Refactoringwerkzeuge für die Sprachen der C-Familie

Ausarbeitung im Rahmen des Softwaretechnik-Seminars

an der TU-Berlin im Sommersemester 2005

Martin Häcker <mhaecker@cs.tu-berlin.de>

Motivation

Moderne Entwicklungsumgebungen bieten dem Programmierer Werkzeuge, die ihn beim verändern der Struktur des Codes (engl. to refactor) unterstützen. Diese Werkzeuge sind enorm wertvoll, da sie die drei Hauptschwierigkeiten des Refactorings – Fehleranfälligkeit, Stupidität und hoher Zeitaufwand – mildern. Allerdings existieren weit entwickelte Werkzeuge, die korrekte Transformationen garantieren, bisher nur für Smalltalk, Java und in Anfängen für Python, Perl, C# und VisualBasic.

Nach wie vor ist aber ein sehr großer Teil des Sourcecodes, der heute verwendet wird, in Sprachen aus der C-Familie geschrieben. Eine normale Unix-Distribution[1] besteht z.B. zu etwa 70% aus Sprachen der C-Familie. Refactoring-Werkzeuge, die die Sprachen aus der C-Familie unterstützen, werden also gebraucht, existieren aber bisher noch nicht.[2]

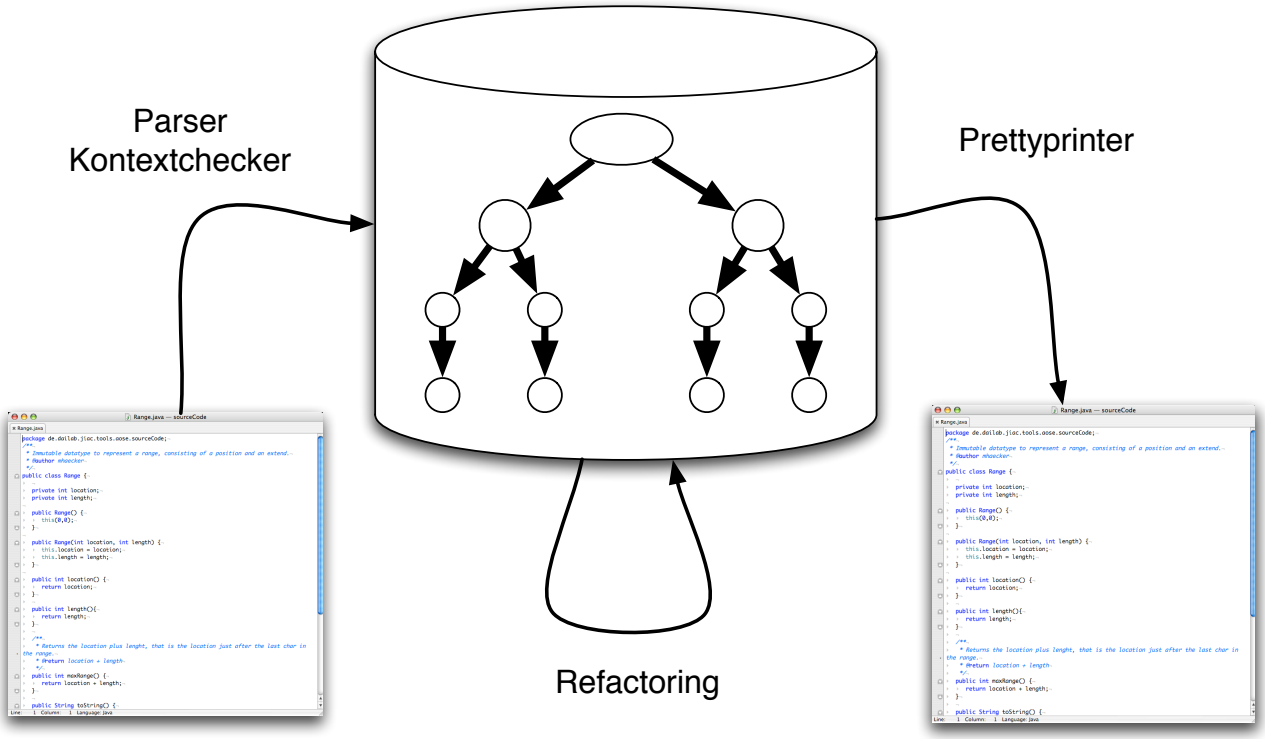
---

1 RedHat 7.1 - http://www.dwheeler.com/sloc/redhat71-v1/redhat71sloc.html

2 Auf http://www.refactoring.com/tools.html sind zwar einige Werkzeuge für C++ aufgelistet, aber sie können nur begrenzt funktionieren, da sie den Preprozessor nicht verstehen. Damit können sie einerseits viele Refactorings in Anwesenheit von Preprozessoranweisungen nicht durchführen, oder für die Korrektheit der Refactorings nicht garantieren.

Das Problem

Ein Refactoring-Werkzeug arbeitet prinzipiell so, das es den vorhandenen Quelltext in eine Graphstruktur – den Abstrakten Syntaxbaum (AST) – parst, auf diesem Graph dann alle Transformationen (Refactorings) durchgeführt werden und der Quelltext dann aus diesem veränderten Graph wieder neu erzeugt wird.



Dieser Ansatz versagt bei Sprachen aus der C Familie, da der Sourcecode Preprozessoranweisungen enthalten kann und diese nach der Bearbeitung durch den Preprozessor verschwunden sind.

Ein Preprozessor arbeitet seinem Namen nach vor dem Kompilieren, indem er ausschließlich Textersetzungen vornimmt. Das ist genau das Problem, da diese Textersetzungen in keinem Zusammenhang mit den Grammatikregeln der entsprechenden Sprache stehen

müssen und daher auch an beliebigen Stellen in deren Statements vorkommen können. (siehe Beispiel rechts)

```
for(i=0;
#if BY_ROW
    i<r;i++)
        s+=a[k][i];
#elif BY_COL
    i<C;i++)
        s+=a[i][k];
#endif
```

Damit lässt sich keine Grammatik finden, die den Preprozessor an beliebigen Stellen in der Sprache zulässt - man kann also keinen Parser bauen, der C und Preprozessor in einen AST parsen kann.

Der Preprozessor muss also ausgeführt werden. Das bedeutet aber, dass viele Informationen im Sourcecode verloren gehen und am Ende der Werkzeugkette der Prettyprinter nicht mehr in der Lage ist, den originalen Sourcecode wieder zu erzeugen.

Man kann also weder den Preprozessor vor der Arbeit mit dem Werkzeug ausführen, noch den Sourcecode direkt parsen.

Andere Programmiersprachen

Java
In dieser Programmiersprache treten die erwähnten Probleme nicht bzw. kaum auf. Es existiert kein Preprozessor, die Sprache ist stark typisiert und Variablen haben ebenfalls einen Typ. Dadurch kann ein Refactoring-Werkzeug leicht statisch analysieren, an welchen Stellen im Programm welche Typen verwendet werden. Probleme entstehen lediglich an den Stellen, an denen im Programm die Introspektionsfähigkeiten von Java verwendet werden und wenn die Virtual Machine von Java genutzt wird, um verschiedene Programmiersprachen interagieren zu lassen (z.B. Python durch das Jython Projekt[3], das es erlaubt, Python-Objekte direkt von Java-Klassen abzuleiten oder aufzurufen). Lösungsansätze sind hier, die Verwendung der Introspektionsfähigkeiten anzumahnen und den Benutzer entscheiden zu lassen, oder aber die dort verwendeten “Strings” als Spezialfall zu erkennen und mit zu behandeln. In gleicher Weise wird auch sprachübergreifendes Refactoring behandelt, indem Refactoring-Werkzeuge für verschiedene Sprachen beginnen zusammenzuarbeiten. Erste Ansätze von diesen Konzepten werden in Eclipse[4] verwirklicht.

3 http://www.jython.org

4 http://www.eclipse.org

Smalltalk
Smalltalk ist stark typisiert, aber Variablen besitzen keinen Typ. Vor allem aber definiert sich ein Typ durch die Nachrichten, die er versteht, was zur Folge hat, dass Objekte vollständig polymorph verwendet werden können, sobald sie die selben Nachrichten verstehen. Daher ist es für ein Refactoring-Werkzeug oft nicht möglich festzustellen an welchen Stellen im Programm eine bestimmte Methode (polymorph) aufgerufen wird - und wo der Aufruf sich auf eine andere, gleichbenannte Methode bezieht. Hier hatten die ursprünglichen Implementatoren John Brandt und Don Roberts[5] die Idee, die dynamischen Fähigkeiten von Smalltalk zu nutzen. Sie setzten voraus, dass jeder Programmierer eine Testsuite hat, die eine ausreichende Abdeckung erreicht. Sie implementierten das Umbenennen einer Methode dann so, dass sie unter dem alten Namen eine “Umleitung” zurückließen, die zusätzlich den aufrufenden ab da permanent änderte, so dass er gleich die umbenannte, neue Methode aufrief. Der Ansatz war also heuristisch - funktionierte aber doch so gut, dass dieses Werkzeug weltberühmt und Vorreiter für alle heute verfügbaren Werkzeuge wurde.

Lösungsansätze für die C-Sprachfamilie
Alejandra Garrido arbeitet an der Universität von Illinois derzeit an diesem Problem (Garrido 2002, 2003a, 2003b und 2005). Ihr Ansatz ist es, einen Pseudo-Preprozessor auszuführen, der die Textersetzung des Preprozessors nur teilweise durchführt, alle Alternativen erhält und damit den Schritt von der Baumstruktur zurück zur Textrepräsentaton ermöglicht.

Diese Idee macht Refactoring mit Sprachen aus der C-Familie möglich - bringt aber auch eine Menge Detailprobleme mit sich. Das Hauptproblem ist, dass Preprozessoranweisungen in C-Sprachen an beliebigen Stellen im Programm erlaubt sind - insbesondere an beliebigen Stellen innerhalb von Statements - und damit jede Grammatik sprengen.

5 CRefactory das ursprüngliche Refactoring Werkzeug: http://st-www.cs.uiuc.edu/users/brant/Refactory/

Das klassische Beispiel ist dieses Stück Code, das je nachdem, ob ein Preprozessor-Symbol gesetzt ist, eine Matrix nach Reihen oder nach Zeilen verarbeitet wird (vielleicht als Geschwindigkeitsoptimierung). Hier wird innerhalb der Definition der for-Schleife eine Alternative eingeführt und es existieren daher zwei mögliche zweite Teile der Schleife. Das an beliebigen Stellen im Code zuzulassen, würde bedeuten, exponentiell viele Grammatikregeln zu erhalten und Eindeutigkeit nicht mehr garantieren zu können. Dieses Problem vermeidet Garrido, indem sie einen Zwischenschritt einführt, den "Statement-Completer" (Garrido 2003b), der in dem Beispiel rechts den Beginn der for-Schleife in das Preprozessor-Statement hineinzieht und so zwei komplette Statements in der Alternative hinterlässt, aber markiert, wo diese Vervollständigung stattgefunden hat.

```
for(i=0;
#if BY_ROW
    i<r;i++)
        s+=a[k][i];
#elif BY_COL
    i<C;i++)
        s+=a[i][k];
#endif
```

```
#if BY_ROW
for(i=0; i<r;i++)
    s+=a[k][i];
#elif BY_COL
for(i=0; i<C;i++)
    s+=a[i][k];
#endif
```

Damit ist es möglich einen Parser zu bauen, der diesen veränderten Sourcecode parsen und in einer Graph-Struktur repräsentieren kann. Vor allem aber ist es möglich, einen Pretty-Printer zu bauen, der diese Markierungsinformationen nutzen kann, um diesen Schritt rückgängig zu machen.

Dieser Parser muss aber zusätzlich zu der normalen Grammatik der C-Sprachfamilie den Preprozessor integrieren. Einige der dafür notwendigen Anpassungen sind:

Alternative Definitionen von Code und Makros
Per "Conditional Compilation", also mit #if, #ifdef und #endif ist es möglich, je nachdem welche Konstanten definiert sind oder welche konstanten Tests wahr ausfallen, verschiedene Varianten des Codes zu erstellen. Häufig wird diese Funktion verwendet, um Code portabel auf verschiedenen Betriebsystemen verwenden zu können. Hier muss der Parser einen mehrdimensionalen Syntaxbaum aufbauen, jeweils mit einem Marker, der Informationen darüber enthält, welche Bedingungen gelten müssen, damit diese Alternative verwendet werden soll. Damit kann das Refactoring-Werkzeug leicht feststellen, welche Alternativen von einer Veränderung, die es vornehmen soll, betroffen sind, und welche nicht betrachtet werden dürfen. Besonders interessant ist dabei die Variante, dass ein Symbol je

nach gültigen Preprozessordirektiven entweder als Funktion oder Makro definiert sein kann (getchar() aus stdio.h ist da ein typisches Beispiel). Hier wird klar, dass ein Werkzeug, das den Preprozessor nicht mit einbezieht, auf jeden Fall versagen muss.

```
#if SYSTEM == SYSV
    #define HDR "sysv.h"
#elif SYSTEM == BSD
    #define HDR "bsd.h"
#else
    #define HDR "default.h"
#endif

#include HDR
```

Ein Problem, das auftreten kann, ist rechts illustriert. Inkludiert werden müssen hier, je nach gerade gültiger Bedingung, verschiedene Header-Dateien. Der Pseudo-Preprozessor muss also das #include so expandieren, dass es letztlich drei davon gibt, wobei jedes mit der entsprechenden Bedingung markiert ist.

Finden von ungültigem Code

Ein gelegentlich verwendeter Trick, um unmögliche Bedingungen (oder solche, die nie eintreten können) zu markieren, ist es, sie mit ungültigem Code zu füllen. Tritt diese Bedingung dann auf, sieht der Benutzer einen Kompilierungsfehler (Siehe Beispiel rechts). Korrekt wäre natürlich die #error Preprozessoranweisung zu verwenden, die genau für diesen Zweck existiert, aber das passiert leider häufig nicht.

```
#ifdef emacs
#ifdef static
#ifndef STACK_DIRECTION
you
loose
-- must know STACK_DIRECTION
at compile-time
#endif
#endif
#endif
```

Hier kann der Parser nicht fortfahren - entweder er ermöglicht also dem Benutzer diese Bedingung als “unmöglich” zu definieren oder er findet selbst heraus, dass eine bestimmte Bedingung zur Kompilierzeit nie eintreten kann. Die beste Reaktion auf diesen Code wäre vermutlich, ein Refactoring anzubieten, das diesen Code in die korrekte Verwendung von #error umwandelt, oder den ungültigen Code mit einer #error Anweisung markiert, und dann das Ende des fraglichen Preprozessor-Conditionals findet und als Aufsetzpunkt verwendet, an dem der Parser weitermachen kann.

Anwenden von Definitionen

Generell lassen sich mit #define definierte Makros wie ein Funktionsaufruf vom Parser repräsentieren. Da aber die Makro-Aufrufe an beliebigen Stellen im Code vorkommen können, müssen sie - zumindest teilweise - an der aufrufenden Stelle eingesetzt werden. Man

```
#define BEGIN {
#define END }

if (x < y) BEGIN y = x; x++; END
```

spricht vom “Expandieren der Makros”. Hier ein Beispiel aus der Bourne-Shell, in dem der Autor versucht hat die Sprache C etwas an Pascal anzugleichen. Im Beispiel links muss der Pseudo-Preprozessor den Makro-Aufruf für BEGIN und END durchführen und die entsprechenden Tokens markieren, damit der Prettyprinter am Schluss in der Lage ist, sie wieder durch die Makros zu ersetzen.

Der Include-Graph
Die #include Anweisung ist sehr einfach zu parsen, sie führt lediglich dazu, dass eine Kante von der Datei, in der sie steht, zu der Datei, in der sie verwendet wird hinzugefügt wird.

Auch dass durch mehrfaches Inkludieren zwar jeweils verschiedene Alternativen im Preprozessor ausgewählt werden können ist kein Problem, da ja sowieso alle Alternativen gleichzeitig betrachtet werden.

Die Größe des AST

```
struct inode {
    struct address_space i_data;
#ifdef CONFIG_QUOTA
    struct dquot *i_dquot[MAXQUOTAS];
#endif
    struct list_head i_devices;
};
```

Die Verwendung des “Statement Completion” Algorithmus kann im schlimmsten Fall, wenn verschachtelte Bedingungen verwendet werden, dazu führen, dass der entstehende AST exponentiell aufgebläht wird.

Glücklicherweise tritt das aber vor allem in struct-Definitionen in der oben dargestellten Art auf, so dass man das Problem stark begrenzen kann, indem man den “Statement Completion” Algorithmus und den Parser so anpasst, dass er Alternativen auch innerhalb von structs erlaubt. In Experimenten konnte Garrido (2005) zeigen, dass mit dieser Maßnahme die Größe des AST nur um wenige Prozent wächst. (etwa 24% für GNU rm und 2% für GNU Flex). Was sich auch zeigte ist, dass Programmierer in der Regel sehr selten inkomplette CPP-Conditionals verwenden, sondern diese hauptsächlich durch den Pseudo-Preprozessor entstehen, der Makro-Aufrufe mit mehreren alternativen Definitionen in den Sourcecode einsetzt.

Wichtigstes Ergebnis ist aber, dass exponentielles Wachstum des AST nicht auftritt und damit der entstehende AST beherrschbar bleibt.

Spezifische Probleme von ObjC

Die Struktur von Objective-C bringt einige interessante Probleme mit sich. Die Sprache ist einerseits hoch dynamisch und fast alle Informationen bleiben auch zur Laufzeit zugänglich, aber die Sprache ist auch kompiliert und das bedeutet, dass zur Laufzeit keine Änderungen durchgeführt werden können, die den Sourcecode betreffen.

Die Sprache selbst ist stark typisiert und kann sowohl statisch als auch dynamisch typisierte Variablen verwenden, außerdem beherrscht Objective-C wie Smalltalk Polymorphie nur auf der Basis, dass Objekte gleiche Nachrichten verstehen.

Damit entsteht das Problem, dass in Objective-C nicht jeder Variablen ein eindeutiger Typ zugeordnet und damit unter Umständen nicht jede Klasse ermittelt werden kann, die von einem bestimmten Refactoring betroffen sein muss. Gleichzeitig ist die Lösung, die Brandt und Roberts für ihren Smalltalk Refactoring Browser angewendet haben, nicht praktikabel, da Obj-C als kompilierte Sprache zur Laufzeit ihren Sourcecode nicht verändern und auch gar nicht ohne weiteres feststellen kann, wer eine Funktion aufgerufen hat.

Ich halte es für möglich das bei kompletter Analyse des Sourcecodes mittels Flussanalyse in fast allen Fällen herauszufinden ist, welches konkrete Objekt eine Nachricht erhält, aber spätestens bei Verwendung der Reflection-Fähigkeiten von Obj-C ist das auf jeden Fall nicht mehr möglich.

Wie stark das die Praktikabilität eines Refactoring Werkzeuges für Obj-C einschränkt, oder ob es andere Lösungen für dieses Problem gibt, bleibt in einer zukünftigen Arbeit zu klären.

Beispiele

Das “rename” Refactoring

Klassischerweise ist die Vorbedingung für das “rename” Refactoring, dass der neue Name nicht mit irgend einem anderen Namen kollidiert, der im gleichen Namensraum gültig ist.

In Anwesenheit des Preprozessors muss diese Bedingung erweitert werden, jetzt ist auch wichtig, ob ein Refactoring nur für eine der Alternativen gültig ist oder auf alle angewendet werden muss. Garrido (2003b) nennt hier als Bedingungen, dass ein Programmelement, das mehrfach definiert ist, dann in nur einer Ausprägung verändert werden kann, wenn die Bedingungen exklusiv und keine Disjunktion sind. Sobald eine Bedingung aus einer Dis-

junktion besteht, müssen alle Definitionen eines Elements umbenannt werden. Ich halte das für falsch, da die Verwendung eines mehrfach definierten Elements ausschlaggebend sein muss für die Entscheidung, ob ein Element nur für sich, oder ob alle Definitionen eines Elements gleichzeitig umbenannt werden müssen. Konkret: Wird ein Element polymorph in mehreren Bedeutungen verwendet, dann müssen auch alle Definitionen eines Elements verändert werden, wenn es umbenannt werden soll. Nur wenn ein Name nicht polymorph verwendet wird, kann nur eine Bedeutung der Alternativen umbenannt werden.

Das “extract function” Refactoring
Dieses Refactoring wird auch als der “Rubikon” eines Refactoring-Werkzeuges bezeichnet, da es nur korrekt durchführbar ist, wenn das Werkzeug die Struktur der Sprache wirklich versteht.[6]

Die Standard-Vorbedingungen für dieses Refactoring sagen aus: dass eine Liste von Statements in eine Funktion konvertierbar sein muss, dass der Ort, an dem die Funktion abgelegt werden soll, eine Funktion definieren darf, und dass der neue Name nicht mit einem bestehenden kollidiert.

Diese Vorbedingungen müssen nach Garrido (2003b) so erweitert werden, dass eine Liste von Statements entweder keine Preprozessordirektiven, oder diese komplett enthält. Außerdem muss der Ort, an dem die neue Funktion definiert wird, nicht nur die Definition von Funktionen zulassen, sondern es müssen dort auch die gleichen Preprozessorbedingungen gelten wie an der Stelle, von der die Statements stammen.

```
int f1() {
    nelems++;
    #ifdef _C1
        q+= j;
        nelems -= q;
    #else
        nelems *= j;
    #endif
}
```

Auch hier bin ich nicht ganz einverstanden, da hier im Beispiel rechts (das auch Garrido verwendet) durchaus nur der markierte Bereich in eine eigene Funktion extrahiert werden kann. Es zieht lediglich nach sich, das die darauf folgende Zeile von “#else” zu “#ifndef _C1”, also der Negation der ursprünglichen Bedingung, modifiziert wird.

Ausblick

Alejandra Garrido konnte zeigen, dass es zwar schwierig, aber durchaus möglich ist den Preprozessor der C-Sprachfamilie mit einem Pseudo-Preprozessor so zu bearbeiten, dass er mit der “Wirtssprache” so kompatibel wird, dass er zusammen mit ihr in einen AST ge-

[6] The Refactoring Rubicon: http://martinfowler.com/articles/refactoringRubicon.html

parst werden kann. Verbleibende Probleme sind hauptsächlich die tatsächliche Umsetzung dieses Ergebnisses in ein Werkzeug und die Erweiterung dieser Arbeit auf andere Sprachen der C-Familie, hauptsächlich auf C++ und Objective-C. Die Erweiterung auf C++ ist dabei konzeptuell relativ einfach, da Klassen dort letztlich nur spezielle Formen von Structs sind und der Template-Mechanismus auf seine Art als funktionale Sprache recht gut definiert ist. Die Erweiterung auf Objective-C bereitet mehr Kopfschmerzen, da es für das Problem der Typbestimmung bei Verwendung der nachrichtenbasierten Polymorphie bisher keine Lösung gibt, gleichzeitig dieses Prinzip in der Sprache aber sehr tief verankert ist und auch extensiv genutzt wird.

Quellen

Garrido, Alejandra, 2002. Challenges of Refatoring C Programs. Proceedings of the Internationional Workshop on Principles of Software Evolution. Orlando, Florida. https://netfiles.uiuc.edu/garrido/www/papers/refactoringC.pdf - Stand 7.8.05

Garrido, Alejandra, 2003a. Program Refactoring in the Presence of Preprocessor directives. https://netfiles.uiuc.edu/garrido/www/prelim.pdf - Stand 7.8.05

Garrido, Alejandra 2003b. Refactoring C with Conditional Compilation. 18th IEEE International Conference on Automated Software Engineering. Montreal, Canada. https://netfiles.uiuc.edu/garrido/www/papers/ASE_paper.pdf - Stand 7.8.05

Garrido, Alejandra, 2005. Analyzing Multiple Configurations of a C Program. Proc of the 21st. International Conference of Software Maintenance. Budapest, Hungary. https://netfiles.uiuc.edu/garrido/www/prelim.pdf - Stand 7.8.05