Karlsruhe Institute of Technology

Institut für Programmstrukturen
und Datenorganisation (IPD)

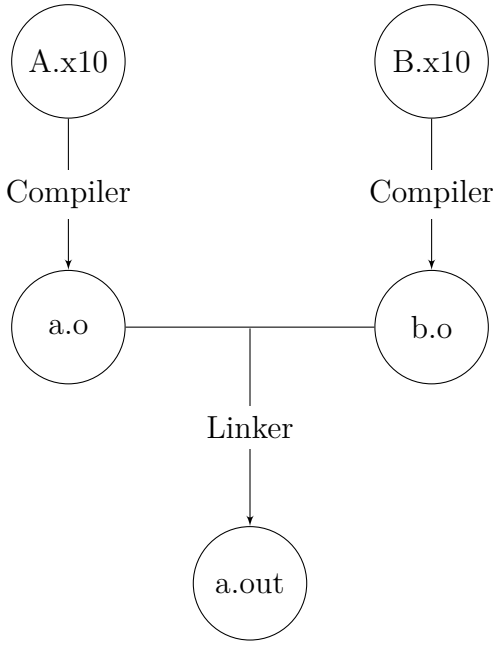
Lehrstuhl Prof. Dr.-Ing. Snelting

Erweiterung des invasiven X10-Compilers um getrennte Übersetzung

Bachelorarbeit von

Christoph Jost

an der Fakultät für Informatik



Erstgutachter:	Prof. Dr.-Ing. Gregor Snelting
Zweitgutachter:	Prof. Dr. Jörg Henkel
Betreuender Mitarbeiter:	Dipl.-Inform. Manuel Mohr

Bearbeitungszeit: 11. Juli 2015 – 10. November 2015

Zusammenfassung

In dieser Bachelorarbeit wird die Implementierung von getrennter Übersetzung für den X10-Compiler beschrieben. Mit dieser Erweiterung können Anwendungsprogramme und die X10-Standardbibliothek unabhängig voneinander übersetzt werden.
Hierbei wird insbesondere auf die statische Initialisierung von Feldern in X10, die Serialisierung von Objekten, sowie generische Klassen und Methoden eingegangen.
Abschließend konnte eine deutliche Compile-Zeit-Reduktion festgestellt werden.

This thesis describes the implementation of modular compilation for the invasive X10-compiler. With the enhancement it is possible to compile user programs and the x10-standard-libary independently.
The focus of the thesis is the static initialization of fields in x10, the serialization of objects, as well as generic classes and methods.
In the final result a considerable improvement of the compile-time-reduction could be noticed.

Inhaltsverzeichnis

1. Einführung

Der am IPD entwickelte invasive X10-Compiler [6] unterstützt in seiner Implementierung nicht die unabhängige Übersetzung mehrerer Übersetzungseinheiten. Dadurch kann die X10-Standardbibliothek nicht vorher übersetzt werden und muss bei jeder Übersetzung eines Anwendungsprogrammes mitübersetzt werden. Das Ziel dieser Arbeit ist es getrennte Übersetzung für den X10-Compiler umzusetzen. Mit dieser Erweiterung des Compilers soll die X10-Standardbibliothek vorübersetzt werden. Anschließend kann diese beim Übersetzen eines Anwendungsprogrammes genutzt werden um die Code-Generierung für Klassen aus der Standardbibliothek zu vermeiden.

In Kapitel 2 werden alle notwendigen Vorkenntnisse zur Umsetzung von getrennter Übersetzung erläutert. Anschließend wird in Kapitel 3 die konkrete Umsetzung diskutiert. In Kapitel 4 werden die Ergebnisse evaluiert und zu letzt in Kapitel 5 ein Fazit gezogen.

2. Grundlagen und Verwandte Arbeiten

2.1. Kompilierungsgrundlagen

Ein typischer Compiler erzeugt aus Quellcode eine ausführbare Datei. Innerhalb des Compiler selbst kommen nach der lexikalischen, syntaktischen und semantischen Analyse, durch das Frontend, oft noch ein Assembler und Linker zum Einsatz. In Abbildung 2.2 ist eine typische Compiler-Pipeline, inklusive der Ausgaben der einzelnen Compiler-Glieder, dargestellt. Der Präprozessor ist hierbei ein optionales Glied und führt eine Vorverarbeitung des Quellcodes durch. Für den konkreten Fall eines C-Compilers ist diese Vorverarbeitung eine reine Textersetzung. Aus dem Ergebnis des Präprozessor erzeugt der Compiler anschließend den Assemblercode.
Der erzeugte Assemblercode wird anschließend durch den Assembler verarbeitet, welcher daraus eine Objektdatei generiert. Dieser enthält architekturabhängigen Maschinencode. Eine Aufgabe des Assembler ist das Auflösen Labels. Ein Label ist ein vom Programmierer definierter Name, der eine eindeutige Position im Assemblercode markiert. Der Assembler wandelt den Sprung mit einem Label als Ziel in einen relativen Sprung um.
Auf den Assembler folgt der Linker, welcher für die Umsetzung von getrennter Übersetzung der wichtigste Teil der Compiler-Pipeline ist. Typischerweise wird ein Compiler vom Benutzer auf den kompletten Quellcode angewendet. Der Linker macht es möglich das Frontend und den Assembler gemeinsam auf Teile des Quellcodes anzuwenden und anschließend die erzeugten Objektdateien zu verknüpfen.
In Abbildung 2.1 ist der Quellcode einer Funktion `f`, die in der Quellcodedatei `a.cpp` definiert wird. Die Hauptfunktion (Mainfunktion) wird in der Quellcodedatei `b.cpp` definiert und ruft anschließend diese Funktion `f` auf.
Der Ablauf zum Erzeugen dieses Programmes ist ebenfalls in Abbildung 2.1 dargestellt. Durch das Deklarieren von `f` in `b.cpp` kann der Compiler den Typ von `f` bestimmen. Mit diesem ist er in der Lage, den Aufruf an `f` zu erzeugen ohne die Definition zu kennen. Der Compiler erwartet lediglich, dass das Symbol `f` in einer beliebigen Datei definiert wird. Der Linker erzeugt aus den übergebenen Objektdateien im letzten Schritt die ausführbare Datei indem er alle Referenzen auf externe Symbole auflöst. Durch diese Aufteilung der Aufgaben eines Compilers ist es möglich den Quellcode aufzuteilen und diese Teile unabhängig voneinander zu kompilieren. Zusätzlich ist der Linker in der Lage

```
// Definitionsdatei a.cpp
void f() {
    // Tu etwas ...
}
```

```
// Definitionsdatei b.cpp
// mit Hauptfunktion
extern void f();

int main() {
    f();
}
```

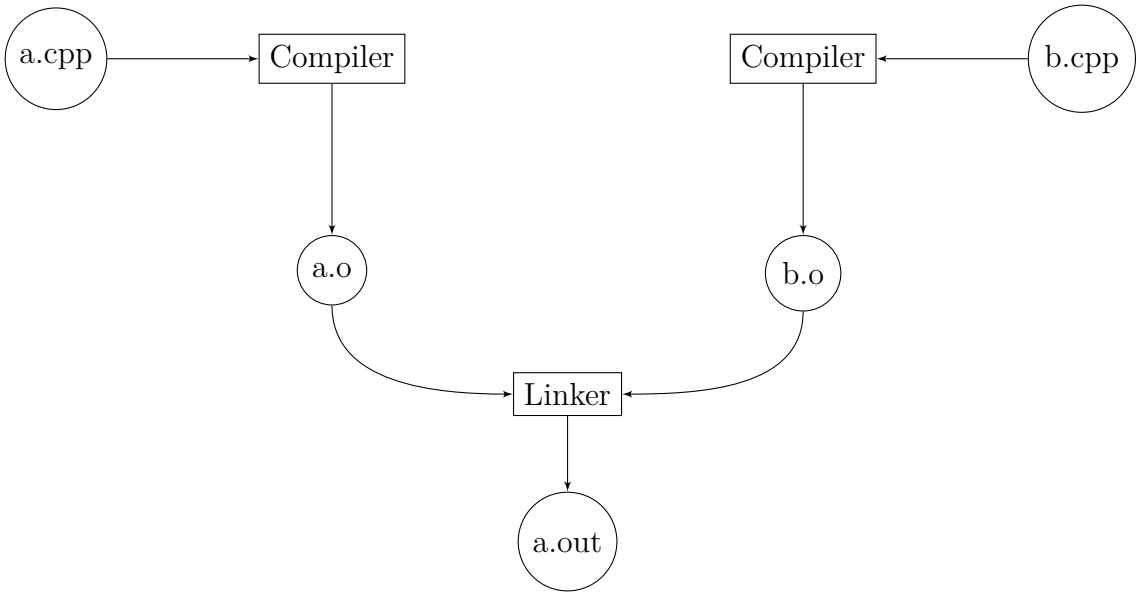


Abbildung 2.1.: Beispiel-C++-Quellcode zur Demonstration des Linkers.

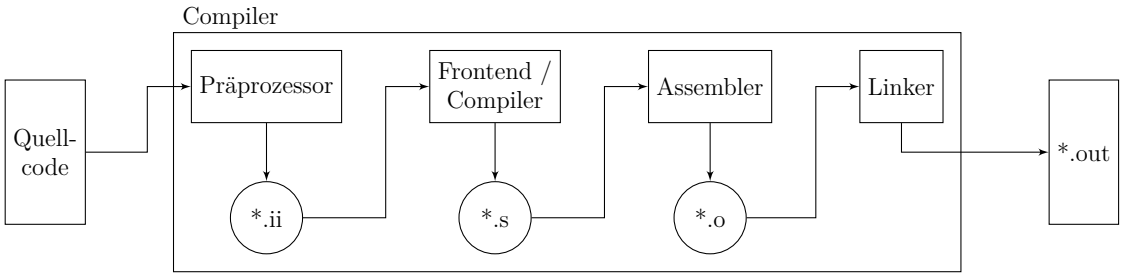


Abbildung 2.2.: Illustration der Interna eines Compilers.

dieses Prinzip für beliebige Symbole (Unterabschnitt 2.7.1) anzuwenden. Einzige Voraussetzung für die Trennung von Deklaration und Definition ist es die Deklaration als extern zu kennzeichnen und eine Definition zu garantieren. Nicht statische Funktionen sind in C standardmäßig als extern markiert.

2.2. Firm

Ein Compiler verwendet eine interne Repräsentation des Programms, genannt Zwischensprache. Eine Zwischensprache ermöglicht es das Analysieren des Quellcodes, das Optimieren der Zwischensprache und das Erzeugen der architekturabhängigen ausführbaren Datei zu entkoppeln. Beim invasiven X10-Compiler wurde entschieden, Firm [4] als Zwischensprache zu verwenden.
Diese hat den Vorteil, dass sie Graph-basiert ist, gegenüber einer verbreiteteren Zwischensprachen-Darstellung als Befehlsliste. Diese entspricht einer Totalordnung, während ein (Firm-) Graph [7] lediglich eine Halbordnung ist. In einer Befehlsliste ist eine exakte Abfolge von Operationen vorgegeben. Betrachtet man als Beispiel eine Addition von 2 auszuwertenden Ausdrücken, so wäre die Reihenfolge der Auswertung beider Operanden in einer Befehlsliste fest.
In Firm hingegen wird die notwendige Ordnung über gerichtete Kanten im Firm-Graph realisiert. Wie in Abbildung 2.3 dargestellt, stehen die einzelnen Operanden des Additionsknoten in keiner gegenseitigen Abhängigkeit, lediglich in einer Abhängigkeit zum Additionsknoten selbst. In dieser Situation würde Firm die Auswertungsreihenfolge der beiden Operanden dieses Knotens nicht festlegen, wodurch mehr Freiraum für Code-Optimierungen entsteht.

2.3. VTables und Laufzeit-Typinformation

2.3.1. VTables

In modernen objektorientierten Sprachen kann eine Klasse (Unterklasse) von einer anderen Klasse (Basisklasse) erben und damit ihre Attribute und Methoden übernehmen. Diese Vererbung wird als eine sogenannte ist-ein (is-a) Beziehung beschrieben, da durch die Vererbung die Unterklasse eine Obermenge der Basisklasse ist. Deshalb ist es möglich, innerhalb einer Klassenhierarchie, eine Referenz vom Typ einer Basisklasse zu erzeugen, die auf ein Objekt vom Typ einer Unterklasse zeigt. Zusätzlich ist es in Unterklassen möglich, beliebige ererbte Methoden neu zu definieren und damit die Basisklassen-Definition zu überschreiben. Die Vererbung erzwingt dabei keine Einschränkung der Basisklasse,

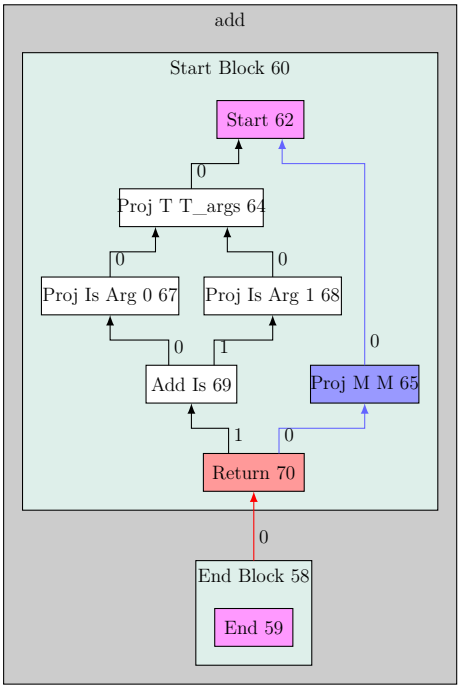


Abbildung 2.3.: Ein Firm-Graph einer Funktion, die zwei Eingabewerte (Arg 0 und Arg 1) addiert und zurückgibt.

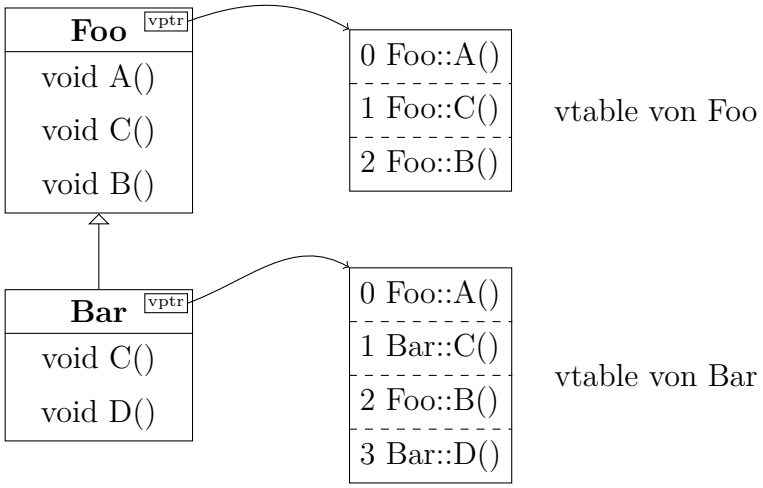


Abbildung 2.4.: Vererbungshierarchie der Klassen `Foo` und `Bar` und ihre zugehörigen *vtables*.

welche wiederum selbst von einer Klasse geerbt haben kann. Viele Sprachen, unter anderem auch X10, unterstützen aufgrund der geringeren Komplexität und einfacheren Verständlichkeit für Klassen keine Mehrfachvererbung, sondern nur die Einfachvererbung, also das Erben von maximal einer Klasse. In dieser kann eine Klassenhierarchie als ein Baum repräsentiert werden, dessen Kanten die einzelnen Vererbungsbeziehungen darstellen. Dabei bedeutet eine Kante (x, y), dass x Oberklasse von y ist. Die soeben erläuterte Vererbungs-Mechanik gilt für den gesamten Vererbungspfad einer Unterklasse bis hin zur Wurzel. Ebenso ererbt eine Klasse sämtliche Attribute und Methoden entlang ihres Vererbungspfades.
Die Schwierigkeit der Vererbung ist es, den Aufruf von Methoden effizient umzusetzen. Durch das Überschreiben muss zwischen dem statischen und dem dynamischen, dem echten, Typ des Objekts, auf das die Referenz zeigt, unterschieden werden. In Abbildung 2.4 ist eine einfache Vererbungshierarchie der Basisklasse `Foo` und der Unterklasse `Bar` dargestellt. Im Rahmen der Vererbung ist ein `Bar`-Objekt auch ein `Foo`-Objekt. In einem Programm ist es also legitim anstelle eines Objekts vom Typ `Foo` ein Objekt vom Typ `Bar` zu verwenden. Deshalb ist es möglich eine Referenz mit dem Typ `Foo` zu erzeugen, die aber auf ein Objekt vom Typ `Bar` zeigt, `Foo var = new Bar();`. Beim Aufruf der Methode `B` einer `Foo`-Referenz, die auf ein Objekt vom Typ `Bar` zeigt, muss korrekterweise die überschriebene Methode `B` aus `Bar` und nicht die Methode `B` aus `Foo` aufgerufen werden, da der dynamische Typ `Bar` ist. Diese Technik wird, wegen der Unterscheidung anhand des dynamischen Typs, als *dynamische Bindung* bezeichnet.
Um so einen Methodenaufruf effizient umzusetzen wird für jede Klasse eine sogenannte `vtable` (virtual method table) angelegt. Zusätzlich speichert jedes Objekt einen Zeiger, den `vptr`, der auf die entsprechende `vtable` des Objekt-Typs zeigt. In dieser `vtable` ist eine Liste von Funktionszeigern für alle Methoden, die die Klasse implementiert, mit jeweils einem festen Index. Der Aufruf an die Methode `B` über eine Referenz `var` mit statischem Typ `Foo` wird nicht mehr als direkter Aufruf einer Funktion umgesetzt. Stattdessen wird die `vtable` des Objekts, auf das die Referenz `var` zeigt, geladen. Hierfür wird der `vptr` des Objekts dereferenziert. In dieser `vtable` wird auf den der Methode `B` zugeordneten Index (2) zugegriffen. Der dort gespeicherte Funktionszeiger wird geladen und kann daraufhin derefereniert und die Funktion ausgeführt werden. Schematisch sieht dieser Aufruf wie folgt aus: `(var.vptr)[2]();`.
Beim Erzeugen einer `vtable` muss die `vtable` der ererbten Klasse übernommen werden. Anschließend kann diese Kopie mit zusätzlichen Methodenimplementierungen erweitert werden. So ist garantiert, dass der Index für ererbte Methoden in der `vtable` sowohl in der Unterklasse als auch der Basisklasse identisch ist. Das Überschreiben von Methoden wird durch das Überschreiben des Funktionszeigers der Oberklasse mit dem Funktionszeiger der Unterklasse in der `vtable` umgesetzt.

```
class InitDispatcher {

    public static val UNINITIALIZED    = 0;
    public static val INITIALIZING     = 1;
    public static val INITIALIZED      = 2;
    public static val EXCEPTION_RAISED = 3;

}
```

Abbildung 2.5.: Ausschnitt aus der Klasse InitDispatcher mit möglichen Zuständen für die Initialisierung von statischen Feldern.

2.3.2. Laufzeittypinformation (RTTI)

Die Runtime-Typ-Info, zu deutsch Typinformation zur Laufzeit, ist eine Sammlung von Informationen über jeden Typen, die zur Laufzeit eines Programmes abgefragt werden kann.
Mit der RTTI ist es möglich, zur Laufzeit herauszufinden, welche Methoden und Attribute ein gegebener Typ hat. Der genaue Inhalt der RTTI-Datenstruktur ist abhängig von der jeweiligen Sprache und dem jeweiligen Compiler.
In Abschnitt 2.5 wird die Serialisierung von X10-Objekten erläutert. Hierfür wird jeder Klasse eine De- und Serialisierungsfunktion zugeordnet, die in einer Tabelle gespeichert werden. Dies geschieht im invasiven X10-Compiler über das Speichern des passenden Tabellen-Indizes in der RTTI-Datenstruktur der jeweiligen Klasse.

2.4. Statische Initialisierung

In X10 werden sämtliche statischen Felder `lazy` initialisiert [8, Kapitel 8.6]. In Abbildung 2.7 sind 2 solche Felder mit je einem trivialen und einem komplexen initialisierenden Ausdruck (Initialisierer) dargestellt. Unter einer lazy (fauler) Initialisierung wird verstanden, dass der Initialisierer eines Feldes, im Beispiel `42` und `g()`, beim ersten Zugriff auf das Feld ausgewertet und das Ergebnis dem Feld zugewiesen wird. Im Falle eines trivialen Initialisierers, einer Initialisierung mit einem konstanten Wert (`42`), ist dies eine Zuweisung der Konstante an das Feld. Bei einem komplexen Initialisierer hingegen (`g()`) muss zuvor der Ausdruck ausgewertet werden, bevor das Ergebnis an das Feld zugewiesen und die Initialisierung abgeschlossen werden kann.
X10 unterstützt Mechanismen zum parallelen Berechnen von Aufgaben (siehe Abschnitt 2.5), weswegen ein paralleler Zugriff auf ein statisches Feld möglich ist. Zudem werden die initialisierenden Ausdrücke in keinster Weise eingeschränkt, weshalb es möglich ist, in einem solchen Ausdruck eine Text-Ausgabe durchzuführen oder beliebige andere stati-

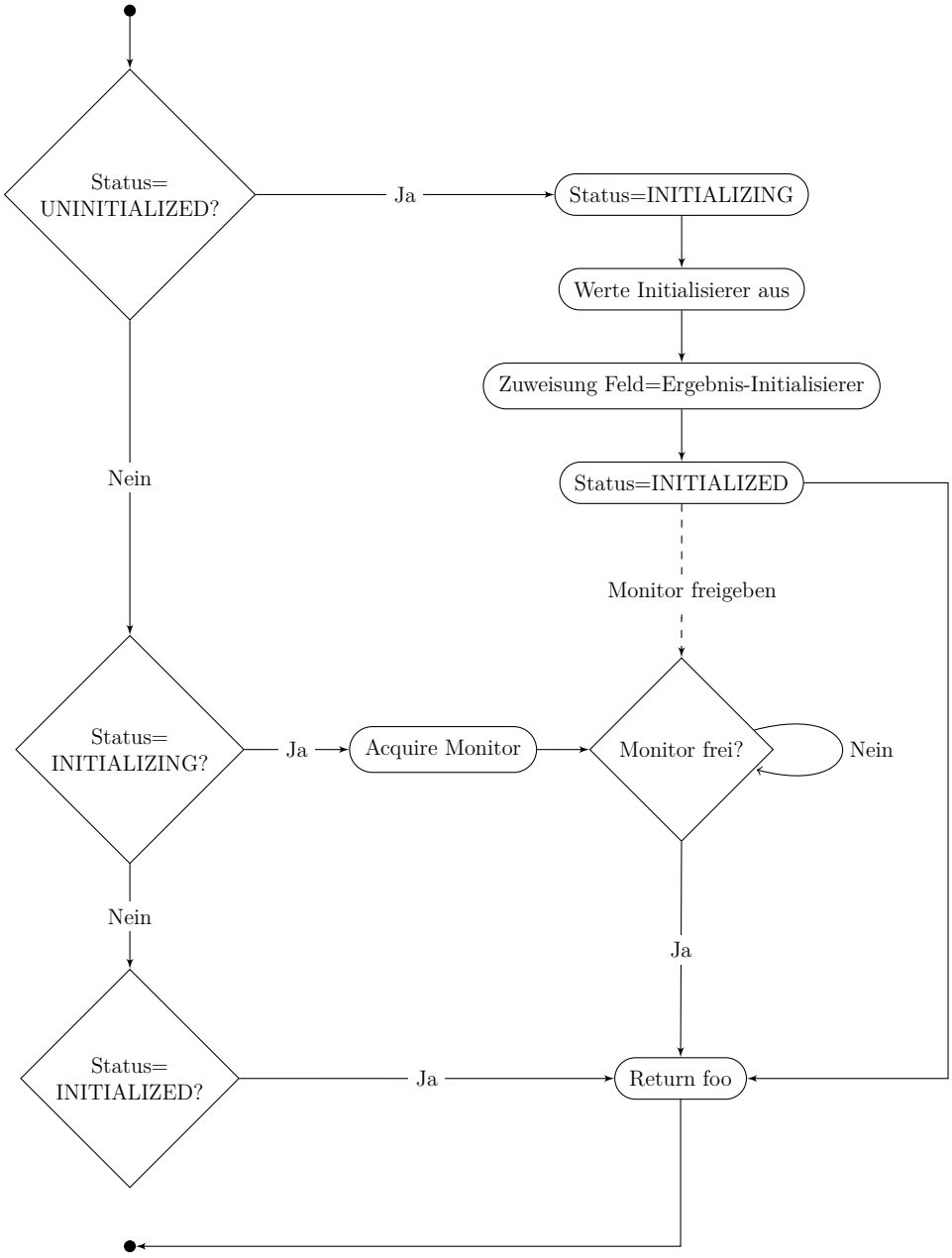


Abbildung 2.6.: Ablaufdiagramm eines Lesezugriffes auf ein statische Feld.

```
class X {
        // trivial
        public static val d = 42;
        // komplex
        public static val f = g();
}
```

Abbildung 2.7.: Statische Felder mit trivialem/komplexen Initialisierer.

sche Funktionen aufzurufen. Das Ziel ist es, Initialisierer genau einmal auszuwerten. Dies hat zur Folge, dass Text-Ausgaben, aber auch andere Seiteneffekte von Initialisierern, nur einmal auftreten. Deshalb ist ein paralleler Zugriff auf ein nicht initialisiertes Feld problematisch. Der X10-Standard spezifiziert zur Lösung dieses Problems einen Mechanismus zum Zugreifen auf statische Felder.
Der X10-Compiler legt dabei für jedes statische Feld eine Zustandsvariable an, in der der Initialisierungszustand des Feldes gespeichert ist. In Abbildung 2.5 befindet sich ein Ausschnitt der Klasse `InitDispatcher`, die der X10-Compiler zur Implementierung des Zugriffs-Algorithmus verwendet. Insbesondere sind dort die möglichen Zustände der Zustandsvariablen definiert.
Zusätzlich generiert der X10-Compiler für jeden lesenden Zugriff auf ein statisches Feld Code, der bei Bedarf den Initialisierer ausführt und dem Feld das Ergebnis zuweist.
Zu Beginn sind alle Feld-Zustände als `UNINITIALIZED` initialisiert. Erfolgt ein Zugriff auf ein Feld mit diesem Zustand, wird dieser zuerst auf den Zustand `INITIALIZING` gesetzt. Dabei muss das Abfragen und Prüfen des Zustands und das Setzen auf `INITIALIZING` zur Vermeidung von Race-Conditions in einer atomaren Operation geschehen. Folgende Zugriffe während der Initialisierung werden genau diesen Zustand, `INITIALIZING`, abfragen und anschließend blockieren und auf das Ende der Initialisierung warten. Tritt dies ein, können auch diese Zugriffe den Feldwert zurückgeben.
Ist die Initialisierung beendet, kann der Wert des Feldes direkt zurück gegeben werden. Dies wird durch den dritten Zustand, `INITIALIZED`, festgelegt.
Der `InitDispatcher` unterstützt noch einen weiteren, vierten, Zustand `EXCEPTION_RAISED`. Dieser wird gesetzt, wenn beim Auswerten des Initialisierers eine Ausnahme auftritt. Sämtliche Folgezugriffe auf so ein Feld führen dann zu einer Ausnahme.

2.5. X10 Places und Serialisierung

X10 hebt sich von anderen Sprachen dadurch ab, dass bereits bei der Spezifikation Schlüsselwörter und Mechanismen definiert wurden, um Aufgaben asynchron und auf sogenannten Places auszuführen [8]. Beides sind Bestandteile des Asynchronous Partitioned Global Address Space Programmiermodell (APGAS), auf welchem X10 basiert.
Ein Place ist ein zusammenhängender und in sich geschlossener Teilbereich des globalen Speichers [8, Kapitel 2.3]. Auf einem isolierten Rechner sind verschiedene Prozesse jeweils ein Place, nicht aber einzelne Threads, da diese sich den Speicher untereinander teilen. Darüber hinaus sind auch die verschiedenen Rechner in einem Netzwerk je ein Place.
Um Berechnungen auf anderen Places als dem aktuellen auszuführen wurde in X10 das Schlüsselwort `at` eingeführt. Mit `at` kann beliebiger Code auf einem anderen Place ausgeführt werden, siehe Abbildung 2.8. Da die Speicherbereiche der einzelnen Places strikt voneinander getrennt sind, muss der X10-Compiler die Daten, auf die innerhalb eines `at`-Abschnitts zugegriffen wird, zwischen den einzelnen Places transferieren.

```
val obj = new Object();
at(B) {
    // Berechnungen ausfuehren
    // mit/auf obj
    obj.someMethod();
}
```

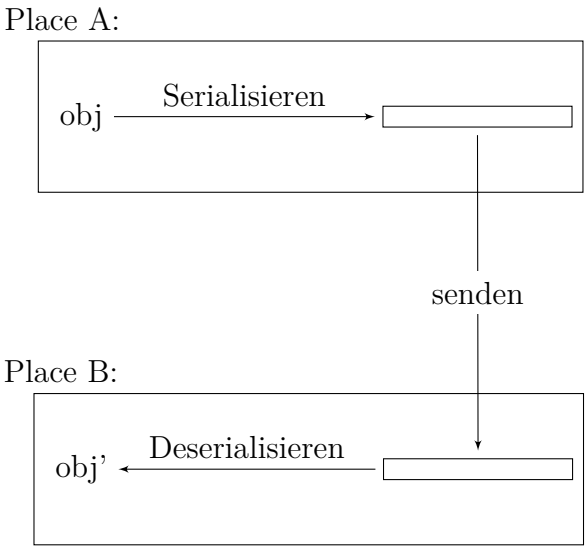


Abbildung 2.8.: Illustration der Ausführung eines at in X10.

In Abbildung 2.8 ist der schematische Ablauf dieses Vorgangs dargestellt. Der X10-Compiler erkennt die Verwendung der Referenz `obj`, die vor dem `at`-Abschnitt erzeugt wurde anhand des Methodenaufrufs `obj.someMethod()` innerhalb des `at`-Abschnitts. Da diese Methode auf einem anderen Place ausgeführt werden soll, muss das Objekt, auf das `obj` zeigt, zuerst in einen Daten-Puffer umgewandelt werden (serialisieren), um es anschließend an `Place B` senden zu können. Dieser empfängt den Daten-Puffer, der von `Place A` gesendet wurde, und konstruiert aus diesem eine Kopie des Objekts (`obj'`) aus `Place A` (deserialisiert es). Anschließend kann der `at`-Abschnitt auf `Place B` ausgeführt werden. Dieser Mechanismus wird vom X10-Compiler umgesetzt. Dabei werden für die Serialisierung und Deserialisierung jeder Klasse entsprechende Funktionen generiert. Um den Objekten ihre entsprechende De- und Serialisierungsfunktion zuzuordnen, generiert der X10-Compiler eine Tabelle, in der diese paarweise pro Klasse aufgelistet sind. Der Tabellen-Index jeder Zeile wird in der RTTI-Datenstruktur der entsprechenden Klasse gespeichert.

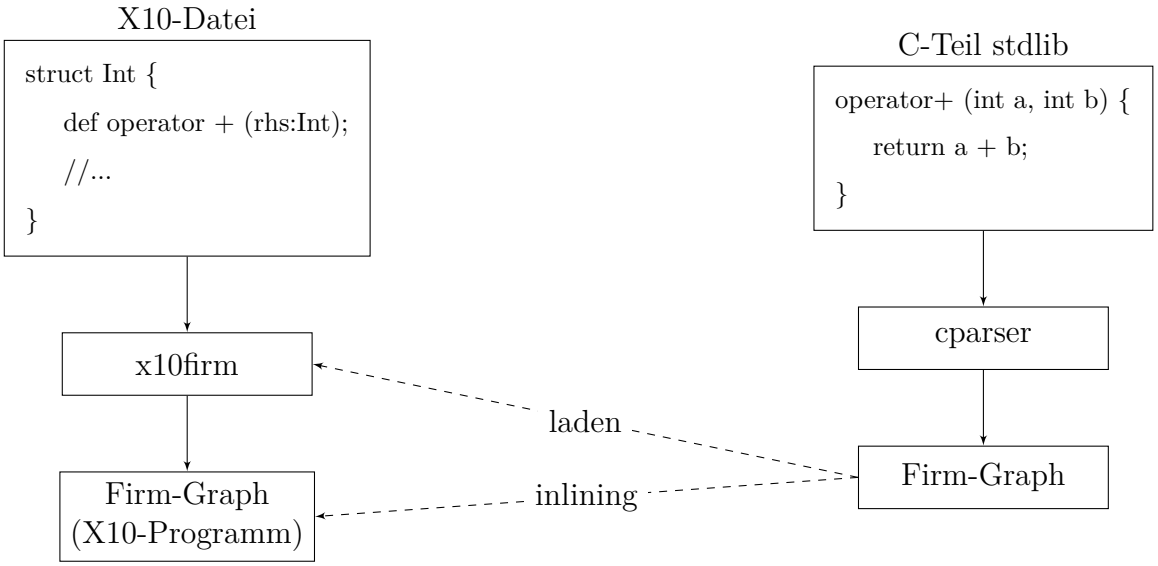
2.6. Inlining bei Basistypen mittels einer Teil-Implementierung der Standardbibliothek in C

In X10 wird, im Gegensatz zu Java, soviel wie möglich in der Sprache selbst und ihrer Standardbibliothek implementiert. Dem Programmierer sollen alle Sprachmittel zur Verfügung gestellt und die Basistypen insbesondere keine Spezialbehandlung erfahren. In Java ist es zum Beispiel möglich einen `String` und einen `int` mit dem `operator +` zu addieren. Dieser konvertiert den übergebenen `int` in einen `String`, indem er dessen

```
public struct Int /* implements ... */ {
    // ...
    public native operator this + (x:Int): Int;
}
```

Abbildung 2.9.: Ausschnitt aus der Implementierung des struct Int.

`toString`-Methode aufruft. Diese Konvertierung wird vom Compiler im Hintergrund umgesetzt indem er die Basistypen `String` und `int` gesondert behandelt. Genau dieser Weg wird in X10 nicht gegangen. Kein Basistyp erfährt eine Spezialbehandlung und alle, auf die Basistypen angewandte Sprachmittel, können vom Programmierer für eigene Typen eingesetzt werden. Dies hat jedoch zur Folge, dass die Operationen auf den Basistypen als Methoden umgesetzt werden. Ein Ausschnitt einer solchen Basistyp-Implementierung ist in Abbildung 2.9 zu sehen. Ohne eine besondere Behandlung würde unter anderem jede Addition als Methodenaufruf umgesetzt werden. Da eine Addition als ein einziger CPU-Befehl umgesetzt werden kann, summiert sich dies zu einem unnötigen Aufwand, insbesondere aufgrund der häufigen Verwendung der Basistypen und zugehörigen Operatoren.
Die betreffenden Operationen sind in den Implementierungen der Basisklassen als sogenannte native Operationen deklariert und verfügen über keinen Methodenkörper. Das Ziel ist es, den Aufruf dieser Operationen durch den konkreten Inhalt der Implementierung zu ersetzen (sogenanntes Inlining). Damit kann das Problem einfach und effizient gelöst werden und ein aufwändiger Methodenaufruf an beispielsweise eine Additionsoperation könnte mit einem einzigen CPU-Befehl umgesetzt werden. Da die nativen Operationen über keine X10-Implementierung verfügen, ist Inlining nicht ohne weiteres möglich.
Im invasiven X10-Compiler wird diese Problematik gelöst, indem dem Compiler eine konkrete Implementierung der nativen Operationen bereit gestellt wird. Dieser Vorgang ist schematisch in Abbildung 2.10 für den Additionsoperator der Klasse Int dargestellt. Da diese Implementierungen nicht in X10 möglich sind, ist der betreffende Teil der Standardbibliothek, in der alle Basistypen implementiert sind, in einem C-Standardbibliothek-Teil implementiert. Am Institut existiert neben dem invasiven X10-Compiler ein C-Compiler, der ebenfalls Firm als Zwischensprache verwendet und somit in der Lage ist, aus der C-Standardbibliothek einen Firm-Graphen zu generieren. Dieser Firm-Graph kann anschließend vom invasiven X10-Compiler geladen werden, um so eine Implementierung für die Operationen der Basistypen bereitzustellen. Mit diesen Informationen kann der invasive X10-Compiler die problematischen Methoden inlinen.



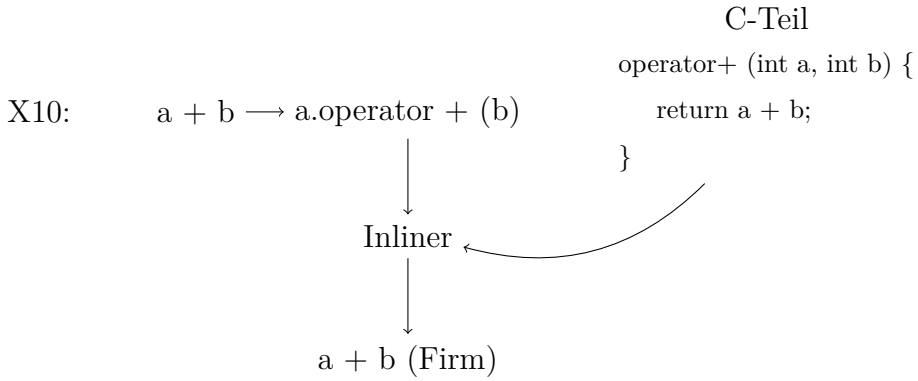


Abbildung 2.10.: Inlining-Optimierung von X10-Basisoperationen durch Laden eines Firm-Graphs, generiert aus einem in C implementierten Teil der Standardbibliothek.

```
// a.cpp
int someVar = 42;
int someFunc(int a, int b) {
    return a + b;
}

// b.cpp
#include <stdio.h>

int __attribute__((weak)) someVar = 1337;
int __attribute__((weak)) someFunc(int a, int b) {
    return a - b;
}

int main() {
    printf("var: %i, ", someVar);
    printf("someFunc: %i\n", someFunc(5, 10);
    return 0;
}
```

Abbildung 2.11.: Verwenddung und Effekt von schwachen und starken Symbolen.

2.7. Linker-Grundlagen

Die Aufgabe des Linkers ist es, verschiedene Objekt- und Bibliotheksdateien zu einer ausführbaren Datei zusammen zu führen. Dabei wird zwischen dynamischem und statischem Linken unterschieden. Dynamisches Linken bezeichnet das Linken zur Laufzeit beim Laden der Bibliotheksdateien. Statisches Linken hingegen wird zur Compile-Zeit durchgeführt und erfolgt direkt nach dem Übersetzen des Quellcodes. Für den X10-Compiler ist nur das statische Linken relevant.

2.7.1. Symbole

In [10] wird ein Symbol als ein externer Name, innerhalb einer Kompilierung, bezeichnet. Zusätzlich muss dieses Symbol von anderen, voneinander unabhängigen Übersetzungseinheiten, die zusammen gelinkt werden, referenziert werden können. Symbole werden für Elemente mit Speicher-Adressen, wie Funktionen oder globale Variablen, vergeben. Für solche Symbole gibt es verschiedene Linker-Annotationen, mit denen das Linken eines Symbols gesteuert werden kann [3].
Der Zweck dieser ist es, zu bestimmen, wie bei mehrfacher Deklaration eines Symbols

```
// Definitionsdatei a.cpp
#include <vector>
std::vector<int> myVectorA;

// Definitionsdatei b.cpp
#include <vector>
std::vector<int> myVectorB;
```

Abbildung 2.12.: Instanzierung der Klasse vector mit dem Typ-Argument int in 2 getrennten Übersetzungen.

verfahren werden soll. Die Standard-Annotation heißt stark (strong). Werden mehrere starke Symbole mit demselben Namen deklariert, so führt dies zu einem Linkerfehler (Mehrfachdeklaration). Wird ein Symbol hinegen als schwach (weak) annotiert, so lässt sich dieses von einem anderen starken Symbol, ohne einen Linkerfehler zu erzeugen, überschreiben. Das schwache Symbol wird in diesem Fall verworfen und stattdessen von einem starken Symbol überschrieben. In Abbildung 2.11 ist die Verwendung von starken und schwachen Symbolen dargestellt. Die Datei `b.cpp` definiert eine Variable `someVar` und eine Funktion `someFunc`, die 2 Argumente addiert. Beide Symbole sind als schwach annotiert. Da die Funktion `someFunc` über eine Definition verfügt, ist der Code aus `b.cpp` fehlerfrei und kann ohne weiteres von gcc kompiliert werden. Die Ausgabe ist in diesem Fall `var: 1337, someFunc: -5`. Weiterhin es möglich in einer anderen Übersetzungseinheit, diese beiden schwachen Symbole zu überschreiben, indem dort ein entsprechendes starkes Symbol definiert wird. Dies wird in der Datei `a.cpp` umgesetzt. Wird diese zusätzlich zu `b.cpp` kompiliert und gelinkt, überschreiben die Symbole aus `a.cpp` die Symbole aus `b.cpp`. Es resultiert die Ausgabe `var: 42, someFunc: 15`.

2.7.2. Comdat-Segment

In C++ und in X10 ist es möglich, generische Klassen zu definieren. Für die Code-Generierung erzeugen diese jedoch ein neues Problem. In Abbildung 2.12 ist ein Beispiel zu sehen, indem 2 getrennte Übersetzungseinheiten eine Instanz der generischen Klasse `vector` mit dem selben Typargument `int` erzeugen. Zusätzlich ist `vector` ein Teil der C++-Standardbibliothek. Diese wurde bereits kompiliert und wird zu jedem Anwendungsprogramm hinzu gelinkt.
Nimmt man an, dass `vector` eine nicht-generische Klasse ohne Typparameter wäre, so wäre der Ablauf der Kompilierung problemfrei. Dem Compiler ist bekannt, dass `vector` bereits kompiliert wurde und sich der erzeugte Code in einer Objektdatei befindet, die mit dem Anwendungsprogramm gelinkt werden wird. Aus diesem Grund muss der Compiler weder beim Kompilieren von `a.cpp`, noch von `b.cpp` Code für `vector` erzeugen.
Für generische Klassen scheitert dieser Ansatz. Dem Compiler ist bekannt, dass `vector`

```
#include <stdio.h>
__attribute__((constructor)) void foo() {
    printf("Hello World!");
}
int main() {
    return 0;
}
```

Abbildung 2.13.: Verwendung der CONSTRUCTOR-Annotation.

eine generische Klasse ist und sich in der Standardbibliothek befindet. Ein C++-Compiler kann für eine generische Klasse aber nur bei konkreten Instanziierungen Code erzeugen. Der Compiler hat keine Information darüber, für welche Typargumente `vector` in der Standardbibliothek bereits instanziiert wurde.
Die Folge ist, dass bei jeder Verwendung einer generischen Klasse der gesamte Code der jeweiligen Instanziierung generiert werden muss. Dies wiederum führt zu dem Problem, dass der Code einer solchen Instanziierung in mehreren Objektdateien definiert werden kann. Ein alternativer Weg wäre der Java-Ansatz, indem der Code nur einmal pro generischer Klasse generiert wird. Dieser wird dann unter allen Instanziierungen geteilt, beschränkt jedoch die Typargumente der generischen Klasse auf ausschließlich Referenz-Typen. Eine Referenz ist unabhängig von dem Typ des zugrunde liegenden Objektes immer gleich groß, weshalb das Objektlayout aller Instanziierungen immer identisch ist und der generierte Code geteilt werden kann.
Um das Problem für mehrere Instanziierungen einer generischen Klasse zu lösen wird das Comdat-Segment eingesetzt. Dieses erlaubt mehrere Deklarationen eines Symbols in verschiedenen Objektdateien[2]. Im Standardfall würde der Linker dies nicht aktzeptieren und eine Fehlermeldung aufgrund einer Mehrfachdeklaration melden. Dies kann verhindert werden indem die betreffenden Symbole im Comdat-Segment platziert werden. Zusätzlich wird spezifiziert wie mit einer Mehrfachdeklaration umgegangen werden soll. Hierfür gibt es verschiedene Optionen. Für das Umsetzen der getrennten Kompilierung reicht es aus, dass es eine Option gibt, die ein beliebiges Symbol behält und den Rest verwirft.

2.7.3. Constructor-Segment

Das Constructor-Segment speichert eine Liste von Funktionszeigern, die bei der Initialisierung des Programmes, also insbesondere vor der Ausführung der Hauptfunktion des Benutzers, ausgeführt werden. Dieses Segment kann genutzt werden, um Initialisierungs-Code auszuführen.
In Abbildung 2.13 ist der Code einer Funktion zu sehen, die mit der entsprechenden Notation (des GCC-Compilers) annotiert wurde, um diese automatisch vor der Haupt-

funktion auszuführen [3]. Da die Hauptfunktion keine Funktionen ausführt, würde das Programm ohne diese Annotation keine Ausgabe erzeugen. Die C-Laufzeitbibliothek des GCC-Compilers führt vor dem Ausführen der Hauptfunktion alle Funktionen aus, für die im Constructor-Segment ein Funktionszeiger gespeichert wurde. Deshalb wird trotzdem `Hello World!` ausgegeben.

3. Entwurf und Implementierung

Die Dokumentation der einzelnen Implementierungen in diesem Kapitel folgen einem festen Schema. Die einzelnen Unterkapitel sind in jeweils 3 Abschnitte geteilt. Eingeleitet wird mit der bisherigen Umsetzung, der in der Abschnittsüberschrift angegebenen Implementierung, im invasiven X10-Compiler. Anschließend wird untersucht, weshalb dieses Vorgehen bei getrennter Übersetzung scheitert und nicht mehr umsetzbar ist. Im letzten Teil wird die Lösung präsentiert und wie sie im X10-Compiler mit getrennter Übersetzung (X10-Compiler$_{\mathrm{SC}}$) umgesetzt wurde. Sofern nicht anders angegeben, bezieht sich X10-Compiler immer auf den invasiven X10-Compiler.

3.1. Grundlegendes Problem mit getrennter Übersetzung

Ohne getrennte Übersetzung kompiliert ein Compiler immer das gesamte Programm inklusive der Standardbibliothek. Der Quellcode dieses Programms und der Standardbibliothek muss dabei für jede Klasse und Methode über eine Definition verfügen. Wird nur ein Teil des Programmes übersetzt, entfallen viele Definitionen, da der Compiler für externe Deklarationen lediglich eine Typ-Prüfung und keine Code-Erzeugung durchführen muss. Deshalb muss der Compiler auf Code aus anderen Übersetzungseinheiten referenzieren. Außerdem muss die Code-Generierung unabhängig von der Whole-World-Annahme sein.

3.2. Serialisierung und Deserialisierung

Der invasive X10-Compiler arbeitet ohne getrennte Übersetzung, wodurch der zu übersetzende Quellcode, wie in Abschnitt 3.1 beschrieben, sämtliche Klassen, die im gesamten Programm verwendet werden, beinhaltet. Deshalb kann der Compiler jeder Klasse eine eindeutige Identifikationsnummer (UID) zuweisen. Zusätzlich generiert der Compiler für jeden Klassentyp eine (De-)Serialisierungsfunktion. Anschließend wird über alle Klassentypen iteriert. Dabei legt der Compiler die zugehörige (De-)Serialisierungsfunktion in

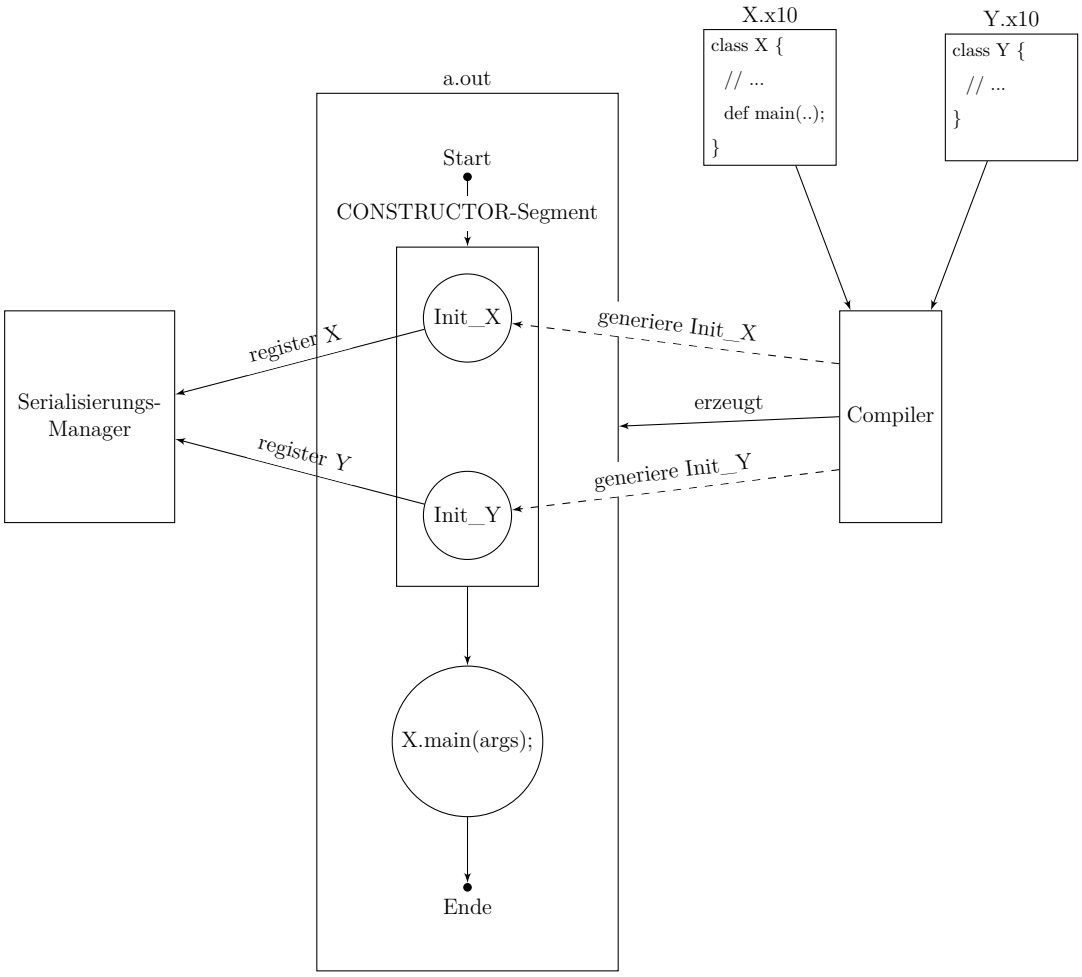


Abbildung 3.1.: Erzeugung und Ablauf der Serialisierungs-Initialisierung.

einer Serialisierungs-Tabelle ab. Als Tabellen-Index wird die UID des Klassentyps verwendet. Da der Compiler die Gesamtanzahl aller Klassentypen kennt, kann die Tabelle lückenfrei mit Serialisierungsfunktionen gefüllt werden.
Soll ein Objekt anschließend zur Laufzeit serialisiert oder deserialisiert werden, so wird die UID des Klassentyps ausgelesen und auf die Tabellen-Zeile mit dem Index der UID zugegriffen. Dort kann anschließend die benötigte Funktion geladen und aufgerufen werden.

Aufgrund der in Abschnitt 3.1 erläuterten Problematik ist es dem invasiven X10-Compiler bei getrennter Kompilierung nicht möglich über alle Klassentypen, die der gesamte Quellcode definiert, zu iterieren. Dies wird beim Betrachten der zeitlichen Abfolge einer Übersetzung eines Anwendungsprogrammes ersichtlich. Bevor das Anwendungsprogramm übersetzt wird, wurde die Standardbibliothek bereits kompiliert. Zu diesem Zeitpunkt existiert das Anwendungsprogramm nicht. Daraus folgt, dass kein Informationsaustausch zwischen den Übersetzungen von Anwendungsprogramm und Standardbibliothek möglich ist. Dadurch ist es nicht mehr möglich, wie bei einer einzelnen Übersetzung zum Übersetzungs-Zeitpunkt eindeutige UIDs an jede Klasse zu vergeben.

Daher wird die Serialisierungs-Tabelle im X10-Compiler$_{\text{SC}}$ erst zur Laufzeit, vor dem Ausführen der Hauptfunktion, aufgebaut. Das ermöglicht es, unabhängig von Informationen über die Typen aus anderen Kompilierungen, immer eine kompakte Tabelle zu erzeugen und eindeutige Klassen-UIDs zu vergeben. Um dies umzusetzen, wurde ein Serialisierungs-Manager implementiert, der das Erzeugen der Serialisierungs-Tabelle, das Registrieren der Klassen und den Zugriff auf die Serialisierungsfunktionen umsetzt. Vor dem Ausführen der Hauptfunktion registrieren sich alle Klassen mit ihren (De-)Serialisierungsfunktionen beim Manager, indem sie eine Registrierungsfunktion aufrufen, die eine Klassen-UID berechnet und zurück gibt. Diese wird in der RTTI-Datenstruktur des Klassentyps gespeichert. Um die Registrierung vor dem Aufruf der Hauptfunktion durchzuführen, wird für jede Klasse eine Initialisierungsfunktion generiert, die die Registrierungsfunktion aufruft. Diese Initialisierungsfunktion wird im Constructor-Segment gespeichert und somit vor dem Aufruf der Hauptfunktion, durch die C-Laufzeitbibliothek, ausgeführt. Soll ein Objekt (de-)serialisiert werden, wird mit der Klassen-UID, die in der RTTI-Datenstruktur der Klasse gespeichert ist, über eine entsprechende Zugriffsfunktion des Managers auf die jeweilige (De-)Serialisierungsfunktion zugegriffen.

3.3. Aufruf der statischen Initialisierer von X10-Klassen

In Abschnitt 2.4 wurde der Zugriff auf ein statisches Feld beschrieben. Dies muss vom Compiler implementiert und für jedes statische Feld der notwendige Code generiert werden. Im invasiven X10-Compiler wurde dies vergleichbar zu Abschnitt 3.2 umgesetzt.

Der Compiler iteriert über alle statischen Felder. Ist das Feld ein triviales, wird es direkt mit dem Wert der initialisierenden Konstanten angelegt. Bei einem komplexen Feld wird eine Zugriffsfunktion generiert. Die Logik dieser Funktion wird in Abschnitt 2.4 beschrieben. Abhängig vom Inhalt der Zustandsvariable des Feldes gibt die Funktion den Feld-Inhalt zurück, initialisiert diesen oder wartet auf das Ende der bereits stattfindenden Initialisierung. Diese Zustandsvariable muss zuvor initialisiert werden. Hierfür generiert der Compiler eine Initialisierungsfunktion, die die Zustandsvariablen für alle statischen Felder initialisiert. Diese Funktion muss vor der Hauptfunktion ausgeführt werden. Der X10-Compiler definiert eine separate Hauptfunktion, `x10_main`, die die Initialisierungsfunktion der statischen Felder aufruft.

Die Problematik bei einer Aufteilung in mehrere getrennte Übersetzungseinheiten ist analog zum Problem bei der Serialisierung in Abschnitt 3.2. Der Compiler kann nicht über alle statischen Felder, die im gesamten Quellcode definiert wurden iterieren, sondern nur über die aus der jeweiligen Übersetzungseinheit. Deshalb kann beim Kompilieren einer getrennten Übersetzungseinheit nicht über alle Felder iteriert und keine einzelne Funktion, die die Zustandsvariablen aller statischen Felder initialisiert, generiert werden.

Um dies zu lösen, generiert der X10-Compiler$_{SC}$ pro Kompilierung eine Initialisierungsfunktion für alle statischen Felder, die in dieser Einheit definiert wurden. Deren Aufrufe können allerdings nicht mehr fest einprogrammiert werden, da die Anzahl an Initialisierungsfunktionen nicht fest ist. Um dennoch alle statischen Felder vor dem Aufruf der Hauptfunktion zu initialisieren, werden die Initialisierungsfunktionen im Constructor-Segment platziert und somit automatisch vor dem eigentlichen Anwendungsprogramm ausgeführt. Um Namenskonflikte und somit Fehler aufgrund der Mehrfachdeklaration einer Funktion zu verhindern, wird dem Funktionsnamen ein eindeutiges Suffix angehängt.

3.4. Laden des aus dem C-Teil der Standardbibliothek erzeugten Firm-Graphen

Ein Teil der Standardbibliothek ist, wie in Abschnitt 2.6 beschrieben, ausgelagert und wurde in C implementiert. Mit dem Firm-Compiler für C (cparser) wurde aus den im C-Teil der Standardbibliothek definierten Methoden entsprechende Firm-Graphen generiert. Diese werden vom X10-Compiler geladen. Beim Aufbauen des Firm-Graphs der Übersetzungseinheit wird bei einer fehlenden Methodendefinition, wie es für die Basistypen der Fall ist, in dieser Liste von Firm-Graphen nach einem Graph für eine Methode mit der selben Signatur gesucht. Ist so ein Graph verhanden wird er verwendet um die fehlende Methodendefinition in der Übersetzungseinheit zu vervollständigen. Anschließend kann eine Inlining-Optimierung vorgenommen werden, um Methodenaufrufe bei

Basistypen zu eliminieren.

Trennt man das Kompilieren in mehrere Übersetzungseinheiten auf, dann muss das Inling separat für alle Einheiten vorgenommen werden. Das bedeutet, dass der Compiler jeden Teil übersetzt und dabei wie in Abschnitt 2.6 beschrieben die Operatoren der Basistypen mittels der Firm-Graphen, der im C-Teil implementierten Methoden inlinet. Die Firm-Graphen der dort implementierten Methoden übertragen sich dabei in den vom Compiler erzeugten Firm-Graphen. Dies führt zu einer Mehrfachdeklaration der Methoden aus dem C-Teil. In Abbildung 3.2 ist diese Problematik dargestellt. Somit kann auf diesem Weg kein Programm mit getrennter Übersetzung erzeugt werden.

Da ohne diese C-Standardbibliothek-Firm-Graphen kein Inlining vorgenommen werden kann, müssen diese bei jeder Kompilierung geladen und die Mehrfachdeklarationen auf einem anderen Weg behoben werden. Beim kompilieren der Standardbibliothek wird ganz normal Code erzeugt. Wird ein Anwendungsprogramm kompiliert, wird für alle Methoden aus dem C-Teil der Standardbibliothek ein Flag gesetzt, sodass für diese kein Code generiert wird. Der Firm-Graph dieser Methoden steht dem Compiler dadurch zum Inlining zur Verfügung ohne mehrfachen Code zu generieren.

3.5. Umgang mit generischen Typen

Eine generische Klasse beziehungsweise eine generische Methode wird mit Typ-Parametern definiert. Wird eine generische Klasse oder Methode verwendet, müssen für diese Typ-Parameter Argumente angegeben werden, die in der jeweiligen Definition substituiert werden können. Diese substituierten Definitionen sind identisch zu regulären Klassen- und Methodendefinitionen.

Der X10-Compiler speichert beim Erzeugen des Codes eine Liste aller benötigten Instanziierungen von generischen Klassen und Methoden. Anschließend werden diese abgearbeitet. Da der Compiler keine getrennte Übersetzung berücksichtigen muss, sind die in Unterabschnitt 2.7.2 genannten Probleme irrelevant. Der Compiler übersetzt den gesamten notwendigen Quellcode für das Programm und kann dabei speichern, ob eine Instanz einer generischen Klasse oder Methode verwendet wird. Dies wird in einer separaten Liste gespeichert, die der Compiler im Anschluss abarbeitet und den Code für alle benötigten Instanziierungen erzeugt.

Die Probeme, die beim getrennten Übersetzen in Bezug auf generische Definitionen auftreten, wurden in Unterabschnitt 2.7.2 erläutert. Der Compiler ist nicht in der Lage, Informationen aus anderen Übersetzungseinheiten zu erhalten. Aus diesem Grund muss innerhalb jeder Übersetzungseinheit für jede generische Instanz Code erzeugt werden, unabhängig davon ob dieser bereits in einer anderen Übersetzungseinheit erzeugt wurde

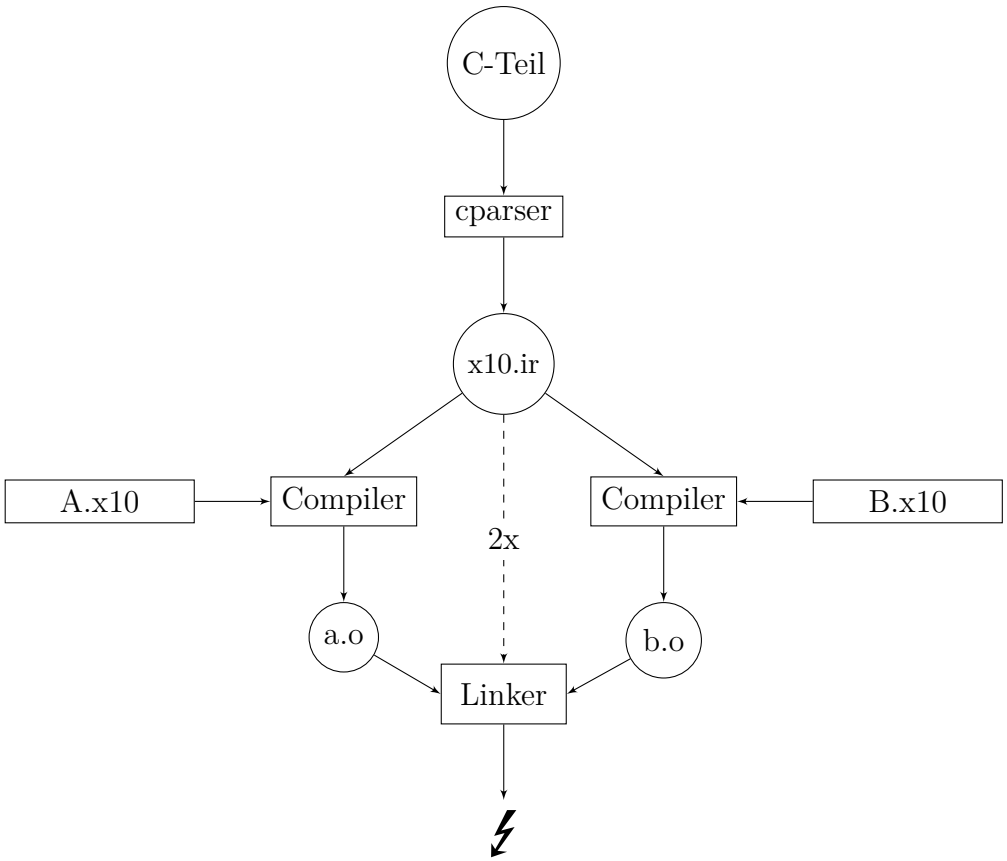


Abbildung 3.2.: Mehrfachdeklaration durch mehrfaches Laden der C-Teil-Implementierung der Standardbibliothek.

oder erzeugt wird. Wird eine generische Klasse oder Methode in mehr als einer Übersetzungseinheit verwendet, werden alle Symbole der generischen Instanz mehrfach deklariert. Dieses Problem tritt sowohl bei generischen Methoden als auch generischen Klassen auf. Zusätzlich tritt das identische Problem für die vtables von generischen Klassen auf. Diese werden ebenfalls in jeder Übersetzungseinheit erzeugt, wenn die generische Klasse instanziiert wird und auch diese führen dann zu einem Fehler aufgrund von Mehrfachdeklaration des jeweiligen Symbols.

Um diese Mehrfachdeklarationen zu verhindern, bleibt keine andere Möglichkeit als die Symbole, die aufgrund einer generischen Klasse oder Methode erzeugt werden, im Comdat-Segment zu platzieren. In C++ wird dies auf die gleiche Art und Weise gelöst. Die Symbole im Comdat-Segment können so annotiert werden, dass der Linker exakt eines davon behält und den Rest verwirft. Der erzeugte Code ist identisch, weswegen das Verwerfen der anderen Symbole korrekt ist. Der X10-Compiler$_{\text{SC}}$ unterscheidet beim Erzeugen des Codes für eine Methode oder eine vtable ob die umgebende Klasse generisch ist und gerade mit gültigen Typ-Argumenten instanziiert wird. Trifft dies zu, wird das erzeugte Symbol im Comdat-Segment platziert, wodurch der Linker später Mehrfachdeklarationen verwerfen kann. Zusätzlich muss beachtet werden, dass eine nicht generische Klasse eine generische Methode definieren darf. Auch die Instanziierungen dieser generischen Methoden von nicht generischen Klassen müssen im Comdat-Segment platziert werden um sämtliche Fehler bezüglich der Mehrfachdeklaration von Symbolen zu verhindern.

Typprüfung generischer Klassen

Klassen müssen generell auf semantische Korrektheit geprüft werden. In X10 ist dies bei generischen Klassen ohne eine konkrete Instanzierung möglich, da wie in Java Typschranken verwendet werden um die notwendige Funktionalität für alle Typparameter zu garantieren. Diese Typschranken müssen bei einer konkreten Instanziierung für die entsprechenden Typargumente geprüft werden. Das vom X10-Compiler verwendete Compiler-Framework *polyglot* [9] sieht vor, dass die Typprüfung genau einmal durchgeführt wird und in dieser alle zu prüfenden Klassen geprüft werden. Dieses Vorgehen ist für eine einzelne Übersetzungseinheit unproblematisch, da alle verwendeten Klassen zu Beginn fest stehen. Bei getrennten Übersetzungseinheiten ist dies nicht mehr der Fall. Zusätzlich ist nicht bekannt, welche generischen Klassen instanziiert werden müssen. Diese Information wird erst beim Erzeugen des Codes für die einzelnen Klassen und Methoden ermittelt. Im Idealfall würden lediglich die relevanten Klassen, die von der Übersetzungseinheit genutzt werden, geprüft. Da *polyglot* nur eine einzige Typprüfung erlaubt und lediglich beim Übersetzen des Quellcodes festgestellt werden kann, welche generischen Klassen verwendet werden, muss deshalb immer die gesamte Standardbibliothek zusätzlich geprüft werden.

3.6. Deaktivieren der Rapid-Typ-Analysis

Beim getrennten Übersetzen muss die Rapid-Typ-Analysis (RTA [5]) deaktiviert werden. Diese setzt die Whole-World-Annahme, also Kenntnis über den ganzen zu übersetzenden Quellcode, voraus. Diese Annahme wird jedoch bei der getrennten Übersetzung gebrochen. Die RTA kann über ein Flag deaktiviert werden.

4. Evaluation

Die Erweiterung des X10-Compilers wurde aufbauend auf der Compiler-Version vom Oktober 2014 implementiert[1]. Auf dem Test-Rechner ist ein Ubuntu 14.04.3 LTS installiert. Der Rechner verfügt über eine Pentium(R) Dual-Core E5300 CPU und insgesamt 4 Gigabyte Arbeitsspeicher. Die Implementierung von getrennter Übersetzung für den invasiven X10-Compiler wurde abgeschlossen. Zur Auswertung, der getrennten Übersetzung, wurde die Compile-Zeit des `HelloWorld`-Programms, mit jeweils dem Optimierungslevel 0 und 3 vom invasiven X10-Compiler als auch dem X10-Compiler$_{SC}$ gemessen. Die Auswertung ist in Abbildung 4.1 abgebildet.

Optimierungsstufe	X10-Compiler	X10-Compiler$_{SC}$	Compile-Zeit-Reduktion
-O0	1m51s	1m06s	41%
-O3	2m37s	1m07s	58%

Abbildung 4.1.: Auswertung und Vergleich des invasiven X10-Compiler und des invasiven X10-Compiler$_{SC}$ anhand des `HelloWorld` Programms.

Zusätzlich wurde in Abbildung 4.2 die Compile-Zeit des größeren X10-Programmes `MultiGrid` mit insgesamt 2545 Zeilen Code gemessen, um insbesondere einen besseren Vergleich zwischen der Compile-Zeit von kleinen und großen Programmen zu erhalten. Die Compile-Zeit-Reduktion von nur 4% liegt an der Verwendung von vielen generischen Klassen aus der Standardbibliothek im `MultiGrid` Programm. Dadurch muss der Compiler für einen Großteil der Standardbibliothek erneut Code erzeugen, weswegen sich die Compile-Zeit kaum verbessert.

Optimierungsstufe	X10-Compiler	X10-Compiler$_{SC}$	Compile-Zeit-Reduktion
-O0	2m00s	1m56s	4%
-O3	3m10s	2m11s	32%

Abbildung 4.2.: Auswertung und Vergleich des invasiven X10-Compiler und des invasiven X10-Compiler$_{SC}$ anhand des `MultiGrid` Programms.

[1] Git-Commit: 8f1bfb025f392c230062cec203c459e1f4118bf4

Insgesamt wirkt die Verbesserung der Compile-Zeit trotz des großen Potentials der getrennten Übersetzung ernüchternd. Dies ist der Problematik der Typprüfung geschuldet, auf die in Abschnitt 3.5 eingegangen wurde.
Da dieses Problem nur bei der Verwendung von generischen Klassen und Methoden relevant ist, wurde eine Vergleichs-Messung des `HelloWorld`-Programms durchgeführt, in der der X10-Compiler$_{SC}$ keine komplette Typ-Prüfung der Standardbibliothek durchführt. Da das `HelloWorld`-Programm keine generischen Klassen und Methoden verwendet, ist es ohne weiteres möglich, dieses zu kompilieren und nur für die notwendigen Klassen eine Typprüfung durchzuführen. Das erzeugte Programm ist korrekt und kann problemlos ausgeführt werden. Diese Messung wird in Abbildung 4.3 mit der Compile-Zeit des X10-Compiler verglichen, um das komplette Potential der getrennten Übersetzung zu zeigen.

X10-Compiler$_{SC}$ mit TP	X10-Compiler$_{SC}$ ohne TP	Compile-Zeit-Reduktion
1m06s	0m05s	93%

Abbildung 4.3.: Vergleich des X10-Compiler$_{SC}$ mit und ohne kompletter Typ-Prüfung der Standardbibliothek.

In Abbildung 4.4 sind die Compile-Zeit-Verbesserungen der beiden Compiler-Varianten, X10-Compiler$_{SC}$ mit und ohne kompletter Typ-Prüfung der Standardbibliothek, zusammengefasst um einen abschließenden Gesamtüberblick über die aktuelle und noch mögliche Geschwindigkeitsverbesserung der Compile-Zeit, am Beispiel des `HelloWorld`-Programms, zu geben.

X10-Compiler	1m51s
X10-Compiler$_{SC}$ mit TP	1m06s
Compile-Zeit-Reduktion	41%
X10-Compiler$_{SC}$ ohne TP	0m05s
Compile-Zeit-Reduktion	96%

Abbildung 4.4.: Zusammenfassung der Compile-Zeit-Verbesserung des `HelloWorld`-Programms durch den X10-Compiler$_{SC}$, mit und ohne komplette Typ-Prüfung der Standardbibliothek, gegenüber dem X10-Compiler.

Daraus folgt, dass der X10-Compiler$_{SC}$ bei der Typprüfung noch deutlich verbessert werden kann. Um diese Ursache zu garantieren wurde eine reine Typprüfung beim Kompilieren des `HelloWorld`-Programms durchgeführt und diese mit dem jVisualVM-Profiler[1] analysiert. Dies ergab, dass der Compiler allein für die Typprüfung des `HelloWorld`-Programms und der Standardbibliothek ca. 30 Sekunden benötigt.

Codequalität

Die Codequalität hat sich im Bereich der Inlining-Optimierung verschlechtert. Bedingt durch das Fehlen der Methodendefinitionen aus der Standardbibliothek, beim getrennten Übersetzen eines Anwendungsprogrammes, ist es dem Compiler nicht mehr möglich Methodenaufrufe aus dem Anwendungsprogramm an Klassen aus der Standardbibliothek zu inlinen. Die Ursache hierfür ist identisch zu der Problematik in Abschnitt 2.6.
Da die Implementierung der getrennten Übersetzung aktuell noch einige Fehler beinhaltet ist es nicht möglich, größere Programme zu übersetzen und die Laufzeit mit der des X10-Compiler zu vergleichen, welcher die Inlining-Optimierung ohne Einschränkung vornehmen kann.
Es sind allerdings keine größeren Performanceeinbußen zu erwarten, da viele Klassen der Standardbibliothek generisch sind und bei der Verwendung dieser Klassen entsprechender Code für die Instanziierung generiert wird. Dieser generierte Code kann ohne Einschränkung inline-optimiert werden.

5. Fazit und Ausblick

Die Implementierung von getrennter Übersetzung war erfolgreich und konnte den invasiven X10-Compiler beschleunigen. Die aktuelle Implementierung ist nicht in der Lage, alle Tests der Test-Suite korrekt zu übersetzen (siehe Anhang A).
Um das volle Potential, wie es in Abbildung 4.3 und Abbildung 4.4 gezeigt wurde, auszuschöpfen, ist es notwendig, die Typ-Prüfung des `polyglot`-Compiler-Frameworks aufzubrechen und auf die Typ-Prüfung des relevanten Teils der Standardbibliothek zu reduzieren.

Literaturverzeichnis

[1] jvisualvm. https://visualvm.java.net/.

[2] Vague linking. https://mentorembedded.github.io/cxx-abi/abi.html#vague.

[3] *GCC - Common function attributes*, 2015.

[4] *libFirm Manual*, 2015.

[5] D. F. Bacon and P. F. Sweeney. Fast static analysis of c++ virtual function calls. In *Proceedings of the 11th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '96, pages 324–341, New York, NY, USA, 1996. ACM.

[6] M. Braun, S. Buchwald, M. Mohr, and A. Zwinkau. An x10 compiler for invasive architectures. Technical Report 9, Karlsruhe Institute of Technology, 2012.

[7] M. Braun, S. Buchwald, and A. Zwinkau. Firm—a graph-based intermediate representation. Technical Report 35, Karlsruhe Institute of Technology, 2011.

[8] Vijay Saraswat, Bard Bloom, Igor Peshansky, Olivier Tardieu, and David Grove. *X10 language specification*, 2015. Version 2.5.

[9] N. Nystrom, M. Clarkson, and A. Myers. Polyglot: An extensible compiler framework for java. In G. Hedin, editor, *Compiler Construction*, volume 2622 of *Lecture Notes in Computer Science*, pages 138–152. Springer Berlin Heidelberg, 2003.

[10] L. Presser and J. R. White. Linkers and loaders. Technical report, 1972.

Erklärung

Hiermit erkläre ich, Christoph Jost, dass ich die vorliegende Bachelorarbeit selbstständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht und die Satzung des KIT zur Sicherung guter wissenschaftlicher Praxis beachtet habe.

Ort, Datum	Unterschrift

A. Sonstiges

A.1. Liste der fehlschlagenden Tests

- AbstractCallBug.x10
- ArrayListTest2.x10
- ArrayTest.x10
- DistArrayTest.x10
- FileTest.x10
- FinishStateStress.x10
- HashMapTest.x10
- HashSetTest.x10
- IMCSerializationTest.x10
- IMCSerializationTest2.x10
- IterableTest.x10
- NQueensDist.x10
- NQueensPar.x10
- PairTest.x10
- RandomTest.x10
- RegionTest.x10

- Serialization.x10
- Serialization2.x10
- StringBuilderTest.x10