Migration von Bauhaus-Werkzeugen auf eine SKilL-Zwischendarstellung

Timm Felden Felix Krause

Universität Stuttgart, Institut für Softwaretechnologie
Universitätsstr. 38, 70569 Stuttgart
{feldentm,krausefx}@informatik.uni-stuttgart.de

Zusammenfassung

In diesem Papier werden die laufenden Migrationsarbeiten in Bauhaus beschrieben. Dabei steht die Migration nicht spezifizierter Kernkomponenten auf entsprechende automatisch generierte SKilL-basierte Komponenten im Mittelpunkt.

1 Einleitung

Bauhaus ist eine Werkzeugkette für statische Programmanalysen, die von der Universität Stuttgart entwickelt wird.[3] Die einzelnen Werkzeuge sind überwiegend in Ada implementiert. Die Analysen fokussieren sich aktuell vor allem auf C-Programme bis etwa eine Million Codezeilen. Da es zunehmend schwierig wird, im Rahmen von Abschlussarbeiten Studenten mit ausreichenden Ada-Kentnissen zu gewinnen, wurde die Zwischendarstellung (IR), über die Werkzeuge miteinander kommunizieren, auf ein sprachunabhängiges Format – SKilL [1] – umgestellt. Für die aus einer maschinenlesbaren Spezifikation generierten Komponenten war dies ohne Weiteres möglich. Gleiches gilt für Werkzeuge, die ausschließlich auf diesen arbeiten. [2] Die Migration der übrigen Komponenten wird im Folgenden beschrieben.

Komponenten, die die bestehende IR-Spezifikation umgehen, teilen sich in zwei Gruppen auf. Die erste Gruppe bilden Werkzeuge, welche Datenstrukturen serialisieren, die sich im Typsystem der ursprünglichen Spezifikationssprache nicht abbilden lassen. Hierbei handelt es sich im Wesentlichen um Arrays und Maps über anderen Typen.

Die zweite Gruppe bilden Werkzeuge, die an der Kapselung der IR vorbei auf die Repräsentation entweder im Speicher oder im Binärstrom zugreifen. Hierzu zählt insbesondere der Linker.

2 Nicht spezifizierte Datenstrukturen

Die Migration nicht spezifizierter serialisierter IR-Datenstrukturen ist konzeptionell denkbar einfach. Alle in Bauhaus serialisierbaren Typen müssen von einem Typ `Storable` erben. Dadurch können serialisierbare Typen leicht durch Textsuche gefunden werden. Wenn man diesen Schritt für so gefundene Typen wiederholt, bis man keine weiteren Typen mehr findet, kann man sicher sein, alle nicht spezifizierten serialisierbaren Typen gefunden zu haben. Ob so gefundenen Zwischendarstellungskomponenten auch genutzt werden, ist zunächst unerheblich.

Hat man alle zu spezifizierenden Typdefinitionen für die Erweiterung der IR gefunden, so kann man daraus eine äquivalente Spezifikation erzeugen und bekommt dann die entsprechenden Datenstrukturen generiert. Die generierten Typdefinitionen finden sich jedoch in generierten Paketen, die sich von den ursprünglichen stark unterscheiden. Das geringste Problem dabei ist, dass der generierte Typ nicht im ursprünglichen Paket generiert wird. In Ada ist es möglich, ein `subtype` zu definieren, was in diesem Kontext einem `typedef` in C entspricht. Dadurch kann man die generierten Typen unter den ursprünglichen Typnamen sichtbar machen.

Ein entsprechendes Vorgehen für den Zugriff auf Felder ist viel schwieriger. Das liegt vor allem an der generierten Zwischendarstellung, die den direkten Feldzugriff unterbindet und über ein Getter-/Setter-Paar kapselt. Dessen Benennung folgt einem offensichtlichen Schema, das sich von handgeschriebenen Zugriffsfunktionen teilweise unterscheidet. Außerdem unterscheidet sich aufgrund von Ada-Feinheiten teilweise die Typisierung der entsprechenden Methoden. Das bedeutet, dass man alle entsprechenden Zugriffe anpassen muss. Dies geschieht mehrheitlich automatisch. So ist es beispielsweise möglich, weitgehend automatisch einen `isPrädikat-` durch einen `getPrädikat`-Aufruf zu ersetzen.

Ebenso können unterschiedliche Integertypen weitgehend automatisch konvertiert werden. Eine Konvertierung der Integertypen, etwa von Integer zu einem garantiert 64 Bit breiten Integer, hat den Vorteil, dass man nicht in den eigentlichen Werkzeugcode eingreifen muss und so nicht Gefahr läuft, dessen Funktionalität wesentlich zu verändern. Davor kann man sich aber nicht immer schützen. So wurde ein Fall gefunden, in dem ein falsch typisierter Iterator über einen Container verwendet wurde, was fälschlicherweise vom Compiler akzeptiert wurde. Das Problem zu beheben ist zwar einfach und nicht aufwändig, es zeigt aber, dass eine vollautomatische Migration nicht gelingen kann. Insgesamt gehen wir

davon aus, dass die Migration nicht spezifizierter serialisierter IR-Datenstrukturen im Rahmen einer Masterarbeit abgeschlossen werden kann. Da die Arbeit noch läuft, ist der Ausgang jedoch ungewiss.

3 Ein neuer Linker

Da der bestehende Linker direkt auf dem Binärstrom arbeitet und dieser inkompatibel geändert wurde, ist der Linker nicht weiter verwendbar. Um das Vorgehen an dieser Stelle nachvollziehen zu können, sind jedoch weitere Effekte relevant. So wurde das bestehende generierte API der IR als Wrapper um das generierte SKilL-API implementiert. Die entstehende Bibliothek ist so groß, dass sie den Start von Werkzeugen um etwa eine halbe Sekunde verzögert. Dabei wird von Compiler und Linker nur ein kleiner Teil der bereitgestellten Funktionen benötigt. Es wäre insbesondere denkbar, auf einen erheblichen Teil der IR-Spezifikation zu verzichten. Dies erfordert allerdings, dass man die Werkzeuge direkt auf eine generierte IR-Implementierung und nicht mehr auf eine zentrale Bibliothek aufbaut, was langfristig ohnehin für alle Werkzeuge beabsichtigt ist, da sie so eine maßgeschneiderte IR-Spezifikation verwenden können.

Daneben gibt es noch zwei funktionale Aspekte. Derzeit durchläuft eine C-Quelldatei im Regelfall vier Werkzeuge, bis sie fertig bearbeitet in der Zwischendarstellung vorliegt. Bei näherer Betrachtung der Transformationsschritte stellt man fest, dass die Aufgaben der beiden neben Compiler und Linker verwendeten Werkzeuge überwiegend in den Linker integriert werden sollten. Die nicht in den Linker integrierten Aufgaben sollten in ein separates Werkzeug ausgelagert werden, das bei Bedarf von nachgelagerten Analysen ausgeführt werden kann. Der zweite Punkt besteht in der Funktionsweise des Linkers selbst. Dieser linkt, wie bei Linkern üblich, auf Namen basierend, Entitäten zusammen. Das hat bei einer auf Programmanalysen ausgerichteten IR jedoch den Nachteil, dass es Duplikate von eigentlich strukturäquivalenten Entitäten geben kann. Ein Beispiel hierfür wäre etwa ein Typobjekt, das `int**` repräsentiert. Unifiziert der Linker alle entsprechenden Entitäten korrekt, so kann beispielsweise von nachfolgenden Analysen Typgleichheit durch Zeigergleichheit entschieden werden, was viel effizienter ist.

Es wird also ein neuer Linker entwickelt, der direkt auf dem SKilL-API aufbaut. Hierdurch wird das Werkzeug nicht nur kleiner und schneller – es verbessert sich auch die Wartbarkeit. Zum einen ist die IR für den Linker auf die von ihm tatsächlich verwendeten Teile beschränkt, was bedeutet, dass man sich nur mit diesen beschäftigen muss. Zum anderen führt die Änderungstoleranz der SKilL-IR dazu, dass sich nachgelagerte Werkzeuge automatisch an die vom Linker erzeugte IR anpassen. Das bedeutet, dass nachgelagerte Werkzeuge die IR beliebig erweitern können, ohne den Linker damit zu beeinflussen. Ebenso können gelinkte Programme mindestens so lange verwendet werden, bis sich die IR-Spezifikation des Linkers ändert.

Außerdem kann als Implementierungssprache zunächst Java verwendet werden, was es erlaubt, die eigentliche Implementierung von einem Studenten durchführen zu lassen. Ebenso sind zunächst für Bestandssoftware nicht untypische Probleme wie das Fehlen von Dokumentation oder offensichtlich nicht mehr korrekte Dokumentation zu beseitigen. Erst wenn übersichtlich dokumentiert ist, wie der Linker genau funktioniert, sollte über Performanz nachgedacht werden. Da es auch eine hochperformante SKilL/C++-Implementierung gibt, ist davon auszugehen, dass man, sobald man die korrekte Algorithmik kennt, ohne Weiteres den Java-Code nach C++ migrieren kann und so mindestens eine zum alten Linker konkurrenzfähige Implementierung erreicht. Insbesondere ist, basierend auf der Erfahrung mit der Implementierung anderer Werkzeuge, davon auszugehen, dass ein mit SKilL/C++ implementierter Linker, der keine Bibliothek dynamisch laden muss, in etwa so groß sein wird wie der alte Linker.

Ähnliches gilt für den Compiler. Hierbei handelt es sich ohnehin um ein C++-Werkzeug, das auf die bestehende IR-Implementierung über einen Wrapper zugreift. Das Ersetzen der Zugriffsfunktionen durch SKilL/C++-Äquivalente erlaubt es, dabei nicht nur die Performanz zu steigern. Da damit Compiler und Linker von der Implementierung restlichen Werkzeugkette entkoppelt wurden, können diese auch aus dem Build-Prozess entfernt werden. Hierdurch wird ein bestehendes Problem bei der Arbeit mit Studenten gelöst. Dieses besteht darin, dass im bestehenden zentral organisierten Bauhaus eine Anpassung der IR-Spezifikation ein Rebuild aller Werkzeuge erfordert. Da Compiler/Linker für erwartbare Änderungen binärkompatibel wären, könnte man diese den Studenten einfach zur Verfügung stellen.

4 Fazit

Insgesamt zeigt sich, dass eine Migration auf eine SKilL-basierte IR für Bauhaus auch dann möglich ist, wenn die Migration überwiegend durch Masterarbeiten erfolgen muss. Ferner zeigt sich, dass die für SKilL-basierte Zwischendarstellungen vorhergesehenen Vorteile in Bauhaus zunehmend erkennbar und entsprechend genutzt werden.

Literatur

[1] SKilL on Github. `https://github.com/skill-lang/skill`.

[2] Timm Felden and Felix Krause. Evolution of a Program Analysis Toolchain. In: *Softwaretechnik-Trends*, volume 37(2), 2017.

[3] Aoun Raza, Gunther Vogel and Erhard Plödereder. Bauhaus – A Tool Suite for Program Analysis and Reverse Engineering. In: *Reliable Software Technologies – Ada-Europe 2006*, *LNCS*, volume 4006, (pages 71–82), 2006.