Institut für Softwaretechnologie
Abteilung Programmiersprachen

Universität Stuttgart
Universitätsstr. 38
70569 Stuttgart

Masterarbeit

Design und Implementierung eines Linkers für SKiIL/Bauhaus

Simon Hanna

Studiengang:	Informatik
Prüfer/in:	Prof. Dr. rer. nat./Harvard Univ. Erhard Plödereder
Betreuer/in:	Dr. rer. nat. Timm Felden
Beginn am:	16. November 2017
Beendet am:	16. Mai 2018

Kurzfassung

Bauhaus ist eine Werkzeugkette die zur Programmanalyse verwendet werden kann. Dabei wurde die Bauhaus Intermediate Language (IML) als Zwischendarstellung verwendet. Mittlerweile verwenden viele der Werkzeuge Schnittstellen die von der Serialization Killer Language (SKilL) erstellt werden. Der von Bauhaus verwendete Linker wurde bisher noch nicht angepasst, weil er direkt auf dem Binärstrom der Zwischenprache arbeitet.

In der vorliegenden Arbeit wird ein neuer Linker für Bauhaus entworfen. Dieser basiert auf einer von `SKilL` bereitgestellten API und ist dadurch wesentlich einfacher zu verstehen und zu warten. Außerdem werden andere Werkzeuge, die Teil des Front-Ends von Bauhaus sind überarbeteitet um das Front-End zu entschlacken.

Inhaltsverzeichnis

1 Einleitung

Bauhaus ist eine Werkzeugkette für Programmanalysen. Sie ist größtenteils in Ada geschrieben und verwendet IML, ein Binärformat, um Programme und deren Analyseergebnisse in einem Graphen zu speichern. Dennis Przytarski hat in seiner Masterarbeit [Prz16] damit begonnen, `SKilL` [Fel14] einzuführen, um die weitere Entwicklung einfacher zu machen. Dabei wurde die vorhandene IML-Schnittstelle angepasst um `SKilL` zu verwenden.

Es wurde allerdings festgestellt, dass der Linker nicht migriert werden kann, weil dieser anders als andere Werkzeuge in Bauhaus, nicht nur die IML-Schnittstelle verwendet, sondern teilweise direkt auf dem Binärstrom der Dateien arbeitet. Daher konnten ab dem Zeitpunkt der Umstellung keine Programme mehr ganz übersetzt werden. Es konnten zwar einzelne Dateien mithilfe des C/C++-Front-Ends gelesen werden, diese konnten allerdings nicht gelinkt werden. Abhilfe verschaffte das Werkzeug `iml2sf` [FW16], welches in der Lage ist, Dateien aus dem alten Format in das neue zu übersetzen.

Da der Linker auf dem Binärstrom arbeitet, ist es praktisch unmöglich, diesen weiter zu verwenden. Stattdessen soll im Laufe dieser Arbeit ein Linker neu implementiert werden.

Gliederung

In Kapitel 2 werden die Ziele der Arbeit vorgestellt.

In Kapitel 3 wird die Arbeitsweise des alten Linkers vorgestellt.

In Kapitel 4 wird der neue Linker vorgestellt und dabei die gewählten Regeln zur Unifikation besprochen.

In Kapitel 5 wird `cafe++` vorgestellt und die Änderungen, die daran vorgenommen werden.

Die Kapitel 6 und 7 behandeln die Werkzeuge `imlstrip` und `imloptimizer`. Die Funktionalität der Programme wird vorgestellt und es wird erklärt welche Funktionen übernommen werden.

In Kapitel 8 werden die Tests für die einzelnen Komponenten vorgestellt.

In Kapitel 9 werden offene Punkte behandelt, die in Zukunft bearbeitet werden könnten.

Kapitel 10 fasst die Arbeit zusammen.

2 Ziele der Arbeit

Hauptziel der Arbeit ist es, den vorhandenen Linker durch einen neuen zu ersetzen. Da der alte Linker teilweise direkt auf dem Binärstrom arbeitet, ist die Migration zu `SKilL` mit erheblichem Aufwand verbunden. Der neue Linker soll dabei die von `SKilL` bereitgestellte API verwenden, damit Änderungen deutlich einfacher eingepflegt werden können.

Das Front-End von Bauhaus umfasst aktuell mehrere unabhängige Werkzeuge. Einstiegspunkt ist dabei das Werkzeug `cafeCC`, welches dafür zuständig ist, alle Werkzeuge des Front-Ends nacheinander aufzurufen. Im Normalfall werden daher folgende Werkzeuge nacheinander auf den Eingabedateien aufgerufen.

1. `cafe++`
   ist das C/C++ Front-End von Bauhaus. Es ist dafür zuständig, Programmdateien einzulesen und einen Graphen für jede Datei zu erstellen.
2. `imlstrip`
   ist ein Werkzeug, das verwendet wird, um bestimmte ungültige Knoten aus den Graphen zu entfernen.
3. `imllink`
   ist der Linker, welcher alle Eingabegraphen zu einem einzigen zusammenführt.
4. `imloptimizer`
   ist ein Werkzeug, um unterschiedliche Konstrukte aus den Graphen zu entfernen. Es kann fünf sogenannte Optimierungen am Graphen durchführen, wobei alle darauf abzielen, Knoten zu entfernen.

Im Rahmen der Arbeit sollen neben der Neuimplementierung des Linkers außerdem einige Änderungen an `cafe++` durchgeführt werden. `imlstrip` und `imloptimizer` sollen möglichst komplett entfernet und dabei ihre Funktionalität in andere Werkzeuge eingepfelgt werden.

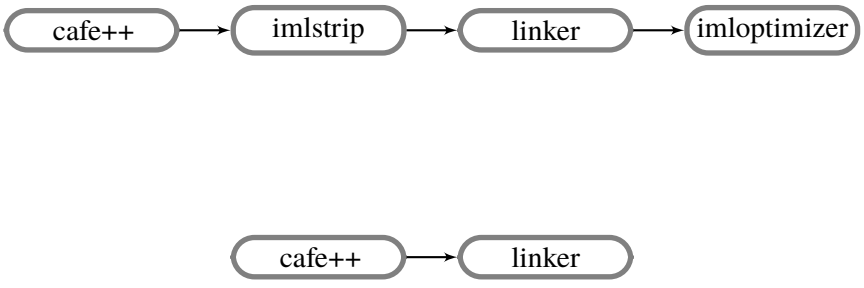

Abbildung 2.1: Front-End Toolchain vor und nach der Arbeit

3 Bisheriger Linker

Weil der alte Linker teilweise `libiml` von Bauhaus verwendet, aber auch direkt auf dem Binärstrom arbeitet, ist dieser nur sehr schwer zu verstehen und seine Arbeitsweise ist kaum bekannt. Dies wird dadurch verstärkt, dass der Linker im wesentlichen nicht kommentiert ist. Manche Kommentare sind eindeutig kopiert worden, ohne Anpassungen vorzunehmen, wodurch sie nachweislich falsch sind.

Im Laufe der Arbeit wurde festgestellt, dass der Linker sich auf Strukturen bezieht, die nicht Teil der Spezifikation von IML sind. Diese Funktionalität ist in libiml versteckt und wird beim Aufruf von `cafe++` in die `.iml`-Dateien geschrieben.

Die Funkionalität des Linkers ist in vier Phasen unterteilt. Dabei kann die Vorarbeit durch `cafe++` bzw. `libiml` als nullte Phase angesehen werden.

3.1 Vorarbeit durch `cafe++`

Neben dem Graphen erstellt `cafe++` auch einige Strukturen, die haupsächlich für den Linker relevant sind. So werden für jede Übersetzungseinheit in Unit die DeclarationTable und UnresolvedDeclarations angelegt. Diese Informationen werden vom Linker verwendet, um zu entscheiden, ob die Übersetzungseinheit überhaupt gebraucht wird oder nicht.

Außerdem wird von libiml eine Map[TNode -> String] erstellt. Diese Struktur wird in die Dateien geschrieben, ist jedoch nicht in der Hierarchie der IMLStruktur dokumentiert. Daher wurde diese Struktur bei der Migration hin zu `SKiL` nicht aufgenommen. Es gibt auch keine Verweise innerhalb des Graphen darauf. Da diese Struktur ausschließlich vom Linker verwendet wird und dieser direkt auf dem Binärstrom arbeitet, ist die Struktur weitgehend unsichtbar.

In dieser Struktur wird für jede TNode ein zusätzliches Name-Mangling durchgeführt. Ziel ist es, allen Typen, welche vom Linker zusammengelegt werden sollen, den selben Namen zu geben.

Das Mangling ist in `bauhaus/projects/libs/iml/utilities/t_nodes_name_mangling.adb` definiert. Geschrieben wird die Struktur in `bauhaus/projects/libs/iml/src/iml_graphs.adb` von der Methode Write_Types.

Die Struktur wird von der `libiml` zwar auch beim Laden von Dateien gelesen, aber nicht exportiert und ist dadurch von außen nur schreib-, nicht aber lesbar.

Soweit es beurteilt werden kann, wird die Struktur nur vom Linker verwendet, da dieser, anders als die anderen Werkzeuge, direkt auf dem Binärstrom der `.iml`-Dateien arbeitet. Es sollte aber untersucht werden, ob andere Werkzeuge in Bauhaus auch direkt auf dem Binätstrom arbeiten. Diese könnten diese Struktur ebenfalls verwenden.

3.2 Programm-Parameter

No Declaration as Definition: Eine Definition ist ein ONode-Knoten, dessen Definitionszeiger auf sich selbst zeigt. Für den Fall, dass der Linker für eine Deklaration keine Defintion findet, wird im Normalfall eine zufällige Deklaration zur Defintion befördert, indem der Definitionszeiger gesetzt wird.

Wenn diese Option allerdings gesetzt ist, werden keine Deklarationen zu Definitionen befördert. Dies wird vermutlich getan, damit im Nachhinein noch ersichtlich ist, dass es sich nicht um eine gültige Definition handelt.

Remove Duplicate Definitions: Diese Option wird verwendet um zu erzwingen, dass Deklarationen, für die es mehr als eine Definition gibt, unifiziert werden. Weshalb dies als eigenständige Option vorhanden ist, ist unklar.

Remove MangledNames: Falls diese Option gesetzt ist, werden alle MangledNames von ONodes entfernt. MangledNames sollten nur für den Linker relevant sein, da anhand dieser Information die Entscheidung der Unifikation getroffen wird. Das Entfernen von MangledNames sollte nur stattfinden, wenn sichergestellt ist, dass kein weiterer Linkvorgang durchgeführt wird. Diese Option kann also als reine Speicheroptimierung angesehen werden.

Remove MangledTypeNames: Damit ist gemeint, ob die versteckte Map von TNodes zu Strings in die Ausgabedatei geschrieben werden soll. Diese Informationen sind nach der Migration zu `SKilL` nicht mehr in den `.iml.sf`-Dateien vorhanden. Es ist äußert unwahrscheinlich, dass ein anderes Werkzeug auf diese Informationen zurückgreift, da dafür der Binärstrom betrachtet werden muss. Die Option ist wahrscheinlich ebenfalls wie das Entfernen von MangledNames nur relevant, wenn keine weiteren Linkaufrufe erfolgen. Weshalb dafür getrennte Optionen verwendet werden ist nicht ersichtlich.

Shared Linking: Das Setzen dieser Option wirkt sich nur darauf aus, ob eine Warnung beim Fehlen einer Mainmethode ausgegeben wird oder nicht. Es wird auch verwendet, um zu signalisieren, dass das Fehlen von manchen Definitionen kein Fehler ist. Die fehlenden Definitionen sollten bei einem späteren Linkvorgang ergänzt werden.

Include Unused Units: Der Parameter entscheidet darüber, ob Übersetzungseinheiten behalten werden sollen, obwohl keine Defintion aus der Einheit Verwendung findet.

3.3 Phase 1

In der ersten Phase werden globale Variablen mit Werten initialisiert, die durch das erste Einlesen der `.iml`-Dateien gewonnen werden. Hier wird entschieden, ob die Datei für den Linkvorgang relevant ist. Das wird anhand der DeclarationTable und UnresolvedDeclarations der Übersetzungseinheiten entschieden. Außerdem wird in dieser Phase auch die versteckte Typ Map geladen.

3.4 Phase 2

In der zweiten Phase findet das tatsächliche Linken statt. Hierzu wird die versteckte Map verwendet, um TNodes zu vereinen. Für TNodes ist die Entscheidung der Unifikation demnach schon vor Aufruf des Linkers getroffen. Dieser muss nur noch für jeden Typnamen einen passenden TNode zufällig auswählen.

In dieser Phase werden auch ONodes vereint. Dafür unterscheidet der Linker drei Fälle:

- Gruppen mit nur einer Definition:
  Hier ist das Unifikationsziel klar.
- Gruppen ohne Definition:
  In diesem Fall wird eine beliebige Deklaration verwendet.
- Gruppen mit mehreren Defintionen:
  Hier wird die erste vollständige Definition verwendet.

Bei Gruppen mit mehreren Definitionen wird allerdings geprüft, ob mehrere Definitionen überhaupt erlaubt sind. Diese Information wird von libIML in die Graphen geschrieben. Dokumentiert ist dies aber nicht. In `SKilL` wurde diese Information nicht übernommen.

Die Information zur Vollständigkeit wird ebenfalls von libIML geschrieben und ist in `SKilL` nicht mehr enthalten. Das Fehlen dieser Informationen ist nicht weiter schlimm, weil sie direkt aus den Eigenschaften der Knoten abgeleitet werden können.

Die Entscheidung ob ein ONode vollständig ist oder nicht, wird in libIML anhand der definierten Typgröße entschieden. Wenn die Größe 0 ist, ist ein Typ unvollständig.

Ob von einem ONode mehrere Defintionen erlaubt sind oder nicht, hängt vom Subtyp ab. OComposites, ONamespaces, OEnumerators und OFields dürfen mehrfach vorhanden sein. Von allen anderen ONodes ist nur eine Defintion erlaubt.

Beim Verstoß gegen die Regeln wird lediglich eine Warnung ausgegeben und die erste Definition verwendet.

Nach einem Kommentar im Code von `imllink` scheint die Reihenfolge der ONodes bei Verwendung von Precompiled Header (PCH) wichtig zu sein. PCHs werden verwendet, um einzelne Header nur einmal übersetzen zu müssen. Die Dokumentation die dem verwendeten kommerziellen Front-Ends beiliegt, enthält ein Kapitel über PCHs. Weshalb die Reihenfolge der ONodes eine Rolle spielen sollte, ist nicht ersichtlich. Aus dem Code vom `imllink` ist dies ebenfalls nicht zu entnehmen.

Die Ordnung der ONodes ist durch die Migration zu SKilL definitiv nicht mehr gegeben. Falls die Ordnung doch relevant sein sollte, muss das Fehlen der Ordung anderweitig kompensiert werden.

3.5 Phase 3

In dieser Phase wird die Ausgabedatei durch das Kopieren aus den Eingabedateien erstellt.

3.6 Phase 4

Obwohl in der alten IML Struktur Daten unerreichbar sind, wenn es keine Zeiger mehr darauf gibt, scheint es Ausnahmen zu geben. In dem letzten Schritt wird die Ausgabedatei gelesen und neu geschrieben. Dadurch verschwinden alle unerreichbaren Knoten. Die Phase wird nur ausgeführt, wenn keine Fehler in vorangegangen Phasen aufgetreten sind.

4 Neuer Linker

In diesem Kapitel wird die Funktionsweise des neuen Linkers vorgestellt. Ziel des Linkers ist es, doppelte Strukturen zu unfizieren. Da Bauhaus ein Analysewerkzeug ist, sollen dabei aber keine Informationen zu dem Programmcode verloren gehen. Die Hauptaufgabe des Linkers besteht deshalb darin, zu entscheiden, welche Knotenarten unifiziert werden sollen und welche Regeln dabei angewandt werden.

Der Linker verwendet eine von `SKilL` erstellte API. Diese API wird anhand einer deutlich gekürzten Spezifikation erstellt. Dabei werden alle Knoten ignoriert, die für andere Programmiersprachen reserviert sind (z.B. Java). Außerdem werden alle Knoten für nachgelagerte Werkzeuge entfernt. Dadurch wird die erstellte API kleiner, was Speicherplatz und auch Ausführungszeit spart.

Die Vorgehensweise des neuen Linkers kann in vier Arbeitsschritte unterteilt werden.

1. Unifikationen berechnen.
2. Speicher für alle nötigen Knoten in der Ausgabedatei reservieren.
3. Eigenschaften der Knoten in die Ausgabedatei schreiben.
4. Ausgabedatei finalisieren.

4.1 Unifikation

Zu Beginn der Ausführung des Linkers findet die Unifikation der Knoten statt. Wichtig ist zu beachten, dass der Linker ausschließlich für die Programmiersprachen C und C++ verwendet wird. Daher beziehen sich die Regeln entweder auf Verhalten des alten Linkers oder direkt auf die Sprachstandards von C [ISO11a] und C++ [ISO11b].

Bei der Unifikation wird nur ein kleiner Teil der Knoten betrachtet, weil viele Knoten Syntax repräsentieren und diese daher nicht unifiziert werden sollten. Die wichtigste Aufgabe des Linkers ist es, vorhandene Typen (TNodes) und Deklarationen (ONodes) zu unifizieren.

Außerdem werden noch Identifier und SLocs unifiziert. Ihre Unifikation ist zwar nicht zwingend notwendig, aber so werden die Graphen kleiner, wodurch Speicherplatz und Ausführungszeit eingespart werden können.

Für jede Knotenart werden Gewinner gekürt. Die Gewinner stellen die Unifikationsziele dar, welche in die Ausgabedatei geschrieben werden. Alle anderen Knoten, die äquivalent zu den Gewinnern sind, werden im Nachhinein durch den jeweiligen Gewinner ersetzt.

Manche Knoten, die bestimmte Eigenschaften aufweisen z.B. Deklarationen ohne Namen, werden von der Unifikation ausgenommen.

Listing 4.1 Unifikation von Knoten

```
unifications = dict()

def unify(nodes, key_method, condition_method):
        winners = dict()
        for node in nodes:
            if condition_method(node):
                key = key_method(node)
                if key not in winners:
                    winners[key] = node
        for node in nodes:
            if condition_method(node):
                key = key_method(node)
                unifications.[node] = winners[key]
```

Bei ONodes gibt es zusätzliche Beschränkungen für die Gewinner. Sie sollten, wie in Abschnitt 3.4 erwähnt, vollständige Defintionen sein, falls diese überhaupt vorhanden sind.

Die allgemeine Vorgehensweise kann in Listing 4.1 nachverfolgt werden.

4.1.1 Ordnung der Unifikation

Für die gewählten Regeln der Unifikation ist es unabdingbar, die Unifikation in einer festen Reihenfolge durchzuführen. Da die Unifikationsregeln der TNodes meist von ONodes abhängig sind, müssen die ONodes zuerst unifiziert werden.

Folgende Knoten müssen nur einmal betrachtet werden, weil ihre Unifikationsregeln nur von direkten Eigenschaften abhängig sind. Dadurch ergibt sich eine totale Ordnung, wichtig ist deshalb nur, dass die Reihenfolge eingehalten wird.

1. SLocPositions können ganz am Anfang unifiziert werden, da diese keine externen Abhängigkeiten haben.
2. Identifier können auch zu Beginn unifiziert werden, weil der Schlüssel der Unifikation nur ein String ist.
3. SLocs basieren auf SLocPositions und Identifier.
4. TBaseTypes sind die Basistypen.
5. ONodes sind die Definitionen. Eine feste Reihenfolge der Subtypen ist nicht zwingend notwendig.
6. TNodes welche sich direkt auf eine Defintion verweisen.

Im Anschluss kann mit der Unifikation von Knoten fortgefahren werden, die mehrmals betrachtet werden müssen. Dies ist notwendig, da ihre Unifikation von Zeigern auf andere TNodes abhängig ist, wodurch eine zyklische Abhängigkeit entsteht. Diese Knoten werden gemeinsam in einer Schleife unifiziert, bis sich keine Änderungen mehr ergeben.

- TPointer

- TArrays
- TTypeQualifier

4.2 Unifikationsregeln

In diesem Kapitel werden die nötigen Regeln vorgestellt, anhand derer Knoten unifiziert werden.

Wichtig ist zu beachten, dass der Linker ausschließlich für die Programmiersprachen C und C++ verwendet wird. Daher beziehen sich die Regeln für die Unifikation auf Eigenschaften dieser Sprachen.

Die zu Unifizierenden Knoten lassen sich in drei Kategorien einteilen:

- TNodes
- ONodes
- Identifier und SLocs

Identifier und SLocs sollten für ein korrektes Verhalten nicht unifiziert werden müssen, allerdings wird durch die Unifikation Speicherplatz eingespart, weshalb eine Unifikation sinnvoll erscheint.

In dem alten Linker werden TNodes anhand ihres Namens unifiziert. Diese Namen wurden von libIML in die Graphen geschrieben. Bei der Migration zu `SKilL` fielen diese aber weg, weshalb sie nicht mehr zur Verfügung stehen. Aus diesem Grund, und auch, um die Funktionalität des Linkers an einer Stelle zu bündeln, wird daher eine alternative Methode verwendet.

TNodes werden anhand ihrer Eigenschaften unifiziert. Dabei werden Regeln aus dem C/C++ Standard angewandt.

ONodes haben in der Regel Namen, anhand derer sie unifiziert werden können. Für einzelne Subtypen gibt es Abweichungen, die in diesem Kapitel erkläutert werden.

4.2.1 Allgemeine Knoten und BasicTypes

Identifier Identifier werden verwendet, um Strings zu speichern. Obwohl es nicht zwingend nötig wäre, diese zu unifizieren, wird durch die Unifikation Speicherplatz gespart.

Unifiziert werden Identifier anhand des enthaltenen Strings.

SLocPosition SLocPositions werden verwendet, um Positionen in Dateien zu beschreiben und werden anhand der Reihe und Spalte unifiziert.

SLoc SLocs werden verwendet, um Positionen im Code zu beschreiben. Eine sinnvolle Regel für die Unifikation wäre, den Pfad sowie die Position zu verwenden.

Allerdings erfüllen SLocs noch eine zusätzliche Aufgabe. Um in IML zu vermerken, dass bestimmte Teile aus einer Makroexpansion resultieren, wird ein MacroContext an eine SLoc angeheftet. Warum dies so modelliert wurde, ist nicht ersichtlich.

Außerdem wurde bei Tests festgestellt, dass nicht alle SLocs über einen nicht-leeren Pfadnamen verfügen. Unifiziert wird daher unter der Bedingung, dass ein Pfadname exisitert.

Die endgültige Regel beachtet dabei neben der SLocPosition und dem Pfad auch den Macrocontext.

Es wird darauf verzichetet, SLocs ohne Macrocontext mit SLocs mit Macrocontext an der gleichen Position zu unifizieren, da nicht spezifiziert ist, wie der Macrocontext verwendet wird.

TVoid und TBoolean Von beiden Typen sollte es maximal eine Instanz geben. Deshlab wird die erste gefundene Instanz verwendet.

TCFloat Floats können in C/C++ nur eine Größe haben, weshalb hier auch die erste gefundene Instanz verwendet wird.

TCDouble Doubles werden anhand der Length unifiziert.

TCChar Chars gibt es in den Varianten signed und unsigned. Die Unifikation erfolgt anhand der Sign. Eventuelle qualifier werden hier nicht betrachtet, da es dafür eigene Knoten gibt.

TCPPWchar Von wchars gibt es wie von chars nur die Varianten signed und unsigned.

TCInt Neben Sign spielt hier auch Length eine Rolle.

IntLiteral Eine Unifikation scheint hier eigentlich nicht angebracht zu sein. Da aber die Länge von Arrays in IML anhand von IntLiterals modelliert ist, werden auch IntLiterals unifiziert.

Es wurde beobachtet, dass Intliterals von ArrayBounds über keine SLoc zu verfügen, deshalb werden nur IntLiterals unifiziert, die keine SLoc besitzen.

Unifiziert wird anhand des Typs.

IntegerConstant Auch hier sollte eine Unifikation eigentlich nicht durchgeführt werden. Weil auch IntegerContstants in Arrays verwendet werden, werden auch sie anhand des Typs unifiziert, aber nur wenn keine SLoc vorhanden ist.

4.2.2 ONodes

Da alle ONodes einen Namen haben, wird generell anhand des MangledNames unifiziert. Das erfolgt nur, wenn die Definition nicht als private markiert ist. Die Gewinner der Unifikation von ONodes sollten immer eine Defintion und nicht eine Deklaration repräsentieren.

OTypedef OTypedef werden unter anderem für typedefs aus C verwendet. Der C11-Standard [ISO11a] nach §6.7.3 erlaubt es, typedefs zu redefinieren, solange der selbe Typ verwendet wird und dieser nicht variabel ist.

Ein Linker sollte typedefs mit dem gleichen Namen problemlos unifizieren dürfen. Der vorhandene Linker tut dies jedoch nicht. Da Bauhaus ein Analysetool ist und es für manche Analysen sinnvoll sein könnte, auf diese Information zurückzugreifen, werden sie auch weiterhin nicht unifiziert. Falls doppelte OTypedefs doch nicht nötig sind, können sie anhand des Namens unifiziert werden.

OEnum OEnum repräsentiert Enums. Wenn Enums Namen haben, können sie einfach anhand des MangledName unifiziert werden. Enums können aber auch ohne Namen definiert werden, diese werden vom alten Linker nicht unifiziert.

Der neue Linker unifiziert Enums generell anhand des MangledName. Enums ohne Namen werden anhand der SLoc unifiziert.

OEnumerator OEnumerator ist ein Wert aus einem Enum. Bei der Verwendung des Werts wird ein Read-Knoten erstellt, welcher über Operand und dann über Name bei einem OEnumerator landet. Solange alle Zugriffe auf das Enum in einer Übersetzungseinheit liegen, erstellt `cafe++` nur eine Instanz von OEnumerator. Wenn es in mehreren Einheiten verwendet wird, unifiziert der alte Linker diese allerdings nicht.

Weshalb der alte Linker diese nicht unifiziert, kann nicht nachvollzogen werden. Da OEnumerator nicht für die Lesezugriffe auf die Werte verwendet werden, sondern nur die tatsächlichen Werte eines Enums repräsentieren, gibt es keinen Grund diese nicht zu unfizieren. Der neue Linker unifiziert OEnumerators anhand des MangledName.

OStackObject und OField Es wurden einzelne Objekte gefunden, die über keinen gültigen MangledName verfügen. Abhilfe verschafft das einbeziehen des Namespaces der Objekte.

4.2.3 TNodes

TNodes werden im Allgemeinen anhand des deklararierenden ONodes unifiziert. Ausnahmen sind die zuvor erwähnten BaiscTypes. Nachfolgend werden einige TNodes vorgestellt, für die es spezielle Regeln gibt.

TTypedefName Ein TTypedefName ist der Typ zu einem OTypedef. Hier ist es wichtig, dass es nur einen Typ von jedem Typedef gibt. Da OTypedefs nicht unifiziert werden, ist es hier nötig den Namen der Deklaration zu verwenden und nicht die Deklaration selbst. Falls in Zukunft OTypedefs unifiziert werden sollen, ist es möglich, diese Regel umzustellen.

TEnum TEnum ist der Typ eines Enums. Der alte Linker unifiziert diese nicht vollständig, weil er auch die OEnums nicht korrekt unifiziert.

Der neue Linker unifiziert sie anhand des OEnums, welches sie repräsentieren.

TRoutine Der alte Linker unifiziert TRoutines anhand ihrer Signatur. Die Signatur wird bereits beim Erstellen der initialen IML-Datei für jede TRoutine hinterlegt.

TRoutine enthält eine Liste von Parametern. OParameter haben einen Verweis auf die Implementierung ihrer deklarierenden Methode. Bei der Unifikation anhand der Signatur verweisen diese auf eine wilkührliche Methode, deren Signatur passt. Die aktuelle Modellierung erweckt durch die Verwendung von OParameter den Eindruck, dass TRoutines immer nur eine ORoutine repräsentieren.

Es kann argumentiert werden, dass TRoutines nur die Signaturen repräsentieren soll. Allerdings stellt sich dann die Frage, weshalb OParameter zur Modellierung der Parameter verwendet werden und nicht TNodes, welche methodenunabhängig wären. Ein Indiz dafür, dass eigentlich die Verwendung von TNodes richtig wäre, ist, dass TRoutines keinen Verweis auf die deklarierende ORoutine besitzen. Andere benutzerdefinierte Typen haben stets einen Verweis auf die jeweilige Deklaration.

Um der aktuellen Modellierung gerecht zu werden, gibt es in dem neuen Linker für jede TRoutine eine eindeutige ORoutine. Wenn in Zukunft nicht OParamter sondern TNodes für die Parameterrepräsentation verwendet werden, gibt es keinen Grund TRoutines nicht anhand der Signatur zur Unifizieren.

TArray TArrays werden anhand des elementaren Typs und der Größe unifiziert. In IML ist die Größe der Arrays nicht direkt als Zahlenwert modelliert, sondern indirekt über IntegerConstant und IntegerLiteral.

Dadurch, dass TArrays einen Verweis auf TNodes haben, ergibt sich eine zyklische Abhängigkeit, weshalb TArrays mehrmals betrachtet werden müssen.

TTypeQualifier TTypeQualifier werden unter der Annahme unifiziert, dass die Struktur des Graphen eine Rolle spielt. Es sollte sichergestellt sein, dass unterschiedliche Reihenfolgen der Qualifier im Sourcecode bereits vom Front-End entsprechend sortiert werden. In dem Linker führt eine Umstrukturierung zu einer teuren Operation in dem alle Kanten im Graphen überprüft werden müssen.

Dadurch, dass TTypeQualifier einen Verweis auf TNodes haben, ergibt sich eine zyklische Abhängigkeit, weshalb TTypeQualifier mehrmals betrachtet werden müssen.

Knoten	NodeId
IMLGraph (Root-Knoten)	1
Attribute Registry	2
SystemNode	3
Initialization Call	4
Initialization Code	5
TNode für TVoid	6
Initialization Code	7
Initialization Statements	8
Routine Conv	9
Pre Call	10
Post Call	11
TPointer auf TRoutine	12
TRoutine ohne Parameter, mit Rückgabewert void	13
EntityLValue (Operand der Routine Conv)	14
LinkerToolInfo	15
Linker Name	16

Tabelle 4.1: Vom Linker statisch vergebene Ids

TPointer TPointer werden anhand des Typs unifiziert, auf den sie zeigen. Dadurch, dass dies ein TNode ist, ergibt sich eine zyklische Abhängigkeit, weshalb TPointer mehrmals betrachtet werden müssen.

4.3 Anlegen von Objekten

Im Anschluss an die Unifikation steht fest, welche Knoten benötig werden. In einem Durchlauf werden alle Knoten erstellt, ohne irgendwelche Eigenschaften der Originalknoten zu übernehmen. Dies ist nötig um sicherzustellen, dass beim Kopieren von Eigenschaften alle verwendeten Knoten schon definiert sind.

Von allen Knoten, die in der Unifikationstabelle vorkommen, wird dabei nur der jewilige Gewinner angelegt. Knoten, die nicht in der Tabelle vorkommen, sind Knoten die nicht unifiziert werden müssen und deshalb direkt in die Ausgabedatei geschrieben werden können.

Es ist zwar nirgendwo spezifiziert, wie die Ids vergeben werden sollen, oder in welchem Bereich sie liegen müssen. Allerdings vergibt die bisherige Implementierung für bestimmte Knoten immer die selben Ids, worauf sich einige Werkzeuge evtl. verlassen. Die fest definierten Ids können in Tabelle 4.1 nachgelesen werden.

Der neue Linker vergibt dieselben festen Ids, um mögliche Inkompatibilitäten zu vermeiden. Jedem andersweitig erstellten Knoten wird eine eindeutige Id zugewiesen. Die Ids werden aufsteigend, beginnenend bei 17 vergeben.

4.4 Kopieren von Eigenschaften

Danach werden die Eigenschaften der Knoten übertragen. Dies erfolgt, falls der Originalknoten nicht in der Unifikationstabelle zu finden ist, oder wenn er in der Tabelle selbst der Gewinner ist. Dies ist wichtig, da bei ONodes mehrere Instanzen vorhanden sein können, wobei nur der Gewinner über die vollständigen Eigenschaften verfügt.

4.5 Finalisierung

Im letzten Schritt werden einige Strukturen angelegt, um für Bauhaus gültige Graphen zu erstellen. Diese werden von dem alten Linker statisch angelegt und werden daher übernommen.

TVoid

Fall kein TVoid in dem gelinkten Programm vorkommt, wird ein TVoid erstellt. In jedem Fall hat TVoid immer die Id 6.

TRoutine

Es gibt immer einen TRoutine-Knoten mit Rückgabewert void und keinen Parametern. Er hat die Id 13.

TPointer

Es gibt immer einen TPointer-Knoten für den zuvor angelegten TRoutine-Knoten mit der Id 12.

ToolInfo

In der ToolInfo werden Informationen zum verwendeten Linker und die übergebenen Parameter beim Aufruf gespeichert. Der Name des Linkers hat dabei die Id 16.

System-Knoten

Dieser Knoten ist der allgemeine Einstiegspunkt in den Graphen. Er wird mit der Id 3 angelegt. Er enthält eine Liste der übersetzten Einheiten und Listen mit Initialisierungs- und Finalisierungsmethoden. Außerdem enthält er eine Liste mit allen ausgeführten Werkzeugen, beginnend mit dem Linkereintrag.

4.5.1 Initialisierungs- und Finalisierungsmethoden

Hier werden die Intialisierungen und Finalisierungen aller Units aufgerufen.

4.5.2 Attribute Registry

Aus dem Code des alten Linkers ist nicht ersichtlich, was dieser Knoten repräsentieren soll. Bei der Untersuchung von einigen konvertierten zu `.iml.sf` `.iml`-Dateien konnte kein Knoten mit der Id 2 gefunden werden. Da nicht klar ist, um was es sich hier handelt wurde dieser Knoten ignoriert.

4.6 Ergebnis

Alle für den Linker relevanten Arbeitsschritte wurden an einer Stelle gebündelt. Das vereinfacht das Verstehen der Vorgehensweise des Linkers erheblich. Durch die Verwendung von Java und der Nutzung von HashMaps stieg die Ausführungszeit aber an.

4.6.1 Performance

Die Ausführungszeit des Linkens steigt durch die Portierung um den Faktor 10. Für kleinere Programme ist der Unterschied kaum zu bemerken, bei größeren Projekten dagegen sehr deutlich.

Anzumerken ist, dass der neue Linker deutlich mehr Operationen an den Dateien durchführt. So muss für jede Eingabedatei die Funktionalität von `imlstrip` ausgeführt werden und zum Abschluss die Datei bereinigt werden. Dies wirkt sich negativ auf die Laufzeit aus.

Bei der Portierung wurde darauf geachtet, den Code möglichst einfach und leicht verständlich zu halten. Außerdem werden die unterschiedlichen Unifikationen nacheinander bearbeitet und es wird nicht darauf geachtet, einzelne Dateien nacheinander komplett zu bearbeiten.

Einer Implementierung des Linkers in perfomanterem C++ steht nichts im Wege. Durch die Verwendung von templates sollte es problemlos möglich sein, die verlorene Zeit wiedergutzumachen.

5 Cafe++

cafe++ ist das von Bauhaus verwendete C/C++-Front-End. Es basiert auf einem komerziellen Front-End, um den tatsächlichen Code zu parsen. Akutell verwendet cafe++ neben dieser komerziellen Bibliothek auch einige aus dem Bauhausprojekt. Im Rahmen der Masterarbeit soll die Anzahl der verwendeten Bibliotheken reduziert werden und dabei insbesondere die libiml Bibliothek ersetzt werden.

5.1 libiml

cafe++ ist ein C++-Programm, während libiml in Ada implementiert ist. Dieser Zustand erschwert das Verstehen des Programms und dadurch auch die Entwicklung. Durch die Umstellung der Binärrepräsentation der verwendeten Zwischensprache zu SKilL ist es möglich geworden, IML Bibliotheken in unterschiedlichen Sprachen zu erstellen.

Um die Komplexität und den Wartungsaufwand von cafe++ zu reduzieren, soll dabei die bisher verwendete Ada Bibliothek durch die generierte C++ Bibliothek ersetzt werden.

5.2 Vorgehensweise

Die Umstellung erfolgt in drei Arbeitsschritten:

1. Ändern der Typnamen
2. Erstellen der Methoden die bisher aus libiml stammten
3. Manuelle Anpassungen

Die Typ- und Klassennamen unterscheiden sich zwischen libiml und SKilL hauptsächlich dadurch, dass in SKilL die Unterstriche wegfallen. Für jede Klasse, die es in libiml gibt, gibt es auch eine Klasse in SKilL. Dadurch ist die Umstellung mithilfe einer Liste der Klassen und dem UNIX-Tool sed sehr einfach.

cafe++ ruft für jeden Kontentypen eine Funktion iml_make_TYP_NAME mit unterschiedlichen Parametern auf. Diese müssen manuell erstellt werden, da vor allem die Reihenfolge und Typen der Parameter korrekt zugeordnet werden müssen. An einigen Stellen konnten die Parameter erst nach Suchen in libiml eindeutig zugeordnet werden.

Einige Methoden aus libiml, zum vorübergehenden Speichern von Werten in Listen, werden der Einfachheit halber mit Makros realisiert. Die Listen werden aus Performanzgründen angelegt und danach vollständig in die Ausgabedatei geschrieben. Ob dies mit SKilL immer noch einen

Perfomanzunterschied verursacht, wurde nicht untersucht. Da die Anzahl der Elemente oft schon feststeht und SKilL für Listen std::vector verwendet, kann darauf evtl. komplett verzichtet werden.

5.3 Inkompatible Änderungen

Bei der Umstellungen werden einige Änderungen vorgenommen, die Funktionalität entfernen. Diese werden in diesem Abschnitt vorgestellt.

Für alle Dateioperationen werden auch in dem C/C++ Code Methoden aus einer Bauhaus-Bibliothek verwendet, um cafe++ plattformunabhängig ausführen zu können.

An dem Institut arbeiten alle Entwickler auf Linux Rechnern, und die Tests werden auch nur in einer Linuxumgebung durchgeführt. Aus diesen Gründen, und auch weil keine Windows Umgebung zum Zeitpunkt der Entwicklung zur Verfügung stand, wurde entschieden vorerst nur Linux zu unterstützen. Falls in Zukunft andere Plattformen auch unterstützt werden sollen, empfiehlt es sich, eine weitverbreitete Bibliothek zu verwenden, welche platformunabhängige Dateioperationen bietet. Nennenswerte Optionen sind hier Qt 5.10 [QT] und Boost C++ [BOOST]. Die Boost-Bibliothek stellt für Bauhaus wahrscheinlich die bessere Wahl dar, da sie wesentlich kompakter ist und unter einer MIT-ähnlichen Lizenz vertrieben wird.

Nicht umgestellt wurden Aufrufe zum Lesen von Konfigurationsoptionen. Dies betrifft nur die Wahl der Dateiendungen der temporären Dateien. Standardmäßig werden .i und .ii für C und C++ verwendet. Durch Einstellen in der cafeCC.config war es möglich, diese Endungen zu ändern. Die Konfiguration wird in einer XML-Datei gespeichert. In dieser Konfigurationsdatei gibt es auch noch andere Optionen, welche die Ausführung von cafe++ beeinflussen. Diese werden allerdings als Parameter von cafeCC an cafe++ übergeben.

Eine Umstellung dazu, die Konfigurationsdatei in C++ zu lesen, wäre am sinnvollsten mithilfe einer Bibliothek zu lösen. Wobei fraglich ist, ob dies für diese beiden Optionen überhaupt nötig ist. Insebesondere weil cafe++ keine eigene Konfigurationsdatei hat und die Optionen deshalb in der Konfigurationsdatei von cafeCC definiert werden.

Falls diese Optionen weiterhin zur Verfügung gestellt werden sollen, ist es wesentlich einfacher diese auch als Parameter an cafe++ zu übergeben.

5.4 Schwierigkeiten bei der Umstellung

Die Umstellung barg einige unerwartete Probleme. So ist der vorhandene C++-Code nicht immer korrekt und lässt sich nicht ohne Modifikation von gcc kompilieren. Da die verwendete SKilL-API Features aus neueren C++-Versionen verwendet, kann cafe++ nicht mit dem Standardkompiler auf den bereitgestellten Servern gebaut werden. Bisher wird eine bestimmte Version des GNAT Compilers für alle Bauhauswerkzeuge verwendet. Die bereitgestellte Version unterstützt aber nicht aktuelle C++ Standards. Nach der Umstellung ist aufgefallen, das GNAT nach dem C++-Standard ungültige Probleme scheinbar tolleriert und diese erst bei der Verwendung eines anderen Compilers sichtbar werden.

Gravierender ist das Problem, dass in libiml, welches in Ada implementiert ist, Arrays nicht immer mit der Indexposition 0 beginnen. Ada erlaubt das Verwenden von beliebigen Indextypen für Arrays. Dadurch beginnt die Zählung in dem C++ Code auch teilweise erst bei 1. Dies führt teilweise zu Abstürzen, weil der letzte Index nach der Umstellung um eins niedriger ist. Alle offensichtlich betroffenen Stellen werden angepasst. Da auch beim Einlesen von größeren Programmen wie GNU Bash keine Speicherfehler beobachtet werden, ist es unwahrscheinlich, dass bei der Umstellung Stellen übersehen wurden.

Hier ist anzumerken, dass die Möglichkeit variable Arrayindizes zu verwenden sehr nützlich sein kann. Wenn die Indizes allerdings über eine Schnittstelle auch an andere Programmiersprachen weitergereicht werden können, sollte darauf verzichtet werden, unübliche Bereiche zu verwenden. Insbesondere wenn diese Programmiersprachen andere Konventionen verwenden, wie hier Ada und C++. Das Nichtbeachten kann fehlerhafte Programme zur Folge haben, in jedem Fall begünstigt es Missverständnisse und führt zu schwer zu verstehenden Code.

5.5 Ergebnis

Durch die Umstellung wächst die Größe von `cafe++` von 3.1 MB auf 13 MB. Allerdings wird `libiml` nicht mehr gebraucht, welche mehr als 45 MB groß ist. Insgesamt ergibt das eine Einsparung von mehr als 30 MB, was ca. 65% entspricht.

Mit der Umstellung ist `cafe++` ein eigenständiges Tool, welches keine externen Abhängigkeiten zu anderen Programmiersprachen hat. Dadurch ist es wesentlich einfacher, das Programm zu warten, weil der gesamte Code von einer C/C++-IDE geladen werden kann. Durch die Umstellung des Compilers von GNAT zu cmake, sollte das kompilieren des Projekts auch wesentlich einfacher sein.

Insbesondere ist nicht mehr die komplette Bauhausumgebung notwendig, um `cafe++` auszuführen.

6 IML Strip

Das Programm `imlstrip` wird von `cafeCC` aufgerufen. Sinn des Programms ist es, ungültige Einträge in den DeclarationTable und UnresolvedDeclarations von Units zu löschen. Ungültig sind dabei Einträge, die auf Knoten verweisen, die von dem Wurzelknoten des Graphen auf anderem Weg unerreichbar sind.

6.1 Aktuelles Programm

Die aktuelle Version des Programms ist nicht korrekt. Sie löscht alle Einträge aus DeclarationTable und UnresolvedDeclarations von Units.

Das Programm merkt sich die vorhandenen Knoten in DeclarationTable und UnresolvedDeclarations. Anschließend werden in einer Schleife alle gespeicherten ONodes betrachtet und überprüft, ob diese sich in der gemerkten Menge befinden. Die gefundenen ONodes werden der Unit hinzugefügt. Die Schleife wird abgebrochen, wenn keine gesuchten ONodes mehr gefunden werden. Listing 6.1 zeigt in einer Python-ähnlichen Sprache die Vorgehensweise des alten Programms.

Der Fehler im Programm liegt an der Umstellung der IML Bibliothek zu `SKilL`. Die alte verwendete Funktion kann nur auf die Daten zugreifen, wenn es Verweise darauf gibt, während die `SKilL`-basierte IML Bibliothek alle Knoten zurückgibt, da sie nicht auf Verweise angewiesen ist.

Listing 6.1 Aktuelles Verhalten von `imlstrip`

```
Declarations = Unit.DeclarationTable.entries
Unresolved = Unit.UnresolvedDeclarations.entries
FoundDeclarations = set()
FoundUnresolved = set()
Unit.DeclarationTable.clear()
Unit.UnresolvedDelarations.clear()
while True:
    for node in Unit.ONodes():
        if Declarations.contains(node):
            Declarations.remove(node)
            FoundDeclarations.add(node)
        if Unresolved.contains(node):
            Unresolved.remove(node)
            FoundUnresolved.add(node)
    if FoundDeclarations.isEmpty() and FoundUnresolved.isEmpty():
        exit
    Unit.DeclarationTable.addAll(FoundDeclaration)
    FoundDeclarations.clear()
    Unit.UnresolvedDeclarations.addAll(FoundUnresolved)
    FoundUnresolved.clear()
```

Das erklärt, weshalb die UnresolvedDeclarations immer leer sind. Weshalb DeclarationTables auch geleert werden, konnte nicht erklärt werden. Da die Aufgabe von `imlstrip` klar ist, wird mit der Portierung fortgefahren, ohne einen Grund für das Fehlen der DeclarationTable-Einträge zu suchen.

Die davon betroffene Funktion ist `Storables.Process_All_Nodes`. Bei der Umstellung wurde die interne Funktionalität der Funktion verändert, ohne dass dies sichtbar gemacht wurde.

Es ist unwahrscheinlich, dass andere Programme von der Umstellung betroffen sind. Der Fehler tritt bei `imlstrip` auf, weil hier offensichtliche Fehler im Graphen beseitigt werden.

Nichtsdestotrotz sollten alle Programme überprüft werden, die diese Funktion verwenden. Da die Methode außerdem Bestandteil einer öffentlichen API ist, sollte die Umstellung dokumentiert werden und evtl. eine Funktion bereitgestellt werden, welche das alte Verhalten aufweist.

6.2 Neues Programm

Die Funktionalität von `imlstrip` wurde im Linker als eigenständiges Modul implementiert. Der Code ist vollständig vom Linker getrennt und wird beim Programmstart des Linkers auf allen Eingabedateien ausgeführt, bevor der eigentliche Linkvorgang beginnt.

Wichtig ist noch zu erwähnen, dass vor dem Linkvorgang alle unerreichbaren Knoten gelöscht werden. Erreichbar sind hierbei alle Knoten, die von dem IMLGraph-Knoten entweder direkt oder indirekt erreichbar sind.

Die Menge der Erreichbaren ONodes wird dadurch gewonnen, alle Verweise von Knoten zu überprüfen. Wenn ein ONode gefunden wird, wird dieser zu der Menge der gefundenen Knoten hinzugefügt. Wichtig ist dabei, darauf zu verzichten die Enumeration von ONodes, die durch `SKilL` bereitgestellt wird, zu verwenden. Diese gibt nämlich alle in der `.iml.sf`-Datei gespeicherten ONodes zurück. Explizit ausgeschlossen werden auch die ONodes, die in DeclarationTable und UnresolvedDeclarations referenziert sind.

Nach diesem Durchlauf werden alle Verweise von den gefundenen ONodes untersucht und diese auch der gefundenen Menge hinzugefügt.

Der letzte Schritt besteht darin, alle ONodes aus DeclarationTable und UnresolvedDeclarations zu entfernen, die nicht in der gefundenen Menge enthalten sind.

In Listing 6.2 ist das Vorgehen in Python-ähnlichem Pseudocode dargestellt.

6.3 Ergebnis

Die Funktionalität von `imlstrip` wird vorübergehend in dem Linker als eigenes Modul bereitgestellt. Bei einer Überarbeitung von `cafe++` sollte untersucht werden, weshalb diese ungültigen Einträge überhaupt entstehen und wie diese verhindert werden können.

Listing 6.2 Neue Implementierung von imlstrip

```
seen = set()

def handleNode(node, type):
    for field in in type.fields()
        if field == DeclarationTable or field == UnresolvedDeclarations:
            continue
        if field is containerType:
            for entry in node.get(field):
                if entry is ONode:
                    seen.add(entry)
        else if node.get(field) is ONode
            seen.add(node.get(field))

def prune(container):
    for node in container:
        if not seen.contains(node):
            container.remove(node)
            node.delete()

for type in allTypes:
    if type is ONode:
        continue
    for node in type.nodes():
        process(node, type)

for node in seen:
    process(node, node.type())

prune(Unit.DeclarationTable)
prune(Unit.UnresolvedDeclarations)
```

7 IML Optimizer

Das Programm `imloptimizer` wird nach dem Linker aufgerufen und entfernt spezielle Konstrukte aus der Ausgabedatei. Dabei kann das Programm mit verschiedenen Parametern aufgerufen werden, um zu steuern welche Operationen durchgeführt werden sollen. Aufgerufen wird das Programm von `cafeCC` ohne spezielle Parameter. Daher werden immer die Standardwerte verwendet. Die einzige Möglichkeit, das Verhalten von `imloptimizer` zu beeinflussen, ist `cafeCC` anzuweisen, keine Optimierung durchzuführen und stattdessen `imloptimizer` selbst aufzurufen.

Leider stand keine Dokumentation des Programms zur Verfügung. Insbesondere gibt es keine Erklärung, warum genau diese Operationen zur Verfügung stehen und was für einen Zweck sie erfüllen sollen.

Die einzige Grundlage für die Aussagen in diesem Kapitel sind daher die aktuelle Implementierung des Programms und das Wissen über die Strukturen des Graphen.

In den Unterkapiteln wird auf die bisher implementierten Operationen eingegangen.

7.1 Entfernen von Deklarationen, die keine Definitionen sind

Die Optimierung besteht darin, ONodes aus dem Graphen zu entfernen, die nur eine Deklaration, nicht aber eine Definition repräsentieren.

In dem neu entwicketlen Linker sind dies nur Deklarationen zu denen keine Definition zu finden war. Dies kann unter anderem der Fall sein, wenn eine Datei bei dem Linken vergessen wird. Eine andere Möglichkeit keine Definitionen zu haben ist, wenn für bestimmte Pfade keine Definitionen geladen werden sollen. Diese Pfade können in der Konfigurationsdatei von `cafeCC` hinterlegt werden. In der Standardkonfiguration enthält die Option nur die include-Pfade auf dem System (`/usr/include/`).

Weshalb diese Knoten entfernt werden sollen, ist nicht dokumentiert. Diese generell zu entfernen, scheint äußert fragwürdig zu sein. Aus diesem Grund wird diese Optimierung nicht übernommen.

7.2 Entfernen von Dereferenzierung von AddressOf Operatoren

Diese Optimierung wird standardmäßig ausgeführt.

Die Konstrukte sind laut dem C11 Standard [ISO11a] §6.5.3.2 gültig. Allerdings erwähnt der Standard auch, dass diese nicht tatsächlich ausgewertet werden sollen. Ein Compiler sollte für diese Konstrukte keinen Code generieren. Sie können aber verwendet werden, um sicherzustellen,

Listing 7.1 deref.c - Derferenzierung von AddressOf

```
int main() {
    int value = 10;
    int* ptr = &value;
    int* ptr2 = *&ptr;
    return *ptr2;
}
```

dass dem Ausdruck kein Wert zugewiesen wird und zum Prüfen, ob der Ausdruck ein gültiger Pointertyp ist. Beide Bedingungen können vom Compiler statisch geprüft werden und sind während der Laufzeit nicht mehr relevant.

Aus diesem Grund sollte diese Optimierung nicht erst nach dem Linken stattfinden. Bei genauer Betrachtung der resultierenden `.iml.sf`-Datei für Programm 7.1 kann festgestellt werden, dass diese Konstrukte tatsächlich von dem von `cafe++` verwendeten C/C++-Front-End entfernt werden. In Zeile 3 wird ein AddressOf-Objekt erstellt, in Zeile 5 ein Dereference-Objekt. In Zeile 4 wird keines der Objekte erstellt. Stattdessen wird direkt ein Read eines EntityLValue erstellt. Die AdressOf und Dereference-Objekte werden erst gar nicht an `cafe++` weitergegeben.

Da es nichts zu entfernen gibt, wird diese Optimierung nicht portiert.

7.3 Entfernen von Null-Elementen in StatementSequences

Diese Optimierung wird standardmäßig ausgeführt.

Die Optimierung besteht darin, Null-Zeiger aus der Liste von Values zu entfernen. Warum diese überhaupt entstehen, ist nicht dokumentiert. Für den Linker sind sie nicht relevant und werden nicht gesondert betrachtet. In `cafe++` werden diese Listen zwar erstellt, aber es werden nie Null-Pointer hinzugefügt. Allenfalls werden NullExpressions eingefügt, welche gültige Knoten in der IML Struktur sind.

Von einer Portierung dieser Optimierung wird daher abgesehen. Falls es wider erwarten doch Null-Einträge in StatementSequences geben sollte, sollten diese in `cafe++` entweder durch NullExpressions ersetzt oder überhaupt nicht gespeichert werden.

7.4 Entfernen von MangledNames

Bei dieser Optimierung werden von allen ONodes die MangledName-Einträge entfernt. Diese Information ist hauptsächlich für den Linker relevant, um eindeutige Bezeichner für unterschiedliche Konstrukte mit dem selben einfachen Namen zu haben. Durch Klassen, Namespaces oder ähnliches werden die einfachen Namen zu einzigartigen erweitert. Nachdem der Linker aufgerufen wird, werden diese Bezeichner nicht mehr benötigt, da alle nötigen Elemente gelinkt wurden und die Auflösung schon eindeutig ist.

Daher kann diese Optimierung als eine reine Speicheroptimierung angesehen werden. Die Implementierung ist denkbar einfach, da sich die Änderungen immer auf einen Knoten beschränken, ohne Abhängigkeiten auf andere Knoten.

Die Funktionalität wird in den neuen Linker integriert. Die Optimierung bleibt standardmäßig deaktiviert, kann aber via Parameter aktiviert werden.

Zu erwähnen ist, dass der alte Linker via Parameter dazu aufgefordert werden kann, MangledNames zu entferenen. Weshalb dies Funktionalität zusätzlich in `imloptimizer` zur Verfügung steht ist nicht dokumentiert.

7.5 LRConversion-Knoten entfernen

Die bisherige Implementierung entfernt alle LRConversions und fügt eine direkte Kante von dem Parent-Knoten zu dem Operand-Knoten ein. Dies wird für alle vorhandenden Knoten durchgeführt. Einzige Bedingung hierbei ist, dass der Parent-Knoten existiert.

Weshalb diese “Optimierung” angeboten wird, ist unklar. Es ist nicht ganz klar was LRConversions repräsentieren und wie sie verwendet werden. Alle diese Knoten aus dem Graphen zu entfernen scheint kein korrektes Verhalten zu sein.

Daher wird diese Funktion nicht übernommen. Falls solch eine Funktion in der Zukunft gebaucht werden sollte, kann der originale Code vom `imloptimizer` bei einer Portierung verwendet werden.

7.6 Ergebnis

Das Entfernen von MangledNames wird in den Linker integriert, alle anderen Funktionen entfallen.

8 Test

In diesem Kapitel wird die Vorgehensweise beim Testen der Linkerimplementierung vorgestellt. Laut Aufgabenstellung der Arbeit soll die Funktionsfähigkeit des gesamten Front-Ends durch Vergleich mit den alten Implementierungen gezeigt werden.

Außerdem soll ein Satz an Testprogrammen ausgewählt werden, mithilfe derer sich die Funktionsfähigkeit nachgelagerter Werkzeuge prüfen lässt.

8.1 Testdaten

Zum Testen werden die in Tabelle 8.1 genannten Programme verwendet und zusätzlich eigene geschrieben. Die bereitgestellten Programme werden in den Abschnitten 8.2, 8.5 und 8.6 verwendet. In Abschnitt 8.4 werden die eigenen Programme verwendet.

8.2 Test anhand der Anzahl der Knoten

Um während der Entwicklung die Korrektheit der Implementierung zu prüfen, wird ein Testprogramm entworfen, welches die Anzahl der Knoten in IML-Graphen vergleicht. Da die Hauptaufgabe des Linkers darin besteht, Knoten zu unifizieren, scheint die Anzahl der Knoten eine sinnvolle Metrik zu sein.

Da alle zum Front-End gehörenden Programme deterministisch sein sollten und immer die gleichen Ausgaben generieren sollten, sollte auch die Anzahl der Knoten stabil bleiben. Erst nach dem Linken können unterschiedliche Programme verwendet werden, die in der Lage sind Ergebnisse unter Umständen in der Datei abzuspeichern.

Dabei werden auf der einen Seite vollständig gelinkte `.iml`-Dateien vom bisherigen Linker verwendet, auf der anderen Seite von `cafe++` erstellte `.iml.sf`-Dateien, welche vom neuen Linker verwendet werden, um die Ausgabedatei zu erstellen. Ziel ist es, identische Anzahlen der Objekte zu erhalten.

Programm	Version
Aget	0.4
make	3.75
darkhttpd	1.2

Tabelle 8.1: Bereitgestellte Testdaten

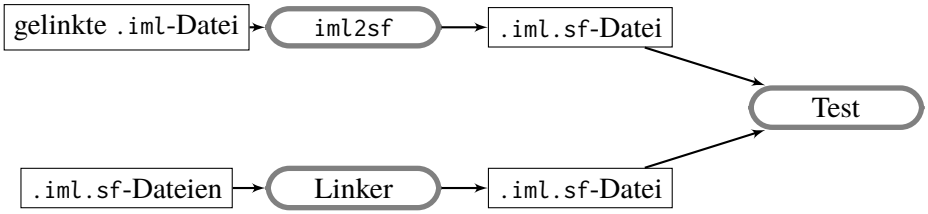

Abbildung 8.1: Testaufbau

Das Werkzeug `iml2sf` wird verwendet, um die alten gelinkten `.iml`-Dateien in das neue Format zu übertragen.

Die Eingabedateien für den neuen Linker sind hierbei nicht die Programmdateien, da dafür eine vollständige Bauhausumgebung notwendig wäre. In Abbildung 8.1 wird die Vorgehensweise veranschaulicht.

8.3 Interpretation der Ergebnisse

Es wurde festgestellt, dass die Anzahl der Knoten teilweise massiv voneinander abweichen. Teilweise wurden diese Ergebnisse erwartet, da z.B. die aufgerufenen Programme in der IML-Datei hinterlegt sind. Sobald sich diese ändern, ändert sich die Anzal der Identifier ganz automatisch. Da für die alten Dateien `imlstrip` und `imloptimizer` aufgerufen wurden, was mit dem neuen Linker nicht mehr geschieht, gibt es eine natürliche und vorhersehbare Abweichung in der Anzahl der Identifier.

Allerdings ist die Abweichung der Anzahl der Identifier wesentlich größer als erwartet. Auch gibt es Abweichungen, die nicht direkt erklärt werden können, wie z.B. Abweichungen bei der Anzahl der SLocs und SLocPositions.

Da die Unifikation von SLocs, SLocPositions und Identifier trivial ist, wird davon ausgegangen, dass der alte Linker diese nicht korrekt oder gar nicht unifiziert. Für diese Knoten ist eine Unifikation nicht zwingend notwendig, da eigentlich keine negativen Auswirkungen zu erwarten sind. Ein Nachteil ist zusätzlich benötigter Speicher. Das kann dazu führen, dass Berechnungn unnötigerweise mehrfach für identische Knoten durchgeführt werden.

Bei näherer Untersuchung fällt jedoch auf, dass unter anderem auch die Anzahl der OEnumerations und OEnums abweichen. Bei manueller Untersuchung der erstellten IML-Dateien mithilfe von skillview [Rat17] konnte ermittelt werden, dass der alte Linker diese entweder gar nicht oder nur unvollständig unifiziert. Details dazu sind in dem Unifikationskapitel zu finden. Anders als bei SLocs und Identifier werden hier wesentliche Bestandteile vom alten Linker nicht korrekt unifiziert.

Diese Beobachtungen führen zu dem Schluss, dass ein Test basierend auf der Anzahl der Knoten falsche Ergebnisse liefern wird.

Da die Funktionsweise des alten Linkers nicht besonders übersichtlich ist, wird die Entscheidung nach der Herkunft der Abweichungen erschwert. Insbesondere kann nicht ausgeschlossen werden, dass sich Fehler im neuen Linker befinden, welche sich auch im alten befinden, sowie Fehler welche sich ausschließlich im Neuen befinden, aber aus Gründen der Unübersichtlichkeit des alten Linkers fälschlicherweise als Fehler im alten Linker betrachtet werden.

Daher wird ein Test entworfen, welcher nicht auf den direkten Vergleich der Linker basiert, sondern nur auf dem neuen Linker basiert.

8.4 Linker Tests

Den Linker anhand von Unittests zu testen scheint wenig sinnvoll, da es wenige gut trennbare Methoden im Linker gibt.

Den Linker anhand der Anzahl der Knoten zu testen erscheint daher trotz der Ungültigkeit beim Vergleich mit dem alten Linker immer noch sinnvoll. Daher wurde ein Satz an C/C++-Programmen geschrieben, welche in denen Konstruktue vorkommen, die der Linker unifizieren sollte. Dabei wird die Anzahl der Knoten mit vorher definierten Ergebnissen verglichen.

Dadurch dass die Tests direkt auf Programmdateien basieren, können sie auch gut bei weiteren Portierungen des Linkers verwendet werden.

8.5 Test anhand der Ausgabe von functionnames2

`functionnames` ist ein einfaches Programm, welches die Namen der deklarierten Methoden auflistet. Bei einem Vergleich des neuen und alten Linkers anhand der bereitgestellten Programme war nur die Reihenfolge der Auflistung ist unterschiedlich. Durch die Umstrukturierungen in `SKilL` ist die Reihenfolge nicht mehr fest, dies sollte keine negativen Effekte haben.

8.6 Test anhand Ausgabe von ccdiml

Das Tool ccdiml wird zum Testen gewählt, weil es zu den Tools gehört, die schon auf den `SKilL` basierten IML-Dateien arbeiten können. Außerdem verfügt es über eine Testsuite, welche darauf basiert, C Programme zu kompilieren und daraufhin die Ausgabe von ccdiml mit den erwarteten Ergebnissen zu vergleichen.

Da die Ausgabe von ccdiml einfach zu überprüfen ist, können auch weitere Programme auf die selbe Weise getestet werden. Bei Abweichungen zwischen dem alten und dem neuen Linker kann leicht überprüft werden, welche Ausgabe korrekt ist.

Es ist anzumerken, dass die Testsuite von ccdiml aus einzelnen Dateien besteht. Hier wird die Funktionalität des Linkers gar nicht gebraucht, daher musste die Ausgabe mit größeren Programmen untersucht werden.

Es konnten keine Änderungen festgestellt werden.

8.7 Test von cafe++

cafe++ verfügt über eine Testsuite, welche darauf beruht, C++ Programme mit cafe++ einzulesen und anschließend mit nachgelagerten Werkzeugen aus Bauhaus zu analysieren. Die Ergebnisse werden mit vorher definierten Werten verglichen. Leider musste festgestellt werden, dass die aktuelle Ada SKilL-API einige Methoden noch nicht unterstützt, welche von den nachgelagerten Werkzeugen verwendet werden. Dadurch war es nicht möglich, die Funktionalität von cafe++ anhand der vorhandenen Testsuite zu testen.

Dieser Misstand ist bisher noch nicht aufgefallen, weil cafe++ bisher noch nicht automatisiert getestet wurde. Mit den vorgestellten Änderungen, sollte es einfacher sein dies in Zukunft zu realisieren.

9 Offene Punkte

In diesem Kapitel werden die offenen Punkte zusammengefasst. Diese umfassen vorgeschlagene Optimierungen und Änderungen an der verwendeten SKiLL-Spezifikation.

9.1 C/C++-Front-End

Die von cafe++ verwendete Bibliothek zum Einlesen von C/C++-Dateien unterstützt keine neuen Versionen dieser Sprachen. Um mit Bauhaus neuere Programme analysieren zu können, sollte eine neuere Version der Bibliothek verwendet werden.

9.2 cafe++

Bei der Migration wurden alle neu eingeführten Funktionen in einem separaten Header definiert. Einige dieser Methoden können mit mehr Wissen über die Arbeitsweise von cafe++ unter Umständen vereinfacht werden. Bei der Arbeit an dem Werkzeug wurden einige Kommentare gefunden, die auf eine nicht vollständig abgeschlossene Entwicklung des Programms hinweisen.

9.3 imlstrip

Die Funktionalität von imlstrip steht zwar wieder zur Verfügung, aber es ist weiterhin unklar, weshalb ungültige Einträge in DeclarationTable und UnresolvedDeclarations in Units vorkommen. Es sollte untersucht werden, weshalb cafe++ diese anlegt.

9.4 Linker

Der Linker wurde in Java implementiert, was einige Vorteile bei der Entwicklung und dem Testen mit sich führte. Um die Ausführungszeit zu optimieren, sollte der Linker in C++ implementiert werden.

9.5 imloptimizer

Es ist unklar weshalb die Operation zur Entfernung von LRConversion-Knoten bereitgestellt wurde. Es sollte untersucht werden, wann genau solche Knoten entstehen und weshalb es sinnvoll sein könnte, diese zu entfernen.

9.6 Spezifikation

In Abschnitt 4.2 werden einige Änderungen an der Spezifikation vorgeschlagen. Wenn diese übernommen werden sollten, müsste wahrscheinlich ein Großteil der Werkzeuge in Bauhaus angepasst werden.

Die fehlende Dokumentation zur Spezifikation führt dazu, dass nicht immer klar ist, was durch die Knoten modelliert wird. Um Arbeiten an Bauhaus zu vereinfachen, sollte bei der Anpassung der Spezifikation damit begonnen werden, diese zu dokumentieren.

10 Zusammenfassung

Der Zweck dieser Arbeit ist eine algemeine Überarbeitung des Front-Ends von Bauhaus. Dafür wurde der Linker von Grund auf neu geschrieben. Die Funktionalität der Werkzeuge `imlstrip` und `imloptimizer` wurde in den größtenteils in den Linker integriert. `cafe++` wurde angepasst, so dass nur noch eine C++-`SKilL`-API verwendet wird und keine Ada Bibliotheken mehr notwendig sind.

In Kapitel 3 wird die Funktionsweise des alten Linkers untersucht. Dabei wurden einige Eigenheiten des bisherigen Front-Ends gefunden, welche nach der Migration zu `SKilL` nicht mehr so benutzbar sind.

Kapitel 4 behandelt die neue Implementierung des Linkers. Die Knoten die für die Unifikation werden vorgestellt und die Regeln anhand denen unifiziert wird besprochen. Es wurde beobachtet, dass der alte Linker einige Strukturen nicht korrekt unifiziert. Diese werden im neuen Linker korrekt behandelt. Bei der Behandlung von TRoutines ist aufgefallen, dass die Spezifikation nicht korrekt ist. Diese sollte dringend angepasst werden.

In Kapitel 5 werden die Änderungen vorgstellt, die an `cafe++` vorgenommen worden sind. `cafe++` ist jetzt ein eigenständiges Programm, welches nicht mehr auf das vorhandensein der Bauhausumgebung angewiesen. Dies gilt für das compilieren des Werkzeugs und auch für die Ausführung. Durch das Wegfallen der Bauhaus-Schnittstellen, ist es wesentlich einfacher geworden daran zu arbeiten.

Kapitel 6 stellt die Funktionsweise des Werkzeugs `imlstrip` vor und bespricht wie dies in den Linker integriert worden ist.

Kapitel 7 behandelt die Optimierungen die von `imloptimizer` durchgeführt worden. Dabei wurde festgestellt, dass die meisten Optimierungen überflüssig sind. Das Entfernen von LRConversions konnte nicht begründet werden und wird daher nicht übernommen.

In Kapitel 8 werden die Ansätze vorgestellt, nach denen die unterschiedlichen Komponenten getestet werden. Es wird eine Strategie vorgestellt mit der der Linker langfristig getestet werden kann. Dabei wird der Linker nicht mit der alten Implementierung getestet.

10.1 Ausblick

Wie in Kapitel 9 erwähnt, sollte die aktuelle Modellierung überarbeitet und dabei auch dokumentiert werden, um weitere Arbeiten an Bauhaus einfacher durchführen zu können.

Abkürzungsverzeichnis

`.iml` IML Dateien vor der Migration zu skill.

`.iml.sf` IML Dateien nach der Migration zu skill.

`imllink` Der alte Bauhauslinker.

`cafe++` Das C/C++-Front-End von Bauhaus.

`imlstrip` Ein Werkzeug, dass bestimmte ungültige Knoten aus den Graphen entfernt.

`imloptimizer` Ein Werkzeug, um unterschiedliche Optimierungen an dem Graphen durchzuführen indem Knoten entfernt werden.

`iml2sf` Ein Werkzeug, um .iml.sf-Dateien zu visualisieren.

`cafeCC` Ein Werkzeug, um das gesamte Bauhaus-Front-End für ein Programm auszuführen.

`libiml` In Ada implementierte Bibliothek zur Interaktion mit IML-Dateien.

`SKilL` Serialization Killer Language.

Literaturverzeichnis

[BOOST] *Boost C++ Libraries*. URL: https://www.boost.org/ (zitiert auf S. 26).

[Fel14] T. Felden. „„Efficient and Change-Tolerant Serialization for Program Analysis Tool-Chains.““ In: *Softwaretechnik-Trends* 34.2 (2014) (zitiert auf S. 7).

[FW16] T. Felden, M. Wittiger. „Migrating Bauhaus from IML to SKilL“. In: *Softwaretechnik-Trends* 36.2 (2016) (zitiert auf S. 7).

[ISO11a] ISO/IEC. *C Programming Language*. 2011 (zitiert auf S. 15, 19, 33).

[ISO11b] ISO/IEC. *C++ Programming Language*. 2011 (zitiert auf S. 15).

[Prz16] D. Przytarski. „SKilLed Bauhaus“. Magisterarb. 2016 (zitiert auf S. 7).

[QT] The Qt Company. *Qt Framework*. URL: https://www.qt.io/ (zitiert auf S. 26).

[Rat17] M. Rathgeber. „SKilL-Graphvisualisierung und -manipulation“. 2017 (zitiert auf S. 38).

Alle URLs wurden zuletzt am 14.05.2018 geprüft.

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschrift