Refaktorisierungen für funktionale Programme

Holger Siegel

Seminarvortrag im Wintersemester 2006/07
Betreuer: Prof. Dr. Michael Hanus

Inhaltsverzeichnis

Zusammenfassung

Dieser Text ist die schriftliche Fassung eines Seminarvortrags, den ich im Wintersemester 2006/07 am Lehrstuhl für Programmiersprachen und Übersetzerkonstruktion der Christian-Albrechts-Universität Kiel halte. Er basiert auf dem Buch *Refactoring oder: wie Sie das Design vorhandener Software verbessern* von Martin Fowler[Fow05], sowie auf dem Artikel *Refactoring Functional Programs* von Simon Thompson [Tho05]. Er soll eine Einführung in das Refaktorisieren funktionaler Programme bieten. Neben der allgemeinen Vorgehensweise beim Refaktorisieren eines Programms werden einige Refaktorisierungen für funktionale Programme sowie kurz das Refaktorisierungs-Tool HaRe vorgestellt.

Große Software-Projekte entwickeln sich im Laufe der Zeit. Hat man eine erste lauffähige Version fertiggestellt, kommen bald neue Anforderungen hinzu, die die Software erfüllen soll. Dabei stellt sich oft heraus, dass Entwurfsentscheidungen, die in einer frühen Phase der Softwareentwicklung getroffen wurden, für die erweiterte Version nicht mehr angemessen sind. Es wird notwendig, die Software umzustrukturieren.

Die Erfahrung zeigt, dass diese Umstrukturierungen allzu gerne vernachlässigt werden, weil es kurzfristig wichtiger erscheint, das neue Feature zum Laufen zu bringen. In der Folge wird das Projekt mit jeder Erweiterung unübersichtlicher und schwerer zu warten, und es wird immer schwieriger, neue Features einzufügen. Dann wird es Zeit, die Struktur der Software behutsam aufzuräumen.

In den letzten Jahren wurden vor allem für die objektorientierte Programmierung Techniken entwickelt, vorhandenen Programmcode zu verbessern, ohne dabei ein funktionierendes Programm zu zerstören. Diese Techniken nennt man *Refaktorisierungen*. Martin Fowler hat mit seinem Buch *Refactoring oder: wie Sie das Design vorhandener Software verbessern* [Fow05] eine umfassende Anleitung zum Refaktorisieren von Java-Programmen vorgelegt. Viele der dort vorgestellten Refaktorisierungen werden von aktuellen Integrierten Entwicklungsumgebungen wie *Eclipse* oder *NetBeans* automatisch durchgeführt.

Was bei der objektorientierten Programmierung Standard ist, steckt bei der funktionalen Programmierung leider noch in den Kinderschuhen. In *Refactoring Functional Programs* [Tho05] stellt Simon Thompson nicht nur eine Vielzahl von Refaktorisierungen für die funktionale Programmiersprache Haskell vor, sondern auch ein Tool, das die Refaktorisierung von Haskell-Programmen automatisieren soll, HaRe – The Haskell Refactorer.

Dieser Text soll eine Einführung in das Refaktorisieren funktionaler Programme bieten. Zunächst wird auf die allgemeine Vorgehensweise beim Refaktorisieren eines Programms eingegangen. Dann werden einige Refaktorisierungen vorgestellt, mit denen man funktionale Programme refaktorisieren kann, die in der Programmiersprache Haskell geschrieben sind. Abschließend wird kurz das Refaktorisierungs-Tool HaRe vorgestellt und versucht, einen Ausblick auf die zukünftige Entwicklung dieses Themas zu bieten.

Alle Codebeispiele sind in der Programmiersprache Haskell geschrieben.

1 Das Refaktorisieren von Software

In diesem Text beziehe mich auf die Definition von Martin Fowler [Fow05, S. 41]:[1]

Refaktorisierung (Substantiv): Eine Änderung an der internen Struktur einer Software, um sie leichter verständlich zu machen und einfacher zu verändern, ohne ihr beobachtbares Verhalten zu ändern.

[1] Wie Bernd Kahlbrand, der Martin Fowlers Buch ins Deutsche übersetzt hat, übersetze ich die Begriffe *Refactoring* und *to refactor* mit *Refaktorisierung* bzw. *refaktorisieren*. Manchmal werden im Deutschen auch die Begriffe *Refaktorierung* und *refaktorieren* verwendet.

Refaktorisieren (Verb): Eine Software umstrukturieren, ohne ihr beobachtbares Verhalten zu ändern, indem man eine Reihe von Refaktorisierungen anwendet.

Das Ziel des Refaktorisierens ist, kurz gesagt, eine Verbesserung des Designs eines Programms, durch die das Programm verständlicher gemacht und die Fehlersuche vereinfacht wird.

Man identifiziert Programmteile, die man besser gestalten könnte, und ändert sie Zug um Zug, bis man mit dem Design des Programms zufrieden ist. Dabei werden Fehler des ursprünglichen Designs eliminiert. Umständlich implementierte Programmteile werden durch einfachere Lösungen ersetzt. Verstreute und mehrfach vorhandene Funktionalität wird an einer Stelle zusammengefasst.

Dadurch wird das Programm oft kürzer. Es bekommt eine klarere, verständlichere Struktur. Es wird leichter, Fehler zu suchen oder neue Funktionalität hinzuzufügen.

Ein Programm zu refaktorisieren ist nicht nur eine Investition in die Zukunft: Schon während des Refaktorisierens kann es vorkommen, dass man Programmfehler findet, die vorher verborgen waren. Auch wird man mit unbekanntem Code schneller vertraut, wenn man ihn refaktorisiert, anstatt zu versuchen, ihn durch bloßes Draufschauen zu verstehen.

Das Ziel des Refaktorisierens ist nicht, das Laufzeitverhalten des Programms zu optimieren. Auch dient es nicht dazu, die Funktionalität des Programms zu ändern.

Die Optimierung des Laufzeitverhaltens hat ihren Platz am Ende der Programmentwicklung. Dann identifiziert man diejenigen Programmteile, die einen Einfluss auf das Laufzeitverhalten haben, und ändert diese gezielt. – Nach der bekannten Faustregel, dass das Programm 90% der Rechenzeit in 10% der Programmzeilen verbraucht, betrifft diese Änderung nur einen kleinen Teil des Codes. Das Refaktorisieren hingegen dient dazu, den gesamten Code besser zu strukturieren. Dabei kann es sogar sein, dass das Programm zunächst langsamer wird. Nach den Erfahrungen, von denen Martin Fowler berichtet, ist es am Ende oft trotzdem schneller, weil man in dem klareren Code die Optimierungsmöglichkeiten besser einschätzen kann.

Während des Refaktorisierens lässt man die Funktionalität des Programms unverändert. So wird sichergestellt, dass beim Refaktorisieren keine Fehler unterlaufen. Ebenso lässt man vorhandenen Code unangetastet, während man neue Funktionen hinzufügt. Kent Beck hat die *Metapher der zwei Hüte* geprägt: bei der Programmentwicklung trägt man entweder den Hut des Refaktorisierers, der nur den vorhandenen Code umstrukturiert, oder den Hut des Programmierers, der nur neuen Code hinzufügt. Martin Fowler schreibt dazu:

> „Wenn Sie Software entwickeln, werden Sie wahrscheinlich feststellen, dass Sie diese beiden Hüte häufig wechseln. Sie versuchen

- Duplizierter Code
- Lange Methode
- Große Klasse
- Lange Parameterliste
- Divergierende Änderungen
- Schrotkugeln herausoperieren
- Neid
- Datenklumpen
- Neigung zu elementaren Typen
- Switch-Befehle
- Parallele Vererbungshierarchien
- Faule Klasse
- Spekulative Allgemeinheit
- Temporäre Felder
- Nachrichtenketten
- Vermittler
- Unangebrachte Intimität
- Alternative Klassen mit verschiedenen Schnittstellen
- Unvollständige Bibliotheksklasse
- Datenklassen
- Ausgeschlagenes Erbe
- Kommentare

Abbildung 1: Eine Liste der Smells nach M. Fowler

> zunächst eine neue Funktion hinzuzufügen. Dann stellen Sie fest, dass dies viel einfacher wird, wenn Sie den Code neu strukturieren. So wechseln Sie den Hut und refaktorisieren eine Weile. Wenn der Code besser strukturiert ist, setzen Sie den anderen Hut wieder auf und fügen die neue Funktion ein. Nachdem Sie die neue Funktion erfolgreich eingefügt haben, stellen Sie fest, dass Sie sie so geschrieben haben, dass sie kaum zu verstehen ist. So wechseln Sie wieder den Hut und refaktorisieren. Dies alles mag nur zehn Minuten dauern, aber Sie sollten sich immer im Klaren darüber sein, welchen Hut Sie gerade tragen." [Fow05, S. 42]

Damit man beim Refaktorisieren nicht versehentlich die Funktionalität eines Programmes ändert, ist es ratsam, das Programm nach jedem Refaktorisierungsschritt zu testen.

1.1 Smells

Es gibt keine präzisen Kriterien, wann man Code refaktorisieren soll. Jedoch gibt es Situationen, in denen eine Refaktorisierung fast immer angebracht ist. Kent Beck und Martin Fowler haben für solche Situationen den Begriff der „Smells" (in der deutschen Übersetzung: übel riechender Code) geprägt. In Martin Fowlers Buch *Refactoring* stellen sie 22 Smells vor [Fow05, S. 67]. Abbildung 1 zeigt eine Liste dieser Smells. Einige betreffen nur das Design

objektorientierter Programme; andere sind mit der gebotenen Vorsicht auch auf funktionale Programme anwendbar. Ein paar von diesen möchte ich an dieser Stelle vorstellen:

Schrotkugeln herausoperieren. Wenn das Programm an mehreren Stellen geändert werden muss, um eine einzige Eigenschaft zu verändern, dann kann es angebracht sein, diese Programmteile an einer Stelle zusammenzuführen. In einem objektorientierten Programm würde man die betroffenen Methoden und Felder in eine einzige Klasse verschieben, oder zu diesem Zweck eine neue Klasse einführen. In einer funktionalen Programmiersprache wie Haskell, die Module unterstützt, kann man z.B. die betroffenen Definitionen in einem Modul zusammenfassen.

Duplizierter Code. Martin Fowler schreibt hierzu:

> „Nummer Eins in der Gestanksparade ist duplizierter Code. Wenn Sie die gleiche Codestruktur an mehr als einer Stelle finden, können Sie sicher sein, dass Ihr Programm besser wird, wenn Sie einen Weg finden, diese zu vereinigen." [Fow05, S. 68]

Duplizierter Code macht das Programm unübersichtlich und verstellt den Blick auf die Gemeinsamkeiten von Programmteilen. Das Programm ist schwer zu warten, weil jede einzelne Änderung an mehreren verstreuten Stellen durchgeführt werden muss.

Spekulative Allgemeinheit verkompliziert den Code unnötig. Beim Programmieren muss man Sonderfälle beachten, von denen niemand weiß, ob man sie jemals braucht. Dadurch wird es schwieriger, diejenigen Features einzubauen, von denen man weiß, dass man sie braucht.

Kommentare werden oft unnötig, wenn man den Code verständlicher macht, zum Beispiel durch 'sprechende' Funktionsnamen oder eine aufgeräumte Programmstruktur.

Lange Methoden sind schwer zu verstehen, außerdem versteckt sich oft duplizierter Code in überlangen Methoden. Hier ist es meistens angebracht, Teile des Codes in Unterprogramme auszulagern, denen man aussagekräftige Namen gibt.

1.2 Design-Strategien für funktionale Programmiersprachen

Im Gegensatz zur objektorientierten Programmierung, in der man zunächst eine Klassenstruktur entwirft, die man dann mit der Implementation der

Methoden füllt, besteht die Entwicklung eines funktionalen Programms oft aus der Erstellung einer Reihe von Prototypen. Simon Thompson:

> „If we accept the final reason [dass funktionale Programme als eine sich entwickelnde Reihe von Prototypen gebaut werden, hs], which appears to be closest to existing practice, we are forced to ask how design emerges. A general principle is the move from the concrete to the abstract, and from the specific to the general." [Tho05]

Simon Thompson nennt dort die folgenden sechs Strategien, nach denen ein Haskell-Programm refaktorisiert werden kann:

Generalisation Eine Funktion, die für einen bestimmten Zweck geschrieben wurde, wird verallgemeinert, indem ein Teil ihres Verhaltens in einen Parameter umgewandelt wird.

Eine Funktion, die aus einer Liste das erste Auftreten eines Listenelementes `x` löscht, kann man wie folgt implementieren:

```
delete               :: Eq a => a -> [a] -> [a]
delete x []          = []
delete x (y:ys)      = if x == y then ys else y : delete x ys
```

Mit dieser Funktion kann man nur solche Elemente löschen, die Instanzen der Typklasse Eq sind. Außerdem ist man auf die Vergleichsfunktion `(==)` dieses Elementtyps festgelegt. Es ist nicht möglich, eine eigene Vergleichsfunktion zu verwenden.

Im Haskell-Standardmodul `List` findet sich die folgende generalisierte Variante, die dem Anwender ermöglicht, eine eigene Vergleichsfunktion `eq` anzugeben:

```
deleteBy             :: (a -> a -> Bool) -> a -> [a] -> [a]
deleteBy eq x []     = []
deleteBy eq x (y:ys) = if x `eq` y then ys
                                   else y : deleteBy eq x ys
```

Die Funktion `delete` erscheint nun als Spezialfall der generalisierten Funktion `deleteBy`:

```
delete               :: Eq a => a -> [a] -> [a]
delete               =  deleteBy (==)
```

Higher-order functions Diese Form von *Generalisation* ist charakteristisch für moderne funktionale Programmierung: Das spezifische Verhalten der Funktion wird aus der Funktion herausabstrahiert und ihr als Parameter übergeben. Hat man zum Beispiel eine Funktion sum geschrieben, die mittels einer Hilfsfunktion `sum_rec` eine Liste von Ganzzahlen aufaddiert

```
sum :: [Integer] -> Integer
sum xs = sum_rec 0 xs
    where sum_rec s []     = s
          sum_rec s (x:xs) = sum_rec (s + x) xs
```

dann kann man die Funktion `sum_rec` aufteilen in einen allgemeinen Teil `fold_rec`, der über die Liste iteriert, und in einen spezifischen Teil `add`, der angibt, wie die Listenelemente verknüpft werden:

```
sum xs = fold_rec add 0 xs
    where fold_rec f z []     = z
          fold_rec f z (x:xs) = fold_rec f (f z x) xs
          add a b = a + b
```

Ersetzt man nun die Funktion `fold_rec` durch die identische Bibliotheksfunktion `foldl`, dann hat man den rekursiven Aufruf komplett eliminiert. Es ergibt sich

```
sum xs = foldl add 0 xs
    where add a b = a + b
```

bzw. noch einfacher

```
sum = foldl (+) 0
```

Commonality Sind zwei Programmteile identisch oder wenigstens ähnlich, dann können sie durch Aufrufe einer einzigen Funktion ersetzt werden. Oft versucht man, durch *Generalisation* und andere Refaktorisierungen Gemeinsamkeiten zweier Funktionen herauszuarbeiten, um dann die gemeinsamen Programmteile in einer einzigen Funktion zu verschmelzen.

Data abstraction Konkrete algebraische Datentypen sind zwar meistens ein guter Ausgangspunkt, um den Prototypen eines Programms zu implementieren, aber dieser Prototyp ist schwer zu modifizieren, wenn sich die Struktur der Daten ändert. Der Wechsel zu abstrakten Datentypen ermöglicht es dann, die Implementation vollständig von der Verwendung zu trennen. Verwendet man außerdem noch Module, um Schnittstellen zu diesen Daten zu definieren, erreicht man eine Datenkapselung ähnlich dem Klassenkonzept der objektorientierten Programmierung. In Abschnitt 2.2 auf Seite 13 wird diese Programmtransformation an einem Beispiel vorgeführt.

Overloading. Hat man ähnliche Funktionen für mehrere Datentypen implementiert, dann kann man diese Funktionen durch die Einführung einer Klassendefinition zu einer einzelnen überladenen Funktion zusammenfassen. Als Ergebnis wird das Programm lesbarer, und durch das Hinzufügen von instance-Deklarationen wird es leicht, das Programm so zu erweitern, dass es auch mit weiteren Datentypen funktioniert.

Monadification In Haskell erlauben Monaden, Funktionen um Berechnungen zu ergänzen, die im Hintergrund ablaufen, oder imperative Effekte rein funktional zu formulieren. Indem man eine Funktion, die im funktionalen Stil geschrieben wurde, im monadischen Stil umschreibt, kann man Teile der Berechnung, z.B. die Behandlung von Fehlern, im Hintergrund von der Monade durchführen lassen und so von der eigentlichen Berechnung trennen. Ebenso kann man eine herkömmliche Funktion so umschreiben, dass man ihr beliebige Monaden „unterschieben" kann. Damit wird es leicht, einen rein funktionalen Programmteil durch Seiteneffekte zu ergänzen oder z.B. eine deterministische Funktion in eine nichtdeterministische umzuwandeln. In Abschnitt 2.2 auf Seite 17 wird diese Transformation an einem Beispiel vorgeführt.

1.3 "Hin zu eine Katalog von Refaktorisierungen"

Martin Fowler hat in seinem Buch *Refactoring* einen Katalog von Refaktorisierungen erstellt [Fow05, S. 105]. Darin hat er 72 unterschiedliche Refaktorisierungen detailliert beschrieben; für jede einzelne das

Einsatzgebiet, also die Situation, in der sie in Betracht kommt, die

Motivation, also eine Erläuterung der Gründe, warum diese Refaktorisierung das Programm verbessern kann, das

Vorgehen, also welche Bedingungen erfüllt sein müssen, damit sie überhaupt anwendbar ist, und was man bei der Umsetzung zu beachten hat, und zuletzt ein

Beispiel, das das Ergebnis an konkretem Programmcode zeigt.

Dieser Katalog hat sich als nützliche Sammlung von Werkzeugen erwiesen. Der Programmierer, der eine Refaktorisierung durchführen will, kann darin schnell nachschlagen, was dabei zu beachten ist. Wer ohne konkretes Ziel in dem Katalog blättert, entdeckt vielleicht Möglichkeiten, seinen Code umzustellen, die ihm vorher nicht bewusst waren oder zu aufwändig erschienen.

Die Refaktorisierungen in Martin Fowlers Buch sind auf die Programmiersprache Java zugeschnitten. Unter `http://www.cs.kent.ac.uk/projects/refactor-fp/catalogue/` haben Klaus Reinke, Simon Thompson und Huiqing Li einen Katalog von Refaktorisierungen für die Sprache Haskell erstellt.

Title: der Name der Refaktorisierung

Identity: ein eindeutiger Bezeichner, z.B. `AddArgument`

Category: Die Kategorie, zu der diese Refaktorisierung gehört

Classifiers: Schlüsselwörter

Internal cross references:

External cross references: Querverweise außerhalb dieses Katalogs, z.B. zu Fowlers Refaktorisierungen

Language: Die Sprache, in der diese Refaktorisierung mit Sicherheit anwendbar ist – in diesem Katalog also meistens Haskell

Description: Eine kurze Beschreibung der Refaktorisierung

`Beispielcode`

- `vor der Links-nach-rechts-Refaktorisierung`
- `nach der Rechts-nach-links-Refaktorisierung`

`Beispielcode`

- `vor der Rechts-nach-links-Refaktorisierung`
- `nach der Links-nach-rechts-Refaktorisierung`

General comment: Allgemeine Kommentare, einschließlich einer Beschreibung der Funktionsweise

Left to right comment: Kommentar zur Links-nach-rechts-Refaktorisierung

Right to left comment: Kommentar zur Rechts-nach-links-Refaktorisierung

Left to right conditions: Bedingungen für die Links-nach-rechts-Refaktorisierung

Right to left conditions: Bedingungen für die Rechts-nach-links-Refaktorisierung

Analysis required: Welche Analysen müssen durchgeführt werden, um die Bedingungen dieser Refaktorisierung zu überprüfen?

Abbildung 2: Aufbau einer Katalogseite von `http://www.cs.kent.ac.uk/projects/refactor-fp/catalogue/`

In Abbildung 2 auf der vorherigen Seite findet sich das Schema einer Katalogseite. Abbildung 3 auf der nächsten Seite zeigt ein Beispiel aus diesem Katalog.

Wie man dort sieht, sind Refaktorisierungen oft bidirektional; zu jeder Refaktorisierung gibt es ein Gegenstück, das diese rückgängig macht.

2 Refaktorisierungen für funktionale Programme

In diesem Abschnitt werden einige Refaktorisierungen vorgestellt. Zunächst strukturelle Refaktorisierungen, die die Struktur des Programms verändern, also Namen, Parameter und Blockstruktur der Funktionen. Dann datenorientierte Refaktorisierungen, die die Struktur der Daten und die Typen der definierten Funktionen betreffen. Zuletzt wird kurz auf modulorientierte Refaktorisierungen eingegangen, die die Aufteilung des Programms auf mehrere Quelltextmodule vereinfachen sollen.

2.1 Strukturelle Refaktorisierungen

delete, duplicate, rename So einfach diese grundlegenden Refaktorisierungen erscheinen, so schwer sind sie doch per Hand durchzuführen: will man eine Funktion umbenennen, dann muss man nicht nur die Stelle ändern, an der sie definiert ist, sondern man muss auch alle Stellen finden, an denen diese Funktion benutzt wird, und dort den Code ändern. Ebenso muss man sicherstellen, dass eine Funktion, die man löscht, nicht noch irgendwo in Gebrauch ist. Ist das Programm auf mehrere Quellcode-Module verteilt oder erlaubt die Sprache, dass gleichnamige Definitionen einander verdecken können, dann stößt man mit einem Texteditor schnell an die Grenzen.

Darum unterstützt heutzutage jede integrierte Entwicklungsumgebung das Löschen, Duplizieren und Umbenennen von Definitionen – letzteres oft direkt, die ersten beiden durch das automatische Aufspüren der Codestellen, an denen eine Funktion aufgerufen wird.

promote, demote Sprachen wie Haskell bieten die Möglichkeit, Definitionen in let-Blöcken zu schachteln. *promote* verschiebt eine Definition in einen äußeren Block, das Gegenstück *demote* in einen inneren.

promoted

```
foo x = sqr (x + 1)
sqr x = x * x
```

demoted

```
foo x = let
          sqr y = y * y
        in sqr (x + 1)
```

Man beachte, dass sich die Definition von `bar` im folgenden Beispiel nicht promoten lässt, da `bar` die lokale Variable `x` verwendet:

Simple Folding

Identity: Folding

Category: Naming

Classifiers: simple folding specialisation expression definition

Internal cross references:

External cross references: XXX

Language: Haskell

Description: Replace an instance of the right hand side of a definition by the corresponding left hand side. This is commonly known as folding, with the inverse termed unfolding or inlining.

```
showAll = (concat . format)
          . map show
table = concat . format
```

```
showAll = table . map show
table = concat . format
```

General comment: The fold transformation of Burstall and Darlington we call a generative fold in that it generates a new recursive definition of a function or constant.

Left to right comment: In the example, an instance of the definition of table is replaced by a call to table itself.
Note that the fold is not performed within the definition of table itself, nor in the body of any function called within the definition of table. A fold of this form will not change the meaning of the program.

Right to left comment: The inverse is an example of unfolding or inlining in which a call to a function or constant is replaced by the appropriate instance of its right hand side.

Left to right conditions: The definition which is folded needs to be in scope at the point of unfolding (this may be prevented by a redefinition of the identifier in an intervening scope).

Right to left conditions: At the site of the unfolding the bindings for the free identifiers of the RHS of the unfolded definition (of `foo`, say) need to be the same as in the definition of `foo`.

Analysis required: Static analysis of bindings; call graph.

Abbildung 3: Ein Beispiel aus dem Katalog der University of Kent

```
foo x = let
          bar y = x * y
        in bar (x + 1)
```

Die dem Promoting ähnliche Transformation *Lambda-Lifting* fügt in solchen Situationen zusätzliche Parameter ein, bis die Definition nicht mehr von lokalen Variablen abhängt. In unserem Beispiel führt das zu:

```
foo x = bar x (x + 1)
bar x y = x * y
```

Das Gegenstück zum Lambda-Lifting nennt sich *Lambda-Dropping*.

introduce definition, unfold definition Wenn ein Ausdruck unübersichtlich groß ist, empfiehlt es sich, Teilausdrücken beschreibende Namen geben. Hierzu dient *introduce definition*. Das Gegenstück *unfold definition* ersetzt in einem Ausdruck eine Variable durch den an sie gebundenen Wert.

Original

```
foo x = sqr (succ x)
succ x = x + 1
```

unfold definition(*succ*)

```
foo x = sqr (x + 1)
succ x = x + 1
```

Original

```
foo x = sqr (x + 1)
```

introduce definition(*succX*)

```
foo x = let
          succX = x + 1
        in sqr succX
```

add argument, remove argument Will man einer Funktion einen Parameter hinzufügen oder einen unbenutzten Parameter entfernen, muss man alle Stellen, an denen diese Funktion aufgerufen wird, ändern. Deshalb ist es sinnvoll, wenn die integrierte Entwicklungsumgebung die Refactorings *add argument* und *remove argument* unterstützt.

added argument y

```
bar x y = x + 1
...
foo = 27 + bar 2 3
```

removed argument y

```
bar x = x + 1
...
foo = 27 + bar 2
```

generalise definition Man verallgemeinert eine Funktion, indem man einen Teilausdruck in einen Wert umwandelt, der ihr durch einen neuen Parameter übergeben wird.

Original

```
incAll xs = map (+ 1) xs
...
foo xs = sum (incAll xs)
```

generalised definition

```
incAllBy n xs = map (+ n) xs
...
foo xs = sum (incAllBy 1 xs)
```

2.2 Datenorientierte Refaktorisierungen

from concrete to abstract type

Mit data-Deklarationen und Pattern Matching hat man in Haskell schnell ein funktionierendes Programm geschrieben. Wenn man es dann erweitert, stößt diese Implementierung bald an ihre Grenzen:

- Die Datenstrukturen liegen für jeden offen; ändert man die Implementierung eines Datentyps, muss man alle Programmteile ändern, die diese Daten verwenden.
- Es gibt keine Möglichkeit, „intelligente" Konstruktoren zu definieren, die zusätzliche Informationen im Datenobjekt speichern oder bestimmte Eingaben ausschließen.

Die Lösung ist, den konkreten Datentyp in einen abstrakten Datentyp umzuwandeln. Diese Refaktorisierung besteht aus mehreren kleineren Refatorisierungen:

- Anstelle der Konstruktoren werden Konstruktor-Funktionen angeboten,
- Fallunterscheidungen werden statt durch Pattern-Matching mittels Diskriminator-Funktionen wie *isNode*, *isLeaf* durchgeführt,
- der Zugriff auf Datenfelder wird statt durch Pattern-Matching durch Zugriffsfunktionen durchgeführt.

Im Folgenden wird an einem einfachen Beispiel gezeigt, wie diese Refaktorisierung schritttweise durchgeführt wird. Das folgende Modul implementiert eine einfache Baumstruktur:

```
module BinaryTree(Tree(..), depth) where

data Tree = Leaf Int
          | Node Tree Tree
```

In einem anderen Modul gibt es eine Funktion, die die Tiefe eines Binärbaums ermittelt:

```
depth :: Tree -> Int
depth (Leaf _)   = 1
depth (Node l r) = 1 + max (depth l) (depth r)
```

add field labels In einem ersten Schritt werden Zugriffsfunktionen ergänzt, mit denen auf Datenfelder zugegriffen werden kann. Haskell bietet die Möglichkeit, die Felder von Tupeltypen zu benennen. Aus diesen Feldnamen werden automatisch Zugriffsfunktionen erzeugt. Es genügt also, die Felder des Datentyps `Tree` zu benennen:

```
data Tree = Leaf {leaf1 :: Int}
          | Node {node1 :: Tree, node2 :: Tree}
```

add discriminators Dann werden Diskriminator-Funktionen ergänzt, durch die man zwischen Leafs und Nodes unterscheiden kann. Dadurch wird es möglich, Fallunterscheidungen ohne Pattern-Matching zu formulieren.

```
isLeaf :: Tree -> Bool
isLeaf (Leaf _) = True
isLeaf _        = False

isNode :: Tree -> Bool
isNode (Node _ _) = True
isNode _          = False
```

add constructors Um die Implementierung des Datentyps vor dem Anwender zu verstecken, werden Konstruktorfunktionen eingeführt.

```
mkLeaf :: Int -> Tree
mkLeaf = Leaf

mkNode :: Tree -> Tree -> Tree
mkNode = Node
```

eliminate patterns Jetzt müssen die Definitionen, die den Datentyp Tree verwenden, so umgeschrieben werden, dass sie stattdessen die neu eingeführten Funktionen verwenden. Pattern-Matching wird durch Diskriminator- und Zugriffsfunktionen ersetzt, Konstruktoren durch Konstruktorfunktionen. Die Funktion `depth` sieht dann so aus:

```
depth t | isLeaf t = 1
depth t | isNode t = 1 + max (depth (node1 tree)) (depth (node2 t))
```

eliminate nested patterns In diesem Beispiel war es einfach, das Pattern-Matching loszuwerden. Weniger offensichtlich ist, wie man mit geschachtelten Patterns umgeht. Hat man zum Beispiel einen Ausdruck

```
foo (Bar [x]) = x + 3,
```

dann muss man zunächst das geschachtelte Pattern zu einem Case-Ausdruck auflösen:

```
foo (Bar xs) = case xs of [x] -> x + 3
```

create an ADT module Zuletzt wird die Export-Liste des Moduls so angepasst, dass nicht mehr die Konstruktoren des Datentyps, aber dafür die neu eingeführten Funktionen exportiert werden. Das Ergebnis ist ein vollständig gekapselter Datentyp, dessen Implementierung man jederzeit ändern kann:

```
module BinaryTree(Tree, mkLeaf, mkNode, isLeaf, isNode,
                  leaf1, node1, node2) where
...
```

Jetzt können wir, ähnlich einer Klassenmethode in der objektorientierten Programmierung, die Funktion `depth` in das Modul `BinaryTree` verschieben.

Memoisation

Unter *Memoisation* versteht man die Technik, den Rückgabewert einer Funktion zwischenzuspeichern, anstatt die Funktion wieder und wieder zu berechnen. Die Einführung von *Memoisation* ist also keine Refaktorisierung im engeren Sinne, sondern eine Laufzeitoptimierung. Trotzdem ist sie als eine Änderung der Datenrepräsentation, die das beobachtbare Verhalten nicht verändert, mit den Refaktorisierungen eng verwandt. Ich führe sie hier kurz vor, um zu zeigen, wozu die aufwändige Transformation im obigen Beispiel nützlich sein kann.

Da die Implementierung des Typs `Tree` vor dem Anwender versteckt ist, kann man jedem Node `n` ein Feld hinzufügen, in dem das Ergebnis des Aufrufs `depth n` gespeichert wird. Dabei wird ausgenutzt, dass Haskell einen Ausdruck erst dann auswertet, wenn sein Ergebnis auch wirklich gebraucht wird:

```
data Tree = Leaf {leaf1 :: Int}
          | Node {node1 :: Tree, node2 :: Tree, d :: Int}

mkNode l r = Node l r (1 + max (depth l) (depth r))
```

Für einen Node muss dann nur noch das Feld `d` ausgelesen werden, um die Tiefe zu ermitteln:

```
depth t | isLeaf t = 1
depth t | isNode t = d t
```

Dank der verzögerten Auswertung von Haskell wird die Tiefe eines Nodes erst dann berechnet, wenn er gebraucht wird. Bei zukünftigen Aufrufen wird dann der gespeicherte Ergebniswert verwendet. Durch eine minimale und lokal begrenzte Änderung des Codes wurde also ein effektiver Memoisation-Mechanismus implementiert.

layered data type

Im folgenden Beispiel fällt auf, dass in der Funktion `eval` viermal die selbe Funktionalität implementiert ist: werte die beiden Subterme aus und verknüpfe sie.

```
data Term = Con Integer
          | Add Term Term
          | Sub Term Term
          | Mul Term Term
          | Div Term Term

eval :: Term -> Integer
eval (Con n) = n
eval (Add t u) = (eval t) + (eval u)
eval (Sub t u) = (eval t) - (eval u)
eval (Mul t u) = (eval t) * (eval u)
eval (Div t u) = (eval t) `div` (eval u)
```

Hier kann es angebracht sein, anstelle eines einzelnen Datentyps einen zweistufigen Datentyp einzuführen. Der Datentyp `Term` besitzt dann nur noch die beiden Konstruktoren `Con` und `Bin` für Konstanten bzw. zweistellige Terme. Letztere werden anhand eines Elementes `Op` unterschieden.

```
data Term = Con Integer
          | Bin Term Op Term
data Op = Add | Sub | Mul | Div
```

Entsprechend wird die ursprüngliche Funktion `eval` aufgeteilt in eine Funktion `sys`, die die Zwischenergebnisse verknüpft, und eine Funktion `eval`, die die Teilterme auswertet und die Verknüpfung der Ergebnisse an die Funktion `sys` delegiert:

```
sys Add = (+); sys Sub = (-)
sys Mul = (*); sys Div = div

eval :: Term -> Integer
eval (Con n) = n
eval (Bin t op u) = sys op (eval t) (eval u)
```

Wie man sieht, ist der Auswerter übersichtlicher geworden, und es ist jetzt einfacher, neue Operatoren hinzuzufügen.

monadification 1

Ist der Ergebnistyp einer Funktion Instanz der Typklasse `Monad`, dann kann man die Funktion manchmal so umschreiben, dass die konkrete Instanz der Monade austauschbar wird. Aus

```
sqrtMaybe :: Double -> Maybe Double
sqrtMaybe x | x < 0  = Nothing
            | x >= 0 = Just (sqrt x)
```

wird dann

```
sqrtM :: Monad m => Double -> m Double
sqrtM x | x < 0  = fail "negativer Radikand"
        | x >= 0 = return (sqrt x)
```

monadification 2

Nicht ganz zufällig ist der im vorletzten Beispiel entstandene Termauswerter identisch mit dem, den Ralf Hinze in seinem Haskell-Kurs unter `http://www.informatik.uni-bonn.de/~ralf/teaching/Hskurs_7.html#SEC49` angibt. Dort führt er vor, wie man diesen Auswerter um verschiedene Funktionalitäten erweitert: Fehlerbehandlung, Reduktionszähler, eine Protokollierung oder Ausgabe der durchgeführten Operationen. All diese Erweiterungen werden haben gemeinsam, dass sie die eigentliche Auswertung nicht beeinflussen, sondern Berechnungen im Hintergrund durchführen. Er führt vor, dass eine naive Umsetzung dieser Erweiterungen ein komplettes Umschreiben des Auswerters erfordert, und dass man sich dieses Umschreiben erspart, wenn man den Auswerter zunächst in monadische Form umwandelt. Dabei bleibt der Datentyp Term unverändert, aber der Typ der Auswertungsfunktion eval ist nicht mehr `Term -> Integer`, sondern `Term -> m Integer` für beliebige Monaden `m`:

```
sys :: Monad m => Op -> m Integer -> m Integer -> m Integer
```

```
sys Add = liftM2 (+); sys Sub = liftM2 (-)
sys Mul = liftM2 (*); sys Div = liftM2 div

eval :: Monad m => Term -> m Integer
eval (Con n) = return n
eval (Bin t op u) = sys op (eval t) (eval u)
```

Dabei ist `liftM2` eine Bibliotheksfunktion von Haskell, die eine Funktion auf zwei monadische Argumente anwendet. Sie ist wie folgt definiert:

```
liftM2 :: (Monad m) => (a1 -> a2 -> r) -> m a1 -> m a2 -> m r
liftM2 f m1 m2 = do { x1 <- m1; x2 <- m2; return (f x1 x2) }
```

Das Vorgehen ist also:

- Ist m der Typ der Monade, dann wird der Ergebnistyp r durch den monadischen Ergebnistyp $m\,r$ ersetzt. Im Beispiel wurde der Ergebnistyp von `eval` von `Integer` nach `m Integer` geändert und der Typ von `sys` an diese Änderung angepasst.
- Werte werden mit `return` in die Monade gehoben.
- 1- bis 5-stellige Funktionsanwendungen werden mit den Bibliotheksfunktionen `liftM, liftM2, ..., liftM5` in die Monade gehoben.

Diesen monadifizierten Auswerter kann man leicht erweitern, z.B. indem man die Funktion `sys` mit einer Fehlerbehandlung ergänzt:

```
sys Add = liftM2 (+); sys Sub = liftM2 (-)
sys Mul = liftM2 (*);
sys Div a b = do a' <- a
                 b' <- b
                 if b'==0 then fail "division by zero"
                         else return (a' `div` b')
```

Man beachte, dass dieses Beispiel mit jeder Monade funktioniert, da in Haskell für jede Instanz der Typklasse Monad die Funktion `fail` zur Verfügung steht.

Maybe to List or Either

Oft verwendet man den Datentyp `Maybe`, um zwischen den beiden Fällen zu unterscheiden, dass entweder ein Wert oder gar kein Wert vorliegt. Tritt nun die Notwendigkeit auf, auch im zweiten Fall einen Ergebniswert zuzückzugeben, dann ist es einfach, den Typ `Maybe t` gegen den Typ `Either t`

auszutauschen. Dann ersetzt der Konstruktor `Just x` den Konstruktor `Left x`, sowie `Right y` den Konstruktor `Nothing`, wobei `y` der Wert ist, der neuerdings übergeben werden soll. Dabei zahlt es sich aus, wenn man statt expliziten Pattern-Matchings überall die Bibliotheksfunktion `maybe` verwendet hat. Denn dann muss man nur noch die Ausdrücke der Gestalt `maybe n f x` durch `either f (const n) x` ersetzen.

Wird es notwendig, statt eines möglichen Ergebnisses mehrere zu übergeben, dann kann man den Typ `Maybe t` durch den Listentyp `[t]` erstzen. Statt `Nothing` übergibt man dann die leere Liste `[]`, statt `Just x` die einelementige Liste `[x]`. Hier zahlt es sich aus, wenn man vorher die obige Refaktorisierung *monadification 2* anwendet und den Code, der Daten von Typ `Maybe` verwendet, monadisch formuliert. Denn sowohl `Maybe` als auch der Listentyp sind Instanzen der Typklasse `Monad`; `return x` liefert im einen Fall `Just x`, im anderen Fall die einelementige Liste `[x]`, `fail _` liefert `Nothing` bzw. die leere Liste `[]`. Entsprechend wendet der mondische Bind-Operator `xs >>= f` die Funktion `f` auf alle Werte der Argumentliste an und liefert die Liste der Ergebnisse. Es reicht also schon, die Typsignatur von `Maybe t` nach `[t]` zu ändern. Will man dann statt eines Wertes x mehrere Werte $x_1, \ldots, x_n$ verwenden, ersetzt man den Ausdruck `return x` durch die Liste $[x_1, \ldots, x_n]$.

delete, duplicate type definition, introduce type/newtype

Auch für Datentypen gibt es diese grundlegenden Refaktorisierungen. Typdeklarationen können umbenannt, gelöscht oder dupliziert werden. Teile von Typausdrücken können durch type- oder newtype-Deklarationen benannt werden. Dabei gilt wie für die strukturellen Refaktorisierungen *delete*, *duplicate*, *rename*, dass im gesamten Programm nach zu ändernden Stellen gesucht werden muss.

enumerated type

Eine Anzahl von Konstanten wird durch eine data-Deklarationen mit einer entsprechenden Anzahl von 0-stelligen Konstruktoren ersetzt. Verwendet man z.B. ganzzahlige Konstanten, um eine Eigenschaft anzuzeigen, dann ist es meistens ratsam, diese Refaktorisierung durchzuführen. Statt

```
type Color = Int
red = 0; green = 1; blue = 2
```

verwendet man dann

```
data Color = Red | Green | Blue
```

und es ist automatisch gewährleistet, dass nur zulässige Werte auftreten.

type generalisation

Hat man eine Funktion für einen bestimmten Zweck geschrieben, dann kann es vorkommen, dass ihre Typdefinition enger gefasst ist, als es notwendig wäre. Hat man eine Funktion `delete :: Integer -> [Integer] -> [Integer]` geschrieben, die aus einer Liste von Ganzzahlen das erste Vorkommen eines Elements löscht, dann liegt es nahe, diese Funktion so zu verallgemeinern, dass sie nicht nur mit Ganzzahlen verwendbar ist, sondern mit allen Typen, für die die Vergleichsfunktion `(==)` definiert ist. Das Ergebnis dieser Refaktorisierung ist die auf Seite 6 angegebene Funktion `delete`.

Neben der Wiederverwendbarkeit gibt es noch einen Grund, möglichst allgemeine Typdefinitionen anzugeben: generische Typen können anzeigen, auf welche Weise die Funktion ihre Parameter miteinander verknüpft. Eine Funktion

```
foo :: Int -> Int -> Int
```

kann alles Mögliche mit den beiden Argumenten machen, während die Funktion

```
bar :: a -> b -> a
```

nichts anderes als die Bibliotheksfunktion const sein kann (wenn man davon absieht, dass sie möglicherweise ihre Argumente strikt auswertet). Ebenso gibt die Signatur

```
foldl :: (a -> b -> a) -> a -> [b] -> a
```

einen Hinweis darauf, wie `foldl` die Listenelemente verknüpft.

2.3 Module und modulorientierte Refaktorisierungen

clean import list, explicit import list, add/remove import Während man an einem Modul arbeitet, verlängert sich die Liste der importierten Funktionen; man verliert den Überblick darüber, von welchen externen Definitionen das Modul abhängig ist. Weil Refaktorierungs-Tools naturgemäß mit der Bindungsstruktur eines Programms umgehen können müssen, bietet es sich an, dass sie die Import-Liste automatisch so klein wie möglich halten. In der integrierten Entwicklungsumgebung Eclipse läuft dieser Mechansmus fast unbemerkt von Programmierer im Hintergrund.

move definition Eine Definition zwischen zwei Modulen zu verschieben, ist aufwändig. Die Import- und Export-Listen aller Module, die diese Funktion verwenden, müssen angepasst werden. Hier kann ein Refaktorisierungs-Tool dem Programmierer die Last abnehmen, den gesamten Quellcode nach diesen Stellen zu durchforsten.

3 Das HaRe-System

In der objektorientierten Programmierung sind Refaktorisierungs-Tools schon lange Standard. Seit 1992 gibt es für Smalltalk den Refactoring-Browser, der einen für die erste Generation der Refaktorisierungs-Tools beachtlichen Leistungsumfang aufweist. Eine Version, die mit VisualWorks zusammenarbeitet, wird unter `http://st-www.cs.uiuc.edu/users/brant/Refactory/` angeboten.

Für Java bieten die großen integrierten Entwicklungsumgebungen umfangreiche Möglichkeiten; allein das Refactor-Menü der Entwicklungsumgebung Eclipse bietet 21 automatische Refaktorisierungen vom simplen Umbenennen von Variablen über das Verschieben von Methoden zwischen Klassen bis hin zur Extraktion neuer Methoden aus markierten Codestellen.

Für das Refaktorisieren von Haskell gab es bis vor kurzem keine Tool-Unterstützung; auch Entwicklungsumgebungen wie Eclipse behandeln Haskell eher stiefmütterlich. Dem soll ein Refaktorisierungs-Tool für Haskell abhelfen, das derzeit an der University of Kent entwickelt wird. Es trägt den Namen HaRe – The Haskell-Refactorer. Die Entwicklung ist Teil des Projektes *Refactoring Functional Programs*, dem auch der weiter oben erwähnte Katalog von Refaktorisierungen entstammt. Die Homepage des Projektes ist `http://www.cs.kent.ac.uk/projects/refactor-fp/`. Dort wird auch die aktuelle Version von HaRe angeboten – derzeit Version 0.3. Bisher unterstützt es nur den Standard Haskell98, wird aber erweitert und soll einmal den gesamten Sprachumfang des aktuellen GHC beherrschen[2].

Das Programm HaRe stellt selbst keinen Editor zur Verfügung, mit dem man Quellcode bearbeiten oder anzeigen könnte. Hierzu verwendet es die Editoren VIM und EMACS. Diese Editoren verfügen über interne Skriptsprachen, mit denen man ihre Funktionalität erweitern kann. Während einer HaRe-Sitzung läuft im Hintergrund das eigentliche Refaktorisierungs-Programm und wartet auf Befehle des Editors. Abbildung 4 auf der nächsten Seite zeigt das Refaktorisierungs-Menü, um das HaRe den Editor EMACS erweitert. Dort ist der Menüpunkt *From concrete to abstract data type* angewählt. Diese Refaktorisierung führt in einem Schritt all die Refaktorisierungen durch, die in Abschnitt 2.2 auf Seite 13 verwendet wurden, um einen konkreten Datentyp in einen abstrakten umzuwandeln.

Ein Parser, der Formatierungen und Kommentare erhält Bevor man ein Programm refaktorisieren kann, muss es es zunächst eingelesen und in einen abstrakten Syntaxbaum umgewandelt werden. Für die Sprache Haskell gibt es mehrere Compiler, deren Quellcode-Parser man hierfür verwenden könnte. Der Glasgow Haskell Compiler bietet dem Anwender so-

[2]Es ist mir nur gelungen, das Update vom 15.04.2005 mit GHC 6.2 zum Laufen zu bringen. Das Update vom 20.01.2006 enthält Fehler und scheint nicht mit dem aktuellen GHC 6.6 verwendbar zu sein.

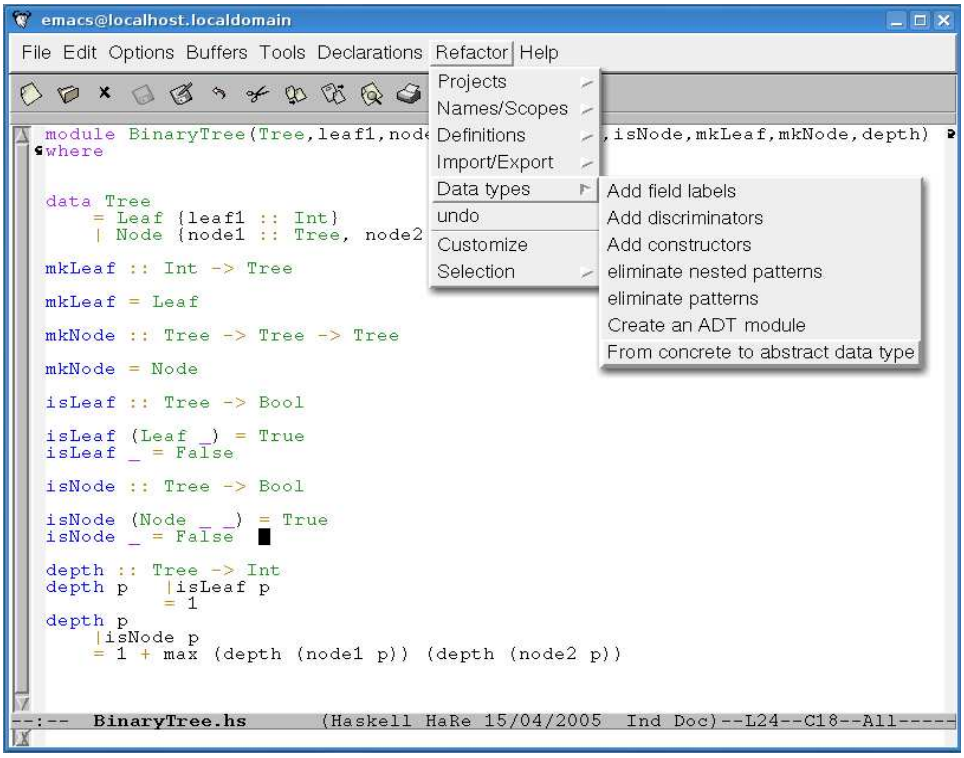

Abbildung 4: eine HaRe-Sitzung in Emacs

gar standardmäßig Bibliotheken an, mit denen er Haskell-Code parsen, als abstrakten Syntaxbaum verarbeiten und formatiert ausgeben kann. Diese vorhandenen Lösungen haben aber den Nachteil, dass sie beim Einlesen des Syntaxbaumes alle Textformatierungen und Kommentare ignorieren, die im Quellcode vorhanden sind. Sie speichern lediglich für einen Teil der syntaktischen Konstrukte deren Startposition im Quellcode.

Es ist schon lästig, wenn man von einem Entwicklungswerkzeug einen bestimmten Formatierungsstil vorgeschrieben bekommt, – ein Refaktorisierungs-Tool, das die gesamte Quellcode-Dokumentation löscht, ist unzumutbar. Aus diesem Grund konnte man für HaRe nicht einfach einen der vorhandenen Parser verwenden, um den zu refaktorisierenden Code einzulesen.

Anstatt den immensen Aufwand auf sich zu nehmen, einen neuen Parser zu schreiben, der die Formatierungen und Kommentare als Teil der Grammatik behandelt, haben die Entwickler von HaRe einen einfacheren Weg gewählt: Neben dem abstrakten Syntaxbaum eines Quellcode-Moduls wird sein *token stream* gespeichert, also die Folge syntaktischer Einheiten, aus denen der abstrakte Syntaxbaum erstellt wurde. In diesem *token stream* sind noch alle Formatinformationen, Quellcodepositionen und Kommentare enthalten. Die Bestandteile des abstrakten Syntaxbaum enthalten ebenfalls

Positionsinformationen, durch die man jedem als Syntaxbaum vorliegenden Ausdruck einen Abschnitt des *token streams* zuordnen kann. Mithilfe dieser Information läßt sich nach der Refaktorisierung die ursprüngliche Quellcode-Formatierung und -Kommentierung wieder herstellen. Wurden dabei im abstrakten Syntaxbaum Unterbäume verschoben, dann wird die alte Formatierung an der neuen Stelle verwendet, da die Positionsinformationen des Syntaxbaumes nach wie vor auf die alte Stelle des *token streams* verweisen. Eine ausführliche Erklärung dieses Vorgehens, einschließlich der Schwierigkeiten, die beim Erzeugen neuer Codeteile auftreten, findet sich unter [Li06, S. 57].

Auf diese Weise war es möglich, das HaRe-System auf der Implementierung des Parsers und des abstrakten Syntaxbaums aufzubauen, die das Programmpaket Programatica (siehe `http://www.cse.ogi.edu/PacSoft/projects/programatica/`) bereitstellt.

4 Ausblicke

M. Fowlers Liste der Smells ist auf die objektorientierte Programmierung zugeschnitten. Manche Punkte kann man auf die funktionale Programmierung übertragen, andere wiederum nicht. Eine vergleichbare Liste, die auf die spezifischen Probleme der funktionalen Programmierung eingeht, ist noch zu schreiben.

Ein Wunschtraum der Entwickler des Refaktorisierungs-Tools HaRe ist die Möglichkeit, komplexe Refaktorisierungen aus atomaren zusammenzusetzen. An einer *domain specific language*, mit der der Anwender aus den vorhandenen Refaktorisierungen größere zusammenbauen kann, wird derzeit geforscht.

Literatur

[Fow05] Martin Fowler. *Refactoring oder: wie Sie das Design vorhandener Software verbessern.* Addison-Wesley, 2005 (ISBN 3-8273-2278-2).

[Tho05] Simon Thompson. *Refactoring Functional Programs.* In *5th International School on Advanced Functional Programming*, Springer LNCS 3622, Seiten 331-357, 2005.

[Li06] Huiqing Li. *Refactoring Haskell Programs.* Ph.D. Thesis, 2006. Verfügbar unter `http://www.cs.kent.ac.uk/projects/refactor-fp/`.