Parameter einzutragen. Über das Interface werden momentan bis zu 100 optionale Parameter unterstützt (siehe Kapitel 3.4.2.1.5).

`char *gets(char *str)`	Interface	Liest eine Zeile von `stdin` und schreibt diese in den Buffer `str`. Das Verhalten ist undefiniert, wenn ein String eingelesen wird, der länger ist als der Speicher auf den `str` verweist. Daher wird von der Benutzung der Methode generell abgeraten.
`int putchar(int ch)`	Interface	Schreibt ein Zeichen in `stdout` und gibt das geschriebene Zeichen zurück.
`int puts(char *str)`	Interface	Schreibt eine Zeichenkette in `stdout` und gibt den Status der Aktion als Integer zurück.
`int fclose(FILE *stream)`	Interface	Schließt eine geöffnete Datei und gibt 0 bei Erfolg zurück.
`int feof(FILE *stream)`	Interface	Gibt einen Wert ungleich 0 zurück, wenn das Ende der Datei erreicht wurde.
`int ferror(FILE *stream)`	Interface	Prüft den gegebenen Stream auf Fehler und gibt 0 zurück, wenn keiner aufgetreten ist.
`int fflush(FILE *stream)`	Interface	Handelt es sich bei dem Stream um einen Ausgabestream, so wird der Ausgabebuffer komplett ausgelesen. Handelt es sich dabei um einen Eingabestream, so wird der Eingabebuffer geleert.
`int fgetc(FILE *stream)`	Interface	Gibt das nächste Zeichen aus dem Stream `stream` zurück.
`char *fgets(char *str, int num, FILE* stream)`	Interface	Liest `num - 1` Zeichen aus `stream`, hängt ein Nullbyte an und schreibt das Ergebnis in `str` und gibt den Pointer zurück.
`FILE *fopen(const char *fname, const char *mode)`	Interface	Öffnet die Datei `fname` mit dem Modus `mode` und gibt einen entsprechenden Stream zurück.
`int fputc(int ch, FILE *stream)`	Interface	Schreibt das Zeichen `ch` in den Stream `stream` und gibt das geschriebene Zeichen (oder einen Fehlerwert) zurück.
`int fputs(const char *str, FILE *stream)`	Interface	Schreibt den String `str` in den Stream `stream` und gibt die Anzahl an geschriebenen Zeichen (oder einen Fehlerwert) zurück.
`int fread(void *buffer, size_t size, size_t num, FILE *stream)`	Interface	Diese Funktion liest `num` Objekte der Größe `size` aus dem Stream `stream` und schreibt sie in den Buffer `buffer`. Der Rückgabewert ist die Anzahl an gelesenen Zeichen.

`int putc(int ch, FILE* stream)`	Interface	Schreibt das Zeichen `ch` in den Stream `stream` und gibt bei Erfolg das geschriebene Zeichen zurück.
`int getc(FILE* stream)`	Interface	Liest ein Zeichen aus dem Stream `stream` und gibt es im Erfolgsfall zurück.
`int getchar()`	Interface	Liest ein Zeichen von `stdin` und gibt es zurück.
`void clearerr(FILE *stream)`	Interface	Setzt alle Fehlermarkierungen des Streams zurück.
`int rename(char *oldfname, char *newfname)`	Interface	Benennt eine Datei um.
`int remove(char *fname)`	Interface	Löscht die genannte Datei.
`void rewind(FILE *stream)`	Interface	Setzt den Zeiger innerhalb des Streams auf die erste Position zurück
`char *tmpnam(char *name)`	Interface	Gibt den Namen (inkl. Verzeichnis) einer möglichen temporären Datei zurück.
`FILE *tmpfile()`	Interface	Erstellt und öffnet eine temporäre Datei und gibt diese zurück.
`int fgetpos(FILE *stream, fpos_t *pos)`	Interface	Schreibt die aktuelle Position innerhalb des Streams `stream` in `pos`.
`int fsetpos(FILE *stream, fpos_t *pos)`	Interface	Setzt die aktuelle Position im Stream `stream` auf `pos`.
`FILE *freopen(const char *fname, const char *mode, FILE *stream)`	Interface	Ersetzt den gegebenen Stream durch einen neuen, der anhand des gegebenen Namen und Modus geöffnet wird.
`long ftell(FILE *stream)`	Interface	Gibt die aktuelle Position im Stream zurück.
`int fseek(FILE *stream, long offset, int origin)`	Interface	Setzt die aktuelle Position innerhalb des Streams anhand eines Offsets und einem Ursprung.
`int fwrite(const void *buffer, size_t size, size_t count, FILE *stream)`	Interface	Schreibt `count` Objekte der Größe `size` aus `buffer` in den Stream `stream` und gibt die Anzahl der geschriebenen Objekte wieder.
`void setbuf(FILE *stream, char *buffer)`	Interface	Setzt den Buffer des Streams mit der festgesetzten Größe.
`int setvbuf(FILE *stream, char *buffer, int mode, size_t size)`	Interface	Setzt den Buffer des Streams mit einer festen Größe.
`int ungetc(int ch, FILE *stream)`	Interface	Legt das Zeichen `ch` zurück in den Stream `stream`.
`void perror(const char *str)`	Interface	Schreibt `str` und eine vom Compiler abhängige Fehlermeldung, die von der globalen `errno`-Variable abhängt, auf die Konsole.

Tabelle B.5: Funktionen aus `locale.h`

Signatur	Status	Beschreibung
`char *setlocale(int category, const char * locale)`	Interface	Setzt bzw. gibt die akutelle Lokale zurÃ¼ck, je nachdem welche Parameter angegbeen werden.

Tabelle B.6: Funktionen aus `ctype.h`

Signatur	Status	Beschreibung
`int isalnum(int ch)`	Interface	Gibt einen Wert ungleich 0 zurück, wenn es sich bei `ch` um einen alphanumerischen Wert handelt.
`int isalpha(int ch)`	Interface	Gibt einen Wert ungleich 0 zurück, wenn es sich bei `ch` um ein Zeichen des Alphabets handelt.
`int iscntrl(int ch)`	Interface	Gibt einen Wert ungleich 0 zurück, wenn es sich bei `ch` um ein Kontrollzeichen handelt.
`int isdigit(int ch)`	Interface	Gibt einen Wert ungleich 0 zurück, wenn es sich bei `ch` um eine Ziffer handelt.
`int isgraph(int ch)`	Interface	Gibt einen Wert ungleich 0 zurück, wenn es sich bei `ch` um ein darstellbares Zeichen handelt, das sichtbar ist (exklusive dem Leerzeichen).
`int isprint(int ch)`	Interface	Gibt einen Wert ungleich 0 zurück, wenn es sich bei `ch` um ein darstellbares Zeichen handelt (inklusive dem Leerzeichen).
`int islower(int ch)`	Interface	Gibt einen Wert ungleich 0 zurück, wenn es sich bei `ch` um ein Kleinschriftzeichen handelt.
`int ispunct(int ch)`	Interface	Gibt einen Wert ungleich 0 zurück, wenn es sich bei `ch` um ein darstellbares Zeichen handelt, das weder alphanumerisch noch ein Leerzeichen ist.
`int isspace(int ch)`	Interface	Gibt einen Wert ungleich 0 zurück, wenn es sich bei `ch` um ein Leerzeichen handelt (Space, Tab etc.).
`int isupper(int ch)`	Interface	Gibt einen Wert ungleich 0 zurück, wenn es sich bei `ch` um ein Großschriftzeichen handelt.
`int isxdigit(int ch)`	Interface	Gibt einen Wert ungleich 0 zurück, wenn es sich bei `ch` um ein gültiges Hexadezimalzeichen handelt.
`int tolower(int ch)`	Interface	Gibt die Kleinschriftversion des Zeichens `ch` zurück.
`int toupper(int ch)`	Interface	Gibt die Großschriftversion des Zeichens `ch` zurück.

Tabelle B.7: Funktionen aus `string.h`

Signatur	Status	Beschreibung
`void *memchr(const void *buffer, int ch, size_t count)`	Interface	Sucht `ch` in den ersten `count`-Zeichen von `buffer` und gibt dessen Position zurück.
`int memcmp(const void *buffer1, const void *buffer2, size_t count)`	Interface	Vergleicht die ersten `count` Zeichen der beiden Buffer.
`void *memcpy(void *to, const void *from, size_t count)`	Interface	Kopiert die ersten `count` Zeichen aus `from` in `to` und gibt `to` zurück.
`void *memmove(void *to, const void *from, size_t count)`	Interface	Kopiert die ersten `count` Zeichen aus `from` in `to` und gibt `to` zurück. Ist im Gegensatz zu `memcpy` für den Fall der Überlappung von `from` und `to` definiert.
`void *memset(void *buffer, int ch, size_t count)`	Interface	Schreibt `ch` in die `count` ersten Zeichen von `buffer`.
`char *strcat(char *str1, const char *str2)`	Interface	Hängt `str` an `str1` an und gibt `str1` zurück.
`char *strchr(const char *str, int ch)`	Interface	Gibt einen Pointer auf das erste Vorkommen von `ch` in `str` zurück.
`int strcmp(const char *str1, const char *str2)`	Interface	Vergleicht die beiden Strings `str1` und `str2`.
`int strcoll(const char *str1, const char *str2)`	Interface	Vergleicht die beiden Strings `str1` und `str2` gemäßt der aktuell gesetzten *Locale*.
`char *strcpy(char *to, const char *from)`	Interface	Kopiert `from` in `to` bis inklusive dem letzten Nullbyte.
`size_t strcspn(const char *str1, const char *str2)`	Interface	Gibt die Position des ersten Zeichens auf `str1` zurück, das in `str2` vorkommt.
`size_t strlen(char *str)`	Interface	Gibt die Länge des Strings `str` zurück (exklusive dem Nullbyte).
`char *strncat(char *str1, const char *str2, size_t count)`	Interface	Hängt die ersten `count` Zeichen von `str2` an `str1` an und gibt `str1` zurück.
`int strncmp(const char *str1, const char *str2, size_t count)`	Interface	Vergleicht die ersten `count` Zeichen von `str1` und `str2`.
`char *strncpy(char *to, const char *from, size_t count)`	Interface	Kopiert `count` Zeichen aus `from` in `to`.
`char *strpbrk(const char* str1, const char *str2)`	Interface	Gibt einen Zeiger auf das erste Zeichen aus `str1` zurück, das in `str2` vorkommt.
`char *strrchr(const char *str, int ch)`	Interface	Gibt einen Zeiger auf das letzte Vorkommen von `ch` aus `str` zurück.

`size_t strspn(const char *str1, const char *str2)`	Interface	Gibt die Position des ersten Zeichens aus `str1` zurück, das nicht in `str2` vorkommt.
`char *strstr(const char *str1, const char *str2)`	Interface	Gibt einen Pointer auf das erste Vorkommen von `str2` in `str1` zurück.
`double strtod( const char *start, char **end)`	Interface	Gibt einen `double` zurück, der aus dem String `start` gelesen wurde. Das Ende der Zahl im String wird in `end` gesetzt.
`long strtol(const char *start, char **end, int base)`	Interface	Gibt einen `long` der zur Basis `base` aus dem String gelesen wurde, der mit `start` referenziert wird, zurück. Die Adresse des ersten nicht geparsten Zeichens wird in `end` geschrieben.
`unsigned long strtoul(const char *start, char **end, int base)`	Interface	Gibt einen `unsigned long` der zur Basis `base` aus dem String gelesen wurde, der mit `start` referenziert wird, zurück. Die Adresse des ersten nicht geparsten Zeichens wird in `end` geschrieben.
`char *strtok(char *str1, const char *str2)`	Interface	Zerlegt den String `str1` anhand der Trennzeichen aus `str2` und gibt das erste Element zurück. Bei jedem weiteren Aufruf mit `NULL` als zweiten Parameter, wird das nächste Element zurückgegeben.
`size_t strxfrm(char *str1, const char *str2, size_t num)`	Interface	Bereitet die ersten `num` Zeichen aus `str2` so vor und speichert sie in `str1`, daß `strcoll` die gleichen Ergebnisse liefert wie `strcmp`.

Tabelle B.8: Funktionen aus `fenv.h`

Signatur	Status	Beschreibung
`int feclearexcept(int flag)`	Interface	Entfernt eine gesetzte Exception aus der `fexcept_t`-Variable.
`int fegetexceptflag(fexcept_t *e, int flag)`	Interface	Gibt den Status der Exception `flag` aus `e` zurück.
`int feraiseexcept(int flag)`	Interface	Setzt die Exception `flag` in der aktuellen `fexcept_t`-Variable.
`int fesetexceptflag(const fexcept_t *e, int flag)`	Interface	Setzt die Exception `flag` in `e`.
`int fetestexcept(int flag)`	Interface	Prüft ob die Exception `flag` geworfen wurde.
`int fegetround()`	Interface	Gibt das aktuelle Rundungsverhalten bei Floating-Point-Variablen zurück.
`int fesetround(int flag)`	Interface	Setzt das Rundungsverhalten für Floating-Point-Variablen.
`int fegetenv(fenv_t *e)`	Interface	Gibt die aktuelle Umgebungskonfiguration für Floating-Points zurück.
`int feholdexcept(fenv_t *e)`	Interface	Setzt `e` als Variable, in der geworfene Exceptions abgelegt werden sollen.
`int fesetenv(const fenv_t *e)`	Interface	Setzt `e` als aktuelle Umgebungsvariable für Floating-Points.
`int feupdateenv(const fenv_t *e)`	Interface	Speichert die aufgetretenen Floating-Point-Exceptions in `e`.

ABBILDUNGSVERZEICHNIS

TABELLENVERZEICHNIS

LISTINGS

LITERATURVERZEICHNIS

[1] *Projekt Bauhaus*.

[2] *unparse-Projekt*.

[3] *The C Standard*. Wiley and Sons, 2003.

[4] Brian W. Kernighan, D. M. R.: *The C Programming Language*, 1988.

[5] Jones, D. M.: *The New C Standard*. 2005.

[6] Keul, S.: *Generierung der Zwischendarstellung IML für Ada95 Programme*. Diplomarbeit, Universität Stuttgart, Institut für Softwaretechnologie, September 2005.

[7] Knauss, M.: *Erweiterung und Generierung der Zwischendarstellung IML für Java-Programme*. Diplomarbeit, Universität Stuttgart, Institut für Softwaretechnologie, Oktober 2002.

[8] Koschke, R. und J.-F. Girard: *An Intermediate Representation for Reverse Engineering Analyses*. In: *Working Conference on Reverse Engineering*, S. 241–250, 1998.

[9] Pingel, S.: *Abstrakte Syntaxbäume*.

[10] Pingel, S.: *Generierung der Zwischendarstellung IML aus Java Classfiles*. Diplomarbeit, Universität Stuttgart, Institut für Softwaretechnologie, Mai 2006.

[11] Rohrbach, J.: *Erweiterung und Generierung einer Zwischendarstellung für C-Programme*, Januar 1998.

[12] Sedgewick, R.: *Algorithms*. Addison-Wesley, Publishing Company, Inc, 1983.

[13] Stroustrup, B.: *The C++ Programming Language*. Addison-Wesley, Publishing Company, Inc, 1987.

[14] Taft, S. T.: *Ada 95 Reference Manual*. Springer Verlag, 1997.

[15] Tahir Karaca, S. S.: *Erweiterung und Generierung der Zwischendarstellung IML für C++ Programme*. Diplomarbeit, Universität Stuttgart, Institut für Softwaretechnologie, März 2003.

INDEX