Graduiertenkolleg 1103

Embedded Microsystems

Albert-Ludwigs-Universität Freiburg

Program Analysis Framework JAVA(X)

Statusbericht

Markus Degen

Betreuer: Prof. Dr. Peter Thiemann
Lehrstuhl: Programmiersprachen

Freiburg, im September 2008

Institut für Informatik

Institut für Mikrosystemtechnik

1 Aktueller Stand der Promotion

Die erarbeiteten Ergebnisse werden momentan ausformuliert, verfeinert und zu einer Dissertation zusammengeschrieben.

2 Zusammenfassung der Dissertation

Während meines Förderzeitraums im Graduiertenkolleg „Eingebettete Mikrosysteme“ entwickelten wir das Programmanalysesystem JAVA(X). In JAVA(X) wird das Typsystem von JAVA mit einem Annotationssystem erweitert. Die Annotationen werden hierbei aus einer benutzerdefinierten Algebra X gezogen. Durch die Freiheiten der Annotationen sind vielseitige Analysen durchführbar. Insbesondere Sicherheitseigenschaften, wie sie in eingebetteten System häufig erwünscht und benötigt werden, können so bereits statisch bei der Kompilierung garantiert werden.

Das Typsystem für JAVA(X) ist vollständig formalisiert [1]. Eine Implementierung inklusive einer Typinferenz und dem dazugehörigen Constraintsystem liegen ebenfalls vor [2]. Zusätzlich konnten wesentliche Schritte zu einer Portierung zu C mit Pointer- und Arraybehandlung gemacht werden.

2.1 Beispielanwendung

Ein mögliches Szenario, in dem JAVA(X) zum Einsatz kommen kann, ist ein Probenehmer (siehe Abbildung 1), der in einer Kläranlage Wasserproben zu einer genaueren Untersuchung nehmen kann. Das System besteht hierbei aus verschiedenen Ventilen (v) und Pumpen (p), dem Array mit Proben und entsprechendem Selektor (m), einer variablen Anzahl von Sensoren oder Timern (s) und dem Controller.

Um die Proben brauchbar und getrennt zu halten, muss jeweils die Leitung leer sein und für eine neue Probeflasche aus den Probenarray gesorgt werden. Ansonsten werden die Proben vermischt oder mit sich bereits in der Leitung befindlichem Wasser verunreinigt. Um die Leitung leer zu pumpen, muss entsprechend die Pumpe rückwärts laufen und die entsprechenden Ventile geöffnet sein. Diese systembedingten Einschränkungen können in JAVA nicht direkt modelliert werden und können lediglich durch dynamische Test zur Laufzeit überprüft werden. Mit einer Modellierung in JAVA(X) können diese und weitere Eigenschaften statisch zur Compilezeit, unabhängig von der jeweiligen Implementierung, sicher gestellt werden.

Der Designer des Systems kann die entsprechenden Bedingungen in Form von Annotationen und Methodensignaturen kodieren. Der Programmierer des Controllers programmiert dann wie gewohnt und erhält gegebenenfalls bei der Übersetzung des Programms spezifische Fehlermeldungen bei falschen Abläufen.

2.2 Grundlagen

Programmiersprachen mit garantierter Typkorrektheit sind in der Lage viele Programmierfehler bereits statisch bei der Übersetzung zu entdecken. So wird z.B. garantiert, dass keine Methode mit Argumenten falschen Typs aufgerufen wird. Dennoch kann die Typkorrektheit viele weitere Laufzeitfehler nicht verhindern. Ein einfaches Typsystem hat keine Möglichkeit, Bedingungen oder Eigenschaften über den Typ hinaus zu testen. Dadurch können z.B. Methoden zwar mit dem korrekten Typ aufgerufen werden, die Argumente verletzen jedoch andere Voraussetzungen. Beispielsweise kann das Abfragen des Kopfes einer leeren Liste und Dividieren durch Null mit einem

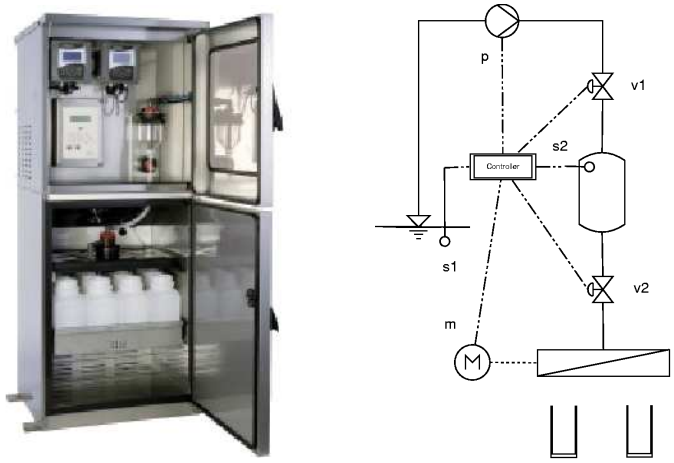

Abbildung 1: Probenehmer.

Standardtypsystem nicht verhindert werden. Es existieren verfeinerte Typsysteme, die solche zusätzlichen Eigenschaften abdecken [5]. Diese erfassen jedoch meist nur vorab im System festgelegte Eigenschaften, wie z.B. ob eine Liste leer ist. Dadurch werden diese Systeme sehr unflexibel und domänenspezifische Probleme können so nicht behandelt werden. Diese Inflexibilität führt dazu, dass diese verfeinerten Typsysteme bisher in der Praxis kaum eingesetzt werden.

Ein zusätzliches Problem tritt auf, wenn Objekte während eines Programmablaufes verschiedene Zustände durchlaufen sollen [3]. Einfache Typsysteme stellen keine Informationen über den momentanen Zustand des Objekts zur Verfügung. Somit kann statisch nicht verhindert werden, dass sich Objekte in einem unzulässigen Zustand befinden oder an Methoden weitergereicht werden, die einen anderen Zustand voraussetzen. Beispielsweise darf der Probenehmer nur dann eine neue Probe durchpumpen, wenn auch ein leeres Probeglas unter den Auffangbehälter gefahren wurde. Nach Abfüllen der Probe ist die entsprechende Flasche voll und somit in einem anderen Zustand. Der Zustand des Systems, ob Leitungen oder Probeflaschen, ändert sich somit linear während des Ablaufs des Programms. Eine Erweiterung des Typsystems muss somit einer Variable an verschiedenen Programmpunkten verschiedene Typen zuweisen. Hierbei stellen Aliase eine zusätzliche Herausforderung dar. Es muss verhindert werden, dass einzelne Referenzen veraltete Informationen über das entsprechende Objekt haben.

Solche Eigenschaften sind insbesondere für eingebettete Systeme wichtig, da gerade hier oft Zustände des Systems über mögliche weitere Aktionen entscheiden und falsche Abfolgen zwar durchführbar sind, aber zu Datenverlusten oder fehlerhaftem Verhalten des Systems führen können. So kann z.B. eine fehlerhafte Ventilstellungen die Sicherheit einer Anlage gefährden, zu einem Abbruch im Programmablauf führen solche Fehler im Allgemeinen aber nicht. Auch ein Abbruch mittels Exception durch dynamische Tests zur Laufzeit ist hier nicht erwünscht, da damit unerwünschte Neustarts nötig werden oder vorab komplizierte, unübersichtliche und potentiell fehlerhafte Behandlungen der Exceptions eingebaut werden müssen.

2.3 Funktionsweise von JAVA(X)

Die Sprache JAVA(X) löst diese Herausforderungen mittels eines annotierten linearen Typsystems [4]. Hierbei werden an die jeweiligen Typen zusätzliche Annotationen angehängt, die sich im Laufe des Programms linear ändern können. Dabei werden zwei unterschiedliche Annotationen verwendet: Zum Einen gibt es in JAVA(X) Wertannotationen, die Informationen über den Zustand

des Objekts halten. Die zweite Annotationsart, die sogenannte Aktivitätsannotation, gibt an, ob das jeweilige Objekt verändert werden darf.

Da die Verfeinerung mittels Wertannotationen, wie oben beschrieben, nicht vorab festgelegt werden soll, ist unser System über Wertannotationen parametrisiert. Der Benutzer von JAVA(X) kann über das Framework eine Annotationsmenge mit einer Algebra X_C angeben, die dann für die entsprechende Klasse C verwendet werden kann. Dadurch ist das System unabhängig von dem jeweiligen Problem und kann schnell an die Anforderungen verschiedener Domänen angepasst werden. Durch die Wertannotationen werden die jeweiligen Typen verfeinert. Das System sichert dem Benutzer für jede korrekt angegebene Wertannotation die Typkorrektheit zu. Durch die Wertannotationen können so mit der Typkorrektheit zusätzliche Eigenschaften über das Programm garantiert werden. Mit den Wertannotationen kann somit beispielsweise sicher gestellt werden, dass die Pumpe nur bei entsprechend geöffneten Ventilen laufen darf.

JAVA(X) löst die Problematik der Aliase mittels weiterer Annotationen, den Aktivitätsannotationen. Es gibt aktive ($\triangle$), inaktive ($\triangledown$) und halbaktive ($\Diamond$) Aktivitätsannotationen. Dabei stellt JAVA(X) sicher, dass es jeweils nur eine aktive Referenz auf ein Feld eines Objektes vorhanden ist. Somit erlauben diese Annotationen lediglich einer Referenz das Objekt zu ändern. Auf diese Weise kann eine verdeckte Zustandsänderung durch andere Aliase verhindert werden. Ein Alias, das eine inaktive Annotation hat, kann auf das Objekt lediglich lesend zugreifen, wohingegen eine aktive Annotation vollen Schreib- und Lesezugriff gewährt. Eine von uns eingeführte dreistellige Relation ($\cdot \succeq \cdot \mid \cdot$) zur Teilung der Aktivitätsannotationen verhindert, dass mehrere Aliase gleichzeitig eine aktive Annotation auf ein Feld haben. Dabei werden inaktive (halbaktive) Annotationen immer zu zwei neuen inaktiven (halbaktiven) Annotationen geteilt: $\triangledown \succeq \triangledown \mid \triangledown$. Bei aktiven Referenzen gibt es entsprechend zwei Möglichkeiten, die aktive Referenz weiter zu geben: $\triangle \succeq \triangle \mid \triangledown$ oder $\triangle \succeq \triangledown \mid \triangle$. Auf diese Weise wird die Aktivität eines Objekts linear propagiert. Die halbaktive Annotation wurde zur besseren Benutzbarkeit des Systems eingeführt. Halbaktive Felder verhalten sich wie standard Objekte in Java. Sie können gelesen und geschrieben werden; es gibt dafür aber keine weiteren Informationen aus den Annotationen über das Objekt, lediglich eine obere Schranke an Wertannotationen kann angegeben werden.

Die Felder von Objekten erhalten jeweils eigene Annotationen. Auf diese Weise ist es insbesondere möglich, dass Felder an andere Referenzen bzw. Methoden mit einer aktiven Annotation übergeben werden können, ohne dabei die Rechte für das gesamte Objekt abgeben zu müssen. Das System garantiert dennoch zu jeder Zeit, dass es auf ein Feld immer nur einen aktiven Zugriffspfad gibt. Falls die entsprechende Referenz nicht halbaktiv ist, gibt es sogar immer genau einen solchen aktiven Zugriffspfad. Dies ist insbesondere für das weiter unten beschriebene Verwerfen von Referenzen wichtig. Sobald eine Referenz eine halbaktive Annotation hat, ist entsprechend auch sichergestellt, dass keine aktive Referenz auf dieses Objekt mehr existiert.

Zusätzlich zu den oben beschriebenen Eigenschaften kann mit JAVA(X) sicher gestellt werden, dass Objekte sauber initialisiert oder aufgeräumt werden, bevor diese verwendet oder verworfen werden können. Somit kann beispielsweise sichergestellt werden, dass ein neues Probenarray zuerst eingebunden wird, bevor weitere Befehle an den entsprechenden Motor zur Ausrichtung gegeben werden, oder Probeflaschen verschlossen werden, bevor diese im aktuellen Programmcode nicht mehr zugreifbar sind. Um solche Prädikate zu überprüfen, wird jedes Objekt, das seine aktive Referenz verlieren soll, auf eine zusätzliche Relation (ρ) überprüft, die angibt, wann das Verwerfen zulässig ist. Die Zustände, in denen ein Objekt verworfen werden darf, werden mit einer Teilmenge der Wertannotationen vom Designer angegeben. Objekte können, solange es eine aktive Referenz gibt, beliebig zwischen den Zuständen wechseln.

Ein besonders Augenmerk wurde bei dem Design von JAVA(X) auch auf die Benutzerfreund-

lichkeit des Systems gelegt. Eine weitere Möglichkeit die oben angesprochenen Herausforderungen zu lösen sind zusätzliche explizite Prädikate. Diese Prädikate müssen jedoch in einer vorgegebenen Logik formuliert werden, was dem Programmierer zusätzliches Wissen über diese Logik und deren Verwendung abverlangt. Die Annotationen sind dagegen verhältnismäßig einfach zu benutzen. Lediglich der Designer, der die Annotation vorgibt, muss die Eigenschaften seiner Domäne genau kennen und formalisieren. Der Designer abstrahiert die benötigten Prädikate und Eigenschaften der Domäne zu einer Halbordnung von Wertannotationen, die Typkorrektheit wird dann direkt von JAVA(X) garantiert.

2.4 Algebra X

Um JAVA(X) einzusetzten muss der Designer der Annotationen für jede Klasse C, für die zusätzliche Prädikate bereits zur Compilezeit sichergestellt werden sollen, eine entsprechende Algebra X angeben. Dabei müssen folgende Mengen mit entsprechenden Einschränkungen angegeben werden:

- Eine Menge mit Halbordnung und kleinstem Element $(X_C, \leq)$, die die möglichen Wertannotationen enthält.
- Ein nicht leeres, nach unten abgeschlossenes Prädikat $\rho_C \subseteq X_C$ derjenigen Annotationen, die verworfen werden können.
- Zusätzlich nach oben abgeschlossene Prädikate $R_C^{\texttt{null}}, R_C^{\texttt{new}} \subseteq X_C$ für `null` und `new` für die Initialisierungswerte der Annotationen.

2.5 Typinferenz für JAVA(X)

Für JAVA(X) wurde ein Parser nebst Typchecker und Typinferenz auf Methodenebene implementiert. Für die Typinferenz wurde ein constraint-generierendes Typsystem erstellt und mit entsprechendem Constraintlöser implementiert. Die Methodensignaturen sollen für die Inferenz dabei vom Designer der Annotationen vorgegeben werden. Eine Inferenz der Methodensignaturen könnte die gewünschten Sicherheitsbedingungen nicht erfassen, da dieses bewusst Einschränkungen an die gültigen Programme hinzufügen.

Trotz der Splitting-Relation, die verschiedene Ausgänge für die Aktivitätsannotationen ermöglicht, konnte mit Hilfe einer zusätzlichen Aktivitätsannotation für den Constriantlöser eine Algorithmus gefunden werden, welcher die generierten Constraints ohne Backtracking in linearer Zeit löst.

2.6 Zusammenfassung und Erweiterungen

JAVA(X) basiert auf JAVA 1.4 und ist vollständig formalisiert inklusive Typkorrektheitsbeweis. Um das System übersichtlich und in einer akzeptablen Größe zu halten, haben wir bei der Formalisierung vorerst auf die Konzepte der Vererbung und Polymorphismus verzichtet, um somit das Kernsystem übersichtlich zu halten. Polymorphismus kann jedoch einfach mit Hilfe von Annotationsvariablen und Constraints realisiert werden. Eine entsprechende Formalisierung ist bereits vorhanden. Ebenso können Interfaces, Inheritance und Casts leicht in das volle System integriert werden.

Da C nach wie vor die Sprache der Wahl für eingebettete Systeme ist, wogegen JAVA vor allem für ‘große’ eingebettete Systeme wie Handys oder ähnlichem verwendet wird, wurden erste Schritte unternommen, die Ideen von JAVA(X) auf die Programmiersprache C zu übertragen. Einige der Konzepte von JAVA(X) konnten dabei direkt übernommen werden, wohingegen bei anderen

Sprachkonstrukten weitere Arbeit geleistet werden muss. Insbesondere der allgemein übliche Umgang mit Pointern in C, im speziellen die Pointerarithmetik, stellt hier eine Herausforderung dar. Das System soll dabei möglichst alle Konzepte von C erfassen. Bisher konnten Arrays statischer Größe in das formale System aufgenommen werden. Erweiterungen an der Syntax ermöglichen hier zusätzliche Erweiterungen, insbesondere auf dynamische Arrays. Dennoch sind bisher auf Grund der Möglichkeiten in C Einschränkungen an der allgemeinen Pointerarithmetik notwendig. Um das System für die Verwendung attraktiv zu machen, wird auch hier auf die Einfachheit und leichte Umsetzung für den Programmierer geachtet.

Literatur

[1] M. Degen, P. Thiemann, and S. Wehr. “Tracking linear and affine resources with Java(X)”. In *Proceedings of the 21st European Conference on Object-Oriented Programming*. ser. Lecture Notes in Computer Science. Berlin, Germany: Springer, 2007, pp. 550–574.

[2] M. Degen, P. Thiemann, and S. Wehr. “Typinferenz für Java(X)”. In *Tagungsband zum 25. Workshop der GI-Fachgruppe Programmiersprachen und Rechenkonzepte*. Bad Honnef, Germany, 2008, in press.

[3] M. Schrefl and M. Stumptner. “Behavior-consistent specialization of object life cycles”. *ACM Transactions on Software Engineering and Methodology*, vol. 11, no. 1, pp. 92–148, 2002.

[4] D. Walker. “Substructural type systems”. In *Advanced Topics in Types and Programming Languages*. B. C. Pierce, Ed. MIT Press, 2005, ch. 1.

[5] H. Xi and F. Pfenning. “Dependent types in practical programming”. In *Proceedings of the 26th ACM SIGPLAN - SIGACT Symposium on Principles of Programming Languages (POPL)*. A. Aiken, Ed. San Antonio, TX, USA: ACM Press, 1999, pp. 214–227.