# Debugging deklarativer Programme durch orakelgesteuerte strikte Auswertung
## Studienarbeit

Holger Siegel

Sommersemester 2007

**Zusammenfassung**

Dieser Text ist die schriftliche Ausarbeitung einer Studienarbeit, die ich im Sommersemester 2007 am Lehrstuhl für Programmiersprachen und Übersetzerkonstruktion der Christian-Albrechts-Universität Kiel angefertigt habe. Gegenstand dieser Arbeit ist, nach dem Prinzip der orakelgesteuerten strikten Auswertung nicht-strikter deklarativer Programmiersprachen, das hier an der CAU Kiel am Lehrstuhl für Programmiersprachen und Übersetzerkonstruktion entwickelt wurde, einen Debugger für eine Teilsprache von Haskell zu implementieren. Nach einer Darstellung der orakelgesteuerten strikten Auswertungsstrategie wird die Implementierung des Debuggers vorgestellt. Abschließend wird ein kurzer Ausblick auf mögliche Erweiterungen dieses Debuggers gegeben.

# Inhaltsverzeichnis

# 1 Deklaratives Debugging durch orakelgesteuerte strikte Auswertung

## 1.1 Debugging für deklarative Programmiersprachen

Deklarative Programmiersprachen mit nicht-strikter Auswertungsreihenfolge stellen besondere Anforderungen an Werkzeuge zur Fehlersuche. Im Gegensatz zu imperativen Sprachen, in denen die Programmausdrücke der Reihe nach abgearbeitet werden, ist die Reihenfolge der Auswertung nicht vorhersehbar; die Beobachtung der Programmausführung anhand eines Programmzählers und des jeweiligen Speicherzustandes würde die Funktionsweise des Programms eher verschleiern als darstellen.

Aus diesem Grund wurden eigene Techniken entwickelt, um deklarative Programme zu debuggen. Ein deklarativer Debugger zeigt nicht die zeitliche Abfolge der Berechnungsschritte an, sondern die auszuwerteten Ausdrücke und die Ergebnisse ihrer Auswertung, so dass der Anwender die Korrektheit dieser Ausdrücke beurteilen kann.

Dabei ist es nützlich, wenn man einem inkorrekten Ausdruck die Unterausdrücke zuordnen kann, von denen sein Wert abhängt, so dass man auch diese inspizieren und damit die Fehlerquelle weiter eingrenzen kann. Besitzt

die Sprache eine nicht-strikte Auswertungsstrategie, so ist diese Zuordnung nicht offensichtlich, weil bei der Auswertung auch Auswertungen außerhalb dieses Ausdrucks angestoßen werden können, die durch die nicht-strikte Auswertungsreihenfolge bisher aufgeschoben waren.

Der Debugger `Hat` (siehe `http://www.cs.york.ac.uk/fp/hat/`) ist ein solcher deklarativer Debugger für die Programmiersprache Haskell. Er ist ein *offline tracer*, d.h. er ermöglicht es, nach dem Ende eines Programmlaufs die ausgewerteten Teilausdrücke zu inspizieren. Hierfür erzeugt er während des Programmlaufs einen sogenannten *redex trail*: eine Datenstruktur, die sämtliche Funktionsaufrufe und deren Ergebnisse enthält. Dieser *redex trail* kann sehr groß werden. Dateigrößen von mehreren hundert Megabytes sind keine Seltenheit. Abgesehen davon, dass das Debuggen nichttrivialer Programme bald an der Festplattenkapazität scheitert, wird das interaktive Debuggen sehr langsam, weil ein Großteil der Zeit mit Dateizugriffen verbracht wird.

Es wäre also viel gewonnen, wenn man ohne *redex trails* auskäme oder zumindest deren Größe signifikant verringern könnte.

## 1.2 Orakelgesteuerte strikte Auswertung

Hier am Lehrstuhl für Programmiersprachen und Übersetzerkonstruktion der Christian-Albrechts-Universität Kiel wurde eine Methode entwickelt und getestet, mit der sich die *redex trails* durch eine Datenstruktur ersetzen lassen, die um mehrere Größenordnungen kleiner ist. Dabei wird, anstatt das Programm einmal auszuführen und dann die dabei gesammelten Daten auszuwerten, das Programm zweimal ausgeführt:

- Zunächst wird das Programm, das der Anwender untersuchen will, mittels verzögerter Auswertung ausgeführt. Die Auswertung findet dabei in der gleichen Reihenfolge statt wie bei der ursprünglichen nicht-strikten Ausführung des Programms. Der einzige Unterschied ist, dass während des Programmlaufs darüber Buch geführt wird, welche Teilausdrücke ausgewertet worden sind und welche nicht für die Berechnung des Endergebnisses benötigt wurden.

- Im zweiten Schritt wird das Programm dann in strikter Auswertungsreihenfolge ausgeführt. Dabei entscheidet die im vorigen Schritt gesammelte Information, das sogenannte *Orakel*, ob ein Ausdruck zu einem Ergebnis ausgewertet oder durch einen Platzhalter ersetzt wird.

Während dieser Ausführung werden dem Anwender die Ergebnisse der Funktionsauswertungen in der Reihenfolge präsentiert, in der sie nach der strikten Auswertungsreihenfolge zu erwarten wären, so dass er einen Eindruck von der Funktionsweise des Programms bekommt.

Nun wird auch klar, warum im Titel von orakelgesteuerter Auswertung die Rede ist: Ähnlich einem Orakel dienen die im ersten Schritt gesammelten Informationen im zweiten Schritt zur Vorhersage des Programmverhaltens.

In der Arbeit *Lazy Call-By-Value Evaluation* von Frank Huch, Bernd Braßel, Sebastian Fischer, Michael Hanus und German Vidal [Braßel et al. 2007] wird der Nachweis geführt, dass beide Programmausführungen tatsächlich zum selben Ergebnis kommen. Gleichzeitig wird darin eine effiziente und speichersparende Methode entwickelt, diese Technik der orakelgesteuerten strikten Auswertung zu implementieren.

## 1.3 Orakelgesteuertes Debugging: Ein Prototyp

Im Zuge dieser Arbeit wurde der Prototyp eines Debuggers erstellt, der nach diesem Prinzip der orakelgesteuerten strikten Auswertung verfährt. Er kann eine Schnittmenge der Programmiersprachen Haskell und Curry verarbeiten. Diese Schnittmenge besteht aus der in [Braßel et al. 2007] beschriebenen funktionalen Kernsprache, erweitert um die Möglichkeit, algebraische Datentypen einzuführen und Funktionen mittels Pattern-Matching zu definieren

Der hier entwickelte Debugger basiert auf der Transformation von Quellprogrammen, so dass die Debugging-Sitzung mit einer beliebigen vorhandenen Laufzeitumgebung durchgeführt werden kann. Er besteht aus drei Teilen, zwei Programmtransformationen und einer Laufzeitbibliothek:

`transform` ist ein in der Programmiersprache Curry implementiertes Programm, das das zu debuggende Programm einliest und so ergänzt, dass das resultierende Programm bei der Ausführung ein Orakel erzeugt. Abgesehen von der Erzeugung des Orakels verhält sich dieses transformierte Programm dabei genau wie das Originalprogramm.

`stricths` liest ebenfalls das ursprüngliche Programm ein und erzeugt daraus ein weiteres Programm in Form von Quellcode der Sprache Haskell. Dieses Programm unterscheidet sich vom ursprünglichen Programm dadurch, dass es ein Orakel entgegennimmt und die aufgerufenen Funktionen in strikter Auswertungsreihenfolge reduziert. Dabei entscheidet

das Orakel für jeden Programmausdruck, ob er ausgewertet oder durch einen Platzhalter ersetzt wird. Während dieser Auswertung kann der Benutzer Argumente und Ergebniswert jedes Funktionsaufrufs ausgeben lassen und, indem er die Korrektheit der Ergebnisse bewertet, den Programmfehler immer weiter eingrenzen, bis er die Fehlerstelle lokalisiert hat.

`StrictSteps.hs` ist eine Laufzeitbibliothek, die dem mittels `stricths` instrumentierten Programm diese Funktionalität zur Verfügung stellt. Sie enthält die Funktionen, die die orakelgesteuerte Auswertung des Programms implementieren, sowie die Funktionen, die das interaktive Debuggen ermöglichen.

Das Debuggen eines Curry-Programms `Example.curry` erfolgt in vier Schritten:

1. Das Programm `transform` erzeugt aus dem ursprünglichen Programm `Example.hs` ein Programm `ExampleOracle.fcy`.

2. Das Programm `ExampleOracle.fcy` wird ausgeführt. Es unterscheidet sich vom ursprünglichen Programm nur dadurch, dass es im Hintergrund die Datei `Example.steps` erzeugt. Diese Datei enthält das Orakel, das später beim interaktiven Debuggen verwendet wird.

3. Das Programm stricths erzeugt aus dem ursprünglichen Programm das Programm `ExampleStrict.hs`.

4. Das Programm `ExampleStrict.hs` wird ausgeführt. Es liest zuerst die Orakeldatei Example.steps. Dann führt es den von `stricths` instrumentierten Programmcode aus und fragt dabei interaktiv nach der Korrektheit der zurückgegebenen Funktionswerte.

Im folgenden Kapitel wird eine solche Debugging-Sitzung im Detail vorgeführt.

# 2 Eine Beispielsitzung

In Abbildung 1 auf der nächsten Seite ist das Beispielprogramm angegeben, das auch in der oben erwähnten Arbeit verwendet wird.[1]

[1]Es unterscheidet sich von dem in [Braßel et al. 2007] angegebenen Programm durch zwei kleine, technisch bedingte Ergänzungen: zum einen wurde die Deklaration der Funk-

```
data Nat = Z | S Nat
data NatList = Nil | Cons Nat NatList
main = length (take (S (S Z)) (fibs Z))
take Z _ = Nil
take (S x) (Cons y ys) = Cons y (take x ys)
length Nil = Z
length (Cons _ xs) = length xs
fibs x = Cons (fib x) (fibs (S x))
fib _ = error "XXX"
```

Abbildung 1: Beispielprogramm aus [Braßel et al. 2007]

Die Funktion *main* definiert zunächst mittels der Funktion *fibs* eine unendliche Liste von Funktionswerten einer Funktion *fib*.Mittels der Funktion *take* werden die ersten beden Elemente dieser Liste in eine neue Liste kopiert. Die Länge dieser Liste wird nun mittels der Funktion *length* ermittelt und als Ergebnis der Programmausführung zurückgegeben. Dabei werden die Funktionswerte von *fib* weder zur Programmausführung benötigt, noch tauchen sie im Endergebnis auf; aufgrund der verzögerten Auswertung bleiben sie daher während der gesamten Programmausführung unausgewertet.

Das Programm enthält jedoch einen kleinen, aber folgenreichen Programmierfehler: anstelle des erwarteten Wertes $(S\ (S\ Z))$ gibt es den Wert $Z$ zurück.

Anhand dieses Beispiels soll nun gezeigt werden, wie die vorliegende Implementierung verwendet werden kann, um ein fehlerhaftes Programm zu debuggen. Hierzu sei angenommen, dass sich das Programm unter dem Namen `Example.curry` im aktuellen Verzeichnis befindet, ebenso wie die zum Debugger gehörenden Dateien:

```
~/oracle> ls -sh Example*
4,0K Example.curry
```

tion *fib* ergänzt, zum anderen der Typ *NatList* als ein Nachbau des dort als gegeben vorausgesetzten generischen Listentyps hinzugefügt. Außerdem war es notwendig, in den tatsächlich verwendeten Programmdateien diejenigen Funktionsnamen, die mit den Namen von *Prelude*-Funktionen kollidieren, um einen Apostroph `'` zu ergänzen.

Zu Beginn wird die erste Programmtransformation durchgeführt, die das Programm um die Möglichkeit, ein Orakel zu erzeugen, ergänzt:

```
~/oracle> ./transform Example main
% restoring /home/pakcs/pakcs/curry2prolog/c2p.state...
Collecting Curry functions to be translated...
CompactFlat: Searching relevant functions in module Example...Prelude...
CompactFlat: Total number of functions (without unused imports): 196
CompactFlat: Number of functions after optimization: 12
```

Damit wurde im aktuellen Verzeichnis eine Datei namens `ExampleOracle.fcy` erzeugt. Sie stellt ein Curry-Programm im maschinenlesbaren FlatCurry-Format dar. Dieses Programm wird nun ausgeführt, um die Orakel-Datei zu erzeugen:

```
~/oracle> ./oracle Example main
% restoring /home/pakcs/pakcs/curry2prolog/c2p.state...
Result: (()) ?
```

Das zu debuggende Programm ist jetzt zum ersten Mal ausgeführt worden. Wenn dabei kein Laufzeitfehler aufgetreten ist, befindet sich im aktuellen Verzeichnis die Datei `Example.steps`. Sie enthält die folgende Liste von Ganzzahlen:

$$[2, 1, 0, 14]$$

Nun ist der erste – nicht-strikte – Teil der Debugging-Sitzung abgeschlossen. Als Nächstes wird die zweite Transformation auf das ursprüngliche Programm angewendet: Aus `Example.curry` wird das Programm `ExampleStrict.hs` erzeugt. Das geschieht mittels der folgenden Befehlszeile:[2]

```
~/oracle> ./stricths --hs Example
```

[2]Ist der Curry-Compiler `kics` installiert, dann kann das Programm `stricths` leicht aus dem Curry-Programm `stricths.curry` erzeugt werden. Hierfür stellt das in der Distribution enthaltene Makefile eine Regel zur Verfügung. Ein einfaches `make stricths` sollte genügen. Ist kein Curry-Compiler vorhanden, kann das Programm `stricths.curry` auch von einem beliebigen Curry-Interpreter ausgeführt werden.

Das Ergebnis ist die Haskell-Programmdatei `ExampleStrict.hs`. Sie enthält das für die Debugging-Sitzung vorbereitete Programm. Die Details dieser Transformation werden in Abschnitt 3 auf Seite 10 erklärt.

Jetzt kann die Debugging-Sitzung beginnen. Hierfür verwenden wir den Haskell-Interpreter `ghci`:

```
~/oracle> ghci ExampleStrict
```

Nachdem das Programm geladen ist, wird die instrumentierte Main-Funktion aufgerufen. Diese lädt automatisch die zuvor erzeugte Orakel-Datei Example.steps:

```
Loading package mtl-1.0.1 ... linking ... done.
 ____    ____    _____
(  _ \  (_  _)  (  _  )   Believe
 ) _ <   _)(_    )(_)(    in
(____/()(____)()(_____)() Oracles
--------type ? for help----------

  main ~> _
```

Der erste Funktionsaufruf wird ausgegeben, in diesem Fall die Funktion *main*. Das Ergebnis ist dabei noch nicht ausgerechnet. Durch Drücken von `?` (*help*) wird eine Liste der möglichen Eingaben angezeigt:

```
usage:
  r        inspect result
  s        skip
  v        toggle verbosity
  i        toggle inspect mode
  q        quit
  <SPACE>  step into
```

Wir drücken `r` (*result*), um das Ergebnis der Auswertung von *main* anzuzeigen:

```
main ~> Z
```

Da das Ergebnis $Z$ falsch ist – wir hatten den Wert $(S\ (S\ Z))$ erwartet –, geben wir `w` (*wrong*) ein. Darauhin fragt der Debugger nach der ersten Unterberechung von *main*:

```
(fiblist Z) ~> _
```

Auch hier ist das Ergebnis noch nicht ausgerechnet. Um das Ergebnis künftig gleich angezeigt zu bekommen, wechseln wir mit `i` in den *inspect mode*. Daraufhin wird das Ergebnis der Auswertung angezeigt:

```
(fiblist Z) ~> (Cons _ (Cons _ _))
```

Es bedeutet, dass der Ausdruck (*fibs Z*) zu einer mindestens zweielementigen Liste ausgewertet wird – was korrekt ist, da *fibs Z* sogar eine unendliche Liste ist. Die Elemente dieser Liste sind nicht ausgewertet, weil sie für die Berechnung des Endergebnisses nicht notwendig waren. Die Korrektheit des Endergebnisses hängt also nicht vom Wert dieser Listenelemente ab.

Darum geben wir `c` (*correct*) ein, um diesen Berechnungsschritt als korrekt zu bewerten. Mit dieser Eingabe werden sowohl der aktuelel Aufruf als auch alle Unteraufrufe als korrekt bewertet Der Debugger springt nun zur nächsten Unterberechnung von *main*:

```
(take' (S (S Z)) (Cons _ (Cons _ _))) ~> (Cons _ (Cons _ Nil))
```

Wir enthalten uns einer Beurteilung dieses Ergebnisses. Dafür antworten wir mit `s` (*skip*), wodurch die Beurteilung dieser Berechnung und aller Unterberechnungen übersprungen wird. Damit gelangen wir zum nächsten Berechnungsschritt:

```
(length' (Cons _ (Cons _ Nil))) ~> Z
```

Diese Berechnung ist offenbar falsch, da sie die Länge einer zweielementigen Liste mit 0 angibt. Daher antworten wir mit `w` (*wrong*). Nun wird nach der ersten Unterberechnung des als falsch bewerteten Funktionsaufrufs gefragt:

```
(length' (Cons _ Nil)) ~> Z
```

Wir enthalten uns einer Bewertung und antworten mit einem Leerzeichen (*step into*), um zu den Unterberechnungen dieses Ausdrucks zu gelangen. Die einzige Unterberechnung ist die folgende:

```
(length' Nil) ~> Z
```

Die Länge einer leeren Liste wird mit 0 angegeben. Das ist offenbar richtig. Also antworten wir mit `c` (*correct*). Damit sind wir am Ende des Programmdurchlaufs angekommen, ohne einen Fehler gefunden zu haben. Daher wird die Auswertung des zuletzt als falsch bewerteten Terms wiederholt; es wird nach dem ersten Funktionsausdruck gefragt, dessen Bewertung übersprungen wurde:

$$
\begin{aligned}
&\textbf{data}\ Nat = Z \mid S\ Nat\\
&inf = S\ inf\\
&min\ Z\ (S\ \_) = Z\\
&min\ (S\ \_)\ Z = Z\\
&min\ (S\ x)\ (S\ y) = S\ (min\ x\ y)\\
&main = min\ (S\ (S\ (S\ Z)))\ inf
\end{aligned}
$$

Abbildung 2: Minimalbeispiel `ExMinimum.curry`

```
(length' (Cons _ Nil)) ~> Z
```

Nun holen wir die Bewertung nach, der wir uns oben enthalten haben. Nach der Eingabe von `w` (*wrong*) ist klar, wo der Programmfehler liegt:

```
found bug in rule:
  lhs = (length' (Cons _ Nil))
  rhs = Z
```

# 3 Zur Implementierung des Debuggers

Mein Beitrag zu diesem Debugger besteht in der Laufzeitbibliothek `StrictSteps.hs`, die die eben vorgestellte interaktive Benutzerschnittstelle zur Verfügung stellt. Es war notwendig, die Transformation `stricths` an diese Laufzeitbibliothek anzupassen, damit die von `stricths` erzeugten Programme diese Laufzeitbibliothek verwenden können.

Im Folgenden sollen die Laufzeitbibliothek `StrictSteps.hs` und diejenigen Aspekte der Programmtransformation `stricths`, die zur Integration dieser Laufzeitbibliothek dienen, vorgestellt werden. Das Programm `ExMinimum.curry` aus Abbildung 2 soll dabei als Beispiel dienen.

Die Transformation `stricths` erzeugt aus diesem Programm ein Haskell-Programm `ExMinimumStrict.hs`. Es verwendet die von `StrictSteps.hs` exportierten Deklarationen. Dafür erhält es die folgenden Import- und Export-Deklarationen:

```
module ExampleStrict (
  Nat (..), showConsNat, inf, min, main) where
import StrictSteps
import Prelude
```

## 3.1 Codierung des Orakels

Ein Orakel ist eine Liste von Wahrheitswerten. Diese Liste wird beim Ausführen der Innermost-Variante schrittweise konsumiert. Hat der nächste Eintrag der Orakelliste den Wert *True*, dann wird die gemäß der strikten Auswertungsreihenfolge nächste Reduktionsschritt ausgeführt. Hat er den Wert *False*, dann wird dieser Reduktionsschritt übersprungen und anstelle des Ergebnisses der Auswertung wird ein Platzhalter-Wert *underscore* zurückgegeben.

Um diese Orakelliste platzsparend darzustellen, wird sie durch eine Liste natürlicher Zahlen codiert. Dabei entspricht eine Zahl $n$ einer Liste von $n$ *True*-Werten, gefolgt von einem *False*-Wert. Eine Ausnahme bildet dabei das letzte Element einer Orakelliste: Damit auch Orakellisten dargestellt werden können, deren letztes Element *True* ist, wird die Liste [0] als leere Liste von Wahrheitswerten interpretiert.

```
type Oracle = BoolStack
type BoolStack = [Int]
```

Die folgenden Funktionen spalten jeweils erste Element einer codierten Liste von dieser ab bzw. legen einen Bool-Wert auf eine solche Liste:

```
popBoolStack :: BoolStack → (BoolStack, Bool)
pushBoolStack :: BoolStack → Bool → BoolStack
```

Eine leere Orakelliste wird repräsentiert durch

```
emptyBoolStack :: BoolStack
emptyBoolStack = [0]
```

Eine unendliche Liste von *True*-Werten ist mit dieser Codierung leider nicht darstellbar. In der vorliegenden Implementierung wird zur Darstellung natürlicher Zahlen der Datentyp *Int* verwendet. Damit ist die größte Liste von

aufeinanderfolgenden *True*-Werten, die mit dieser Codierung darstellbar ist, durch folgende Deklaration gegeben:

$$allTrue :: BoolStack$$
$$allTrue = [maxBound]$$

**Beispiel** Für das Beispielprogramm wird als Orakel die folgende Liste erzeugt:

$$[5, 12]$$

Diese Liste repräsentiert gemäß der oben beschriebenen Codierung eine Liste von 18 Wahrheitswerten: auf fünf *True*-Werte folgen ein *False*-Wert und schließlich zwölf *True*-Werte. Der erste *True*-Wert gibt an, dass der gesamte Ausdruck *main* ausgewertet werden soll. Die nächsten vier *True*-Werte geben an, dass die vier ersten Elemente der unendlichen Liste *inf* berechnet werden. Der nun folgende Wert *False* läßt die Berechnung von *inf* an dieser Stelle abbrechen.[3] Der Rest der Liste besteht nur noch aus *True*-Werten. Sie geben an, dass das restliche Programm in strikter Auswertungsreihenfolge ausgeführt wird, ohne dass dabei die Berechnung weiterer Teilterme übersprungen wird.

## 3.2 Erweiterung algebraischer Datentypen

Die Transformation `stricths` erweitert die vom Programm deklarierten algebraischen Datentypen um jeweils einen neuen argumentlosen Konstruktor, der für einen unbekannten, unausgewerteten Term steht. Durch die Technik der orakelgesteuerten Auswertung ist gewährleistet, dass dieser Ausdruck für die Berechnung des Endergebnisses nicht benötigt wird.

Aus der Sicht der denotationellen Semantik entspricht *underscore* dem undefinierten Wert $\perp$. Und tatsächlich könnte man *underscore* durch den von der Sprache Haskell bereitgestellten Wert *undefined* darstellen. Dass man stattdessen die algebraischen Datentypen durch neue Konstruktoren ergänzt, die diesen Wert darstellen, hat einen pragmatischen Grund: Der Debugger hat keine Möglichkeit festzustellen, ob ein Ausdruck den Wert *undefined* hat.

[3] Auf die Darstellung zirkulärer Datenstrukturen wird in Abschnitt 4.1 auf Seite 23 eingegangen.

Bei der Anzeige von Funktionsergebnissen können jedoch Ausdrücke auftreten, die auch den Wert *underscore* enthalten. Beim Versuch, einen solchen anzuzeigen, würde der Debugger dann in eine Endlosschleife laufen oder mit einem Fehler abbrechen.

**Beispiel** Die Deklaration von *Nat* des Beispielprogramms wird um den Konstruktor *NatUnderscore* ergänzt:

> **data** *Nat* = *NatUnderscore* | *Z* | *S Nat* **deriving** *Show*

## 3.3 Darstellung von Konstruktortermen

Der Debugger muss die Möglichkeit haben, die Argumente und Resultate von Funktionsaufrufen als Text darzustellen. Üblicherweise wird hierzu die Typklasse *Show* verwendet, die für ihre Instanzen eine Funktion *show* zur Verfügung stellt. Diese Methode hat jedoch zwei Nachteile, aufgrund derer eine Verwendung in diesem Debugger nicht in Frage kommt: Zum einen kann das zu debuggende Programm selbst Instanzen von *Show* deklarieren, die nur einen Teil der jeweiligen Datenstruktur anzeigen. Zum anderen gibt es keine Möglichkeit, die Ausgabe der Funktion *show* zu begrenzen, so dass der Versuch, eine zyklische Datenstruktur anzuzeigen, in einer Programmschleife enden muss.

Deshalb wurde der Datentyp *ConstructorTerm* eingeführt, der für Tupeltypen eine einheitliche Baumdarstellung bereitstellt:

> **data** *ConstructorTerm* = *ConsTerm String* [*ConstructorTerm*]
> | *ConsUnderscore*

Der Konstruktur *ConsTerm* stellt Konstruktorterme dar, wie sie in der Deklaration des Datentyps definiert wurden. Das erste Argument gibt den Namen des Konstruktors an, das zweite eine Liste seiner Komponenten. Der Konstruktor *ConsUnderscore* steht für den Platzhalter *underscore*, der bei der orakelgesteuerten strikten Auswertung anstelle des Ergebnisses eines übersprungenen Funktionsaufrufs eingefügt wird.

Die Transformation `stricths` erzeugt für jeden algebraischen Datentypen $T$ eine Funktion *showCons*$T$, die einen Wert dieses Typs als Konstruktorterm des Typs *ConstructorTerm* darstellt.

**Beispiel** Für den Datentyp *Nat*, der im Beispielprogramm deklariert wird, sieht diese Funktion so aus:

```
showConsNat :: Nat → ConstructorTerm
showConsNat x0
  = case x0 of
    NatUnderscore → consUnderscore
    Z     → consTerm "Z" []
    S x1 → consTerm "S" [showCons x1]
```

**Die Klasse** *StrictCurry* *StrictCurry* ist die Klasse der im Debugger beobachtbaren Datenstrukturen. Für jeden im zu debuggenden Programm deklarierten Datentyp erzeugt die Transformation `stricths` eine Instanz-Deklaration. Sie verbindet den Datentyp mit der oben beschriebenen Darstellung der Konstruktorterme sowie der jeweiligen Repräsentation für übersprungene *underscore*-Werte.

```
class StrictCurry a where
  underscore :: a
  showCons :: a → ConstructorTerm
    -- Default-Implementierung
  underscore = error "I stumbled over an underscore"
  showCons _ = ConsUnderscore
instance StrictCurry (a → b) where
  showCons _ = ConsUnderscore
```

Die angegebene Default-Implementierung stellt sicher, dass der Debugger auch mit Datentypen, die nicht um einen *underscore*-Konstruktor erweitert werden konnten, umgehen kann. Für diese Datentypen würde die Auswertung von *underscore* zwar zu einem Laufzeitfehler führen, dies passiert aber nicht, da das Prinzip der orakelgesteuerten strikten Auswertung sicherstellt, dass *underscore* nur für solche Ausdrücke eingesetzt wird, die erwiesenermaßen nicht ausgewertet werden. Die Anzeige durch *showCons* ist trotzdem möglich, weil showCons unabhängig von seinem Argument das Ergebnis *ConsUnderscore* liefert.

Die Instanz-Deklaration für Funktionen ist notwendig, weil bei der Berechnung auch teilausgewertete Funktionen als Zwischenergebnisse auftreten können.

**Beispiel** Für den Datentyp *Nat* des Beispiels wird die folgende Instanz-Deklaration erzeugt:

```
instance StrictCurry Nat where
  underscore = NatUnderscore
  showCons = showConsNat
```

## 3.4 Programmzustand des Debuggers

Der Programmzustand des Debuggers teilt sich auf in einen globalen Zustand *DebuggerState*, der bei jedem Berechnungsschritt aktualisiert wird, und die Betriebsart *StepMode*, die jeweils für einen Unterausdruck angibt, auf welche Weise er ausgewertet werden soll.[4]

**Globaler Zustand** Der globale Zustand des Debuggers wird durch die folgende Datenstruktur repräsentiert:

```
data DisplayMode = DisplayMode{
  verbose :: Bool,
  optionalResult :: Bool}
```

Der *Displaymode* gibt den Anzeigemodus des Debuggers an. Die erste Komponente gibt an, ob ausführliche Statusmeldungen ausgegeben werden sollen. Die zweite gibt an, ob vor der Auswertung von Teilausdrücken die Möglichkeit gegeben werden soll, diese zu überspringen.

```
data DebuggerState = DebuggerState{
  oracle  :: Oracle,
```

[4]Es wäre auch möglich gewesen, die Betriebsart als eine Kompomente des globalen Zustands zu implementieren. Dann müßte die Betriebsart vor der Berechnung eines Unterausdrucks auf den jeweiligen Wert und nach Ende der Berechnung wieder auf den ursprünglichen Wert gesetzt werden. Das würde aber verschleiern, dass die Berechnungsschritte des Debuggers mit ihrer Betriebsart parametrisiert sind.

*displayMode* :: *IORef DisplayMode*,
*skipped* :: *BoolStack*,
*unrated* :: *BoolStack*}

*oracle* ist die Liste der verbleibenden Orakelwerte,

*displayMode* ist der oben beschriebene Anzwigemodus des Debuggers,

*skipped*, *unrated* geben an, welche der zu bewertenden Funktionsaufrufe noch unbewertet sind (*True*) bzw. als korrekt bewertet wurden (*False*). Dabei enthält *skipped* die Bewertungen für die in diesem Programmlauf angezeigten Funktionsaufrufe, während *unrated* die Bewertungen für in diesem Lauf noch nicht angezeigte, aber möglicherweise in einem vorheigen Lauf bewertete Funktionsaufrufe enthält.

Die Behandlung der Orakelwerte ist in Unterabschnitt 3.1 auf Seite 11 erklärt, die der anderen Komponenten in Unterabschnitt 3.7 auf Seite 19.

**Betriebsart** Ein Ausdruck kann in drei Betriebsarten ausgewertet werden, die in dem Datentyp *StepMode* codiert sind:

*StepInteractive* Funktionsaufrufe werden angezeigt und vom Anwender interaktiv bewertet. Dazu werden zunächst alle Unterberechnungen in der Betriebsart *StepBackground* ausgeführt, und das Gesamtergebnis wird angezeigt. Optional werden dann die Unterberechnungen in der Betriebsart *StepInteractive* wiederholt.

*StepBackground* Der Berechnungsschritt wird im Hintergrund ohne Interaktion mit dem Anwender ausgeführt. Für alle Unterberechnungen bleibt dabei die Information, ob sie als korrekt bewertet wurde, erhalten.

*StepCorrect* Die Funktion wird ohne Interaktion ausgeführt, nach der Korrektheit ihrer Unterfunktionen wird nicht mehr gefragt.

## 3.5 Berechnungsschritte: Der Typ *Step a*

Der Typ *DebugReport* wird verwendet, um eine fehlerhafte Codestelle zu beschreiben. Hierzu enthält er einen Konstruktorterm *lhs* und einen Konstruktorterm *rhs*. Die Komponente *lhs* gibt den Funktionsaufruf an, in dessen *right hand side* ein Bug gefunden wurde. Dabei wird im Wurzelknoten

des Konstruktorterms das Namensfeld für den Funktionsnamen und die Liste der Unterausdrücke für die Argumente des Funktionsaufrufs verwendet. Die Komponente *rhs* gibt den als falsch beurteilten Funktionswert dieses Aufrufs an.

```
data BugReport = BugReport{
  lhs :: ConstructorTerm,
  rhs :: ConstructorTerm}
```

Der Typ *DebugMonad a* stellt die vom Debugger überwachten Berechnungsschritte dar.

```
type DebugMonad a
  = StateT DebuggerState (ErrorT (Maybe BugReport) IO) a
```

Das Ergebnis eines Berechnungsschritts ist in die IO-Monade gehoben, da der interaktive Debugger während der Berechnung Ein- und Ausgabeaktionen durchführen muss.

Diese Monade ist durch den Monaden-Transformer *ErrorT* erweitert, so dass nicht nur das Ergebnis zurückgegeben werden kann, sondern auch die Berechnung mit der Meldung einer fehlerhaften Codestelle (*Just bug*) oder des Programmabbruchs (*Nothing*) abgebrochen werden kann. Sobald eine fehlerhafte Berechnung gefunden wurde, wird die Ausführung abgebrochen und die fehlerhafte Programmstelle als Ergebnis zurückgeliefert.

In einem weiteren Erweiterungsschritt ist dieser Typ durch den Monadentransformer *StateT* erweitert, so dass während der Berechnung der Zustands des Debuggers ausgelesen und geschrieben werden kann.

```
type Step a = StepMode → DebugMonad a
```

*Step a* ist der Typ eines Berechnungsschritts, wobei *a* der Typ des Ergebnisses ist. Er besteht aus einem Berechnungsschritt, der mit der Betriebsart *StepMode*, in der er ausgeführt werden soll, parametrisiert ist.

## 3.6 Kombination von Berechnungsschritten

Die Funktion *evalIfNeeded* konsumiert einen Orakeleintrag. Abhängig von diesem wertet sie entweder ihr Argument *a* aus und gibt das Ergebnis der Auswertung zurück, oder sie gibt den Platzhalter *underscore* zurück.

```
evalIfNeeded :: StrictCurry a ⇒ Step a → Step a
evalIfNeeded a mode = do
  state ← get
  let (orc, needed) = popBoolStack (oracle state)
  put (state{ oracle = orc })
  (if needed then
      a mode
    else
      return underscore)
```

Das folgende Funktionenpaar dient dazu, Programmschritte zu kombinieren:

```
(>>>=) :: StrictCurry a ⇒ (Step a) → (a → Step b) → Step b
a >>>= b = λmode → do
  a' ← evalIfNeeded a mode
  b a' mode
return' :: StrictCurry a ⇒ a → Step a
return' x = λ_ → return x
```

Die Funktion (>>>=) ruft zunächst die Funktion *evalIfNeeded* mit dem ersten Argument $a$ auf, die das Argument entweder auswertet oder durch den Platzhalter *underscore* ersetzt. Dann wird das zweite Argument $b$ auf das Ergebnis dieses Aufrufs angewendet. Da bei dieser Anwendung auf die Funktion ($\gg=$) des Typs *DebugMonad* zurückgegriffen wird, wird dabei sowohl die Verwaltung des Debuggerzustands durch *StateT* als auch die Ausnahmebehandlung durch *ErrorT* im Hintergrund durchgeführt.

Die Funktion *return'* wandelt ihr Argument in einen Berechnungsschritt um, der dieses Argument als Ergebnis liefert.

Sieht man davon ab, dass (>>>=) mittels der Funktion *evalIfNeeded* im Hintergrund Orakelwerte konsumiert und möglicherweise die Auswertung des ersten Arguments unterdrückt, dann bildet der Typ *Step a* mit den Funktionen (>>>=) und *return'* ebenso wie *DebugMonad* mit den Funktionen ($\gg=$) und *return* eine Monade.

**Beispiel** *inf* :: *Step Nat*

Die Funktion *inf* ist nach der Transformation durch `stricths` kein Datum vom Typ *Nat* mehr, sondern ein Berechnungsschritt, dessen Ausführung ein Ergebnis vom Typ *Nat* liefert.[5]

$$min :: Nat \to Nat \to Step\ Nat$$

Die Funktion *min* ist nach wie vor eine zweistellige Funktion, die als Argumente Daten vom Typ *Nat* erwartet. Das Ergebnis ist nun aber ein Berechnungsschritt, dessen Ausführung erst das Ergebnis liefert.

$$main :: IO\ ()$$

Die Funktion *main* wird von der Transformation `stricths` gesondert behandelt. Sie ist nun eine IO-Aktion vom Typ *IO* (). Diese Aktion enthält die gesamte Debugger-Sitzung vom Laden der Orakeldatei bis zur interaktiven Fehlersuche (vgl. Unterabschnitt 3.8 auf Seite 21).

## 3.7 Die Funktion *traceFunCall*

Die Funktion *traceFunCall* bildet die Schnittstelle zwischen dem interaktiven Debugger und dem instrumentierten Programm. In diesem sind alle Funktionsdeklarationen durch Aufrufe von *traceFunCall* ergänzt. *traceFunCall* hat die folgende Typsignatur:

$$traceFunCall :: StrictCurry\ a \Rightarrow \\ String \to [ConstructorTerm] \to Step\ a \to Step\ a$$

Das erste Argument ist der Name der zu debuggenden Funktion. Das zweite Argument ist eine Liste von Konstruktortermen, die die Funktionsargumente repräsentieren. Das dritte Argument ist der – in die Monade *Step* gehobene – Berechnungsschritt, der der rechten Seite dieses Funktionsaufrufs entspricht. Damit das Ergebnis vom Debugger dargestellt werden kann, wird dabei gefordert, dass der Ergebniswert eine Instanz der Typklasse *StrictCurry* ist. Das vierte Argument ist die Betriebsart des Debuggers.

Die Funktionsweise von *traceFunCall* hängt von dieser Betriebsart ab:

[5] Auf die Konsequenzen dieser Entscheidung wird in Abschnitt 4.1 auf Seite 23 eingegangen.

**Betriebsart** *StepCorrect* In dieser Betriebsart wird nur der Funktionswert berechnet. Dieser und alle Unterfunktionsaufrufe werden als korrekt angesehen. Daher bleiben die Komponenten *skipped* und *unrated* des Debuggerzustands bei der Ausführung unverändert; lediglich Orakelwerte werden konsumiert.

**Betriebsart** *StepBackground* In dieser Betriebsart wird ebenfalls nur der Funktionswert berechnet. Da aber die vorher getätigten Bewertungen der Unterfunktionen bei dieser Auswertung erhalten bleiben sollen, werden nicht nur Orakelwerte konsumiert, sondern er wird auch bei jedem Aufruf von *traceFunCall* ein Wert von *unrated* entfernt und auf *skipped* gelegt. So können bei einem Neustart des Debuggers die im vorigen Lauf getätigten Beurteilungenerhalten bleiben, indem man die nach dem Ende des ersten Laufs auf *skipped* liegenden befindlichen Beurteilungen wieder auf *unrated* legt.

War der beim Aufruf von *traceFunCall* von *unrated* entfernte Wert *False*, dann war der Funktionsaufruf in einem vorhergehenden Lauf als korrekt bewertet worden und die Funktion *expr* wird in der Betriebsart *StepCorrect* ausgeführt.

**Betriebsart** *StepInteractive* In dieser Betriebsart findet die gesamte Kommunikation zwischen dem Debugger und dem Anwender statt. Zunächst wird das Ergebnis *result* in der Betriebsart *StepBackground* berechnet und – vorläufig – in *skipped* als unbewertet eingetragen, dann wird es dem Anwender zur Bewertung vorgelegt.

- Hat der Anwender den Aufruf als korrekt bewertet, dann gelten auch alle Unteraufrufe dieses Aufrufs als korrekt. Daher werden die Bewertungen aller Unteraufrufe von *skipped* und *unrated* sowie die vorläufige Bewertung des aktuellen Ausdrucks von *skipped* entfernt und durch einen Eintrag mit dem Wert *False* ersetzt, der die Korrektheit der gesamten Unterberechnung anzeigt.

- Hat der Anwender den Aufruf als falsch bewertet, dann wird der Debugger neu gestartet – dieses Mal jedoch nur mit dem Unterausdruck, der das falsche Ergebnis geliefert hat. Wird in diesem ein Fehler gefunden, so wird er als Ergebnis des Debugger-Laufs zurückgegeben. Wird darin kein Fehler gefunden, dann enthält der Ausdruck *expr* selbst den Fehler; dieser wird als Ergebnis zurückgegeben.

- Überspringt der Anwender die Bewertung (*skip*), dann wird das Ergebnis *result* zurückgegeben, die Bewertungen in *skipped* – die während der vorherigen Auswertung im Modus *StepBackground* von *unrated* entfernt und zu *skipped* hinzugefügt worden sind – bleiben dabei erhalten.
- Wechselt der Anwender zur Bewertung der Unterausdrücke (*step into*), dann wird dieser Unterausdruck in der Betriebsart *StepInteractive* ausgewertet. Die Bewertung des aktuellen Ausdrucks, der in *skipped* als unbewertet eingetragen ist, bleibt dabei erhalten.

Ist in der Komponente *displayMode* des Debuggerzustands das Flag *optionalResult* gesetzt, dann wird beim Aufruf zunächst kein Ergebnis berechnet, sondern nur der Funktionsaufruf angezeigt. Erst auf Anforderung wird das Ergebnis berechnet. Vorher ist es nicht möglich, den Funktionsaufruf zu bewerten, sondern nur, das Ergebnis zu überspringen oder zur Bewertung der Unterausdrücke zu wechseln.

**Beispiel** Die instrumentierte Funktion *min* macht deutlich, wie die Funktionsparameter in Konstruktorterme umgewandelt und als solche der Funktion *traceFunCall* übergeben werden:

```
min x1 x2
  = traceFunCall "min" [showCons x1, showCons x2]
    (...)
```

## 3.8 Aufruf einer Debugging-Sitzung

Die beiden Funktionen *traceWithStepfile* und *traceProgram* dienen als Einstiegspunkte für eine Debugging-Sitzung.

**traceProgram** Die Funktion *traceProgram* hat die folgende Signatur:

$$traceProgram :: StrictCurry\ a \Rightarrow Step\ a \rightarrow Oracle \rightarrow IO\ ()$$

Sie erhält als Argument einen Ausdruck vom Typ *Step a*, der das zu debuggende Programm darstellt, sowie eine Orakelliste, die für die orakelgesteuerte strikte Auswertung des Programms verwendet werden soll. Das Programm

wird von *traceProgram* interaktiv ausgeführt und dabei so lange wiederholt, bis alle Funktionsaufrufe bewertet sind oder ein fehlerhafter Aufruf gefunden wurde. Der folgende Aufruf startet eine Debugger-Sitzung mit einem Ausdruck, der dem Ausdruck *main* des Beispielprogramms gleichwertig ist, aber ein explizit übergebenes Orakel verwendet:

```
traceProgram ((inf >>= \ x1 -> min (S (S (S Z))) x1)) [4,12]
```

**traceWithStepfile** Die Funktion *traceWithStepfile* hat die folgende Signatur:

$$traceWithStepfile :: StrictCurry\ a \Rightarrow String \rightarrow Step\ a \rightarrow IO\ ()$$

Sie lädt eine Orakeldatei und führt damit eine Debugging-Sitzung durch, bei der das Programm, das im zweiten Argument übergeben worden ist, ausgeführt wird. Der Name der Orakeldatei besteht aus dem ersten Argument, das um das Suffix `".steps"` ergänzt worden ist.

**main** Hat eine Deklaration des zu transformierenden Programms den Namen *main*, dann wird sie von der Transformation `stricths` nicht nur zu einem Berechnungsschritt transformiert, sondern sie wird zusätzlich mit einem Aufruf von *traceWithStepfile* versehen, in dem eine Orakeldatei geladen wird. Der Name der Orakeldatei ergibt sich dabei aus dem Namen des transformierten Programmoduls.

Im Beispiel wird *main* zum folgenden Ausdruck transformiert:

```
main :: IO ()
main = traceWithStepfile "Example"
  (traceFunCall "main" []
    (...)
```

Die Auslassung (...) steht dabei für den Funktionsrumpf von *main*, der wie die Funktionsrümpfe gewöhnlicher Funktionen instrumentiert ist.

Da *main* nun den Typ *IO* () hat, kann man das transformierte Programm ohne weitere Änderungen mit einem Haskell-Compiler wie `ghc` compilieren. Ebenso kann man eine Debugging-Sitzung starten, indem man in einem Haskell-Interpreter das Modul `ExampleStrict` lädt und dann den Ausdruck *main* auswerten läßt. Dabei wird dann automatisch die vorher erzeugte

Orakel-Datei `Example.steps` geladen. Eine Debugging-Sitzung, bei der die Funktion *main* von Hand aufgerufen wird, könnte z.B. so ablaufen:

```
Prelude> :l ExampleStrict
(...)
ExampleStrict> main
starting trace
(...)
main ~> S (S (S Z))) c
no bugs found.
```

# 4 Ausblicke

In diesem Abschnitt sollen verschiedene Aspekte der Transformation `stricths` genauer betrachtet und Ausblicke auf mögliche Erweiterungen gegeben werden.

## 4.1 Zirkuläre Datenstrukturen

Im Beispiel aus Abbildung 2 auf Seite 10 enthält das ursprüngliche Programm eine zirkuläre Definition. Sie beschreibt eine unendliche Datenstruktur, die durch eine endliche zirkuläre Struktur dargestellt wird:

$$\begin{aligned} &\mathit{inf} :: \mathit{Nat} \\ &\mathit{inf} = S\ \mathit{inf} \end{aligned}$$

Im instrumentierten Programm ExampleStrict ist diese Deklaration in einen rekursiven Funktionsaufruf übersetzt.

$$\begin{aligned} &\mathit{inf} :: \mathit{Step}\ \mathit{Nat} \\ &\mathit{inf} = \mathit{traceFunCall}\ \texttt{"inf"}\ [\,] \\ &\quad (\mathit{inf} >>>= \lambda x1 \rightarrow \mathit{return}'\ (S\ x1)) \end{aligned}$$

Bei der orakelgesteuerten strikten Auswertung wird also nur eine endliche Annäherung an diese unendliche Datenstruktur erzeugt, nämlich die Liste $S\ (S\ (S\ \_))$. Formal gesehen, wird also anstelle des kleinsten Fixpunktes der Nachfolgeroperation $\mu_S = \infty$ eine endliche Annäherung $S^4(\bot)$ berechnet.

Die in [Braßel et al. 2007] vorgestellte Semantik sieht jedoch vor, dass die zirkuläre Datenstruktur auch im instrumentierten Programm als solche dargestellt wird. Nach der Regel

$$\Gamma : C\,\overline{x_n} \;\Downarrow_\epsilon\; \Gamma : C\,\overline{x_n}$$

werden dort Konstruktoren direkt im Heap gebunden – nullstellige Funktionen wie die obige instrumentierte Version von *inf* sind dabei überhaupt nicht vorgesehen. Dennoch hat es Vorteile, zirkuläre Definitionen *nicht* als zyklische Datenstrukturen darzustellen, sondern die bisher implementierte Variante beizubehalten.

**Vorteile zirkulärer Datenstrukturen** Für die Verwendung zirkulärer Datenstrukturen spricht vor allem die Effizienz. Alle Strukturen werden nur einmal erzeugt; sie werden nur dort erzeugt, wo sie definiert sind, und sie werden beim rekursiven Zugriff auch nicht kopiert. Bei der Erzeugung werden auch keine Orakelschritte gezählt, so dass das Orakel nicht unnötig vergrößert wird.

**Vorteile endlicher Annäherungen** Der Vorteil endlich angenäherter Datenstrukturen ist, dass diese im Debugger direkt ausgegeben werden können. Dabei sieht der Anwender, welche Teile der Datenstruktur für die Berechnung relevant waren. Im Beispiel hat der Debugger nach der Korrektheit der folgenden Auswertung gefragt:

```
(min (S (S (S Z))) (S (S (S (S _))))) ~> (S (S (S Z)))
```

Hätte der Wert inf als zirkuläre Datenstruktur vorgelegen, dann wäre es notwendig gewesen, die Ausgabe generell durch eine maximale Länge oder eine maximale Schachtelungstiefe zu begrenzen: der Anwender würde in der Regel entweder zu viele oder zu wenige Daten präsentiert bekommen.

**Ein Kompromißvorschlag** Eine mögliche Lösung ist, rekursive Deklarationen durch zirkuläre Datenstrukturen darzustellen und eine Funktion *prune* einzuführen, die zu einer vorhandenen zirkulären Datenstruktur eine endliche Annäherung erstellt. Damit kann man an allen Stellen, an denen man die tatsächlich verwendeten Teile eines Ausdrucks $x$ isolieren will, diese durch den Ausdruck $prune\,x$ ersetzen. In unserem Beispiel würde man die Deklaration von *min* wie folgt abändern:

$$
\begin{aligned}
&\mathit{min}\ \mathit{x1}\ \mathit{x2}\\
&\quad = \mathit{prune}\ \mathit{x1} >\!\!>\!\!= \lambda \mathit{x1pruned} \rightarrow\\
&\qquad \mathit{prune}\ \mathit{x2} >\!\!>\!\!= \lambda \mathit{x2pruned} \rightarrow\\
&\qquad\quad \mathit{traceFunCall}\ \texttt{"min"}\ [\mathit{showCons}\ \mathit{x1Pruned}, \mathit{showCons}\ \mathit{x2Pruned}]\\
&\qquad\qquad (...)
\end{aligned}
$$

Auf diese Weise könnte man für jeden Aufruf von *min* genau bestimmen, auf welche Teile der Argumente zugegriffen wird. Gleichzeitig wären die unbeobachteten Daten effizient durch zirkuläre Strukturen dargestellt.

Legt man die operationale Semantik aus [Braßel et al. 2007] zugrunde, in der die mittels `let`-Ausdrücken gebundenen Werte nur bei Bedarf erzeugt und andernfalls durch einen Platzhalter-Wert *underscore* ersetzt werden, dann kann die Funktion *prune* durch die folgenden Regeln erklärt werden:

$$
\begin{aligned}
\mathit{prune}\ (\lambda x.e) &\equiv \lambda x.e\\
\mathit{prune}\ (C\ \overline{v_n}) &\equiv \overline{\text{let } x_n = \mathit{prune}\ v_n \text{ in}}\ C\ x_1 \ldots x_n
\end{aligned}
$$

Die Funktion *prune* läßt sich mithilfe von Let- und Case- Ausdrücken implementieren, daher kann er durch eine einfache Quelltexttransformation in das auszuwertende Programm eingefügt werden. Hierzu müßte die Typklasse *StrictCurry* um eine Funktion *prune* ergänzt werden. Analog zur automatisch erzeugten Funktion *showConsNat* würde dann z. B. für den Datentyp *Nat* des Beispiels die folgende Funktion erzeugt werden:

$$
\begin{aligned}
&\mathit{pruneNat} :: \mathit{Nat} \rightarrow \mathit{Step}\ \mathit{Nat}\\
&\mathit{pruneNat}\ x = \textbf{case}\ x\ \textbf{of}\\
&\quad Z \rightarrow \mathit{return}'\ Z\\
&\quad S\ x' \rightarrow \mathit{pruneNat}\ x' >\!\!>\!\!= \mathit{return}' \circ S
\end{aligned}
$$

## 4.2 Imperative Programme

Die Berechnungsschritte der orakelgesteuerten strikten Auswertung enthalten Nutzerinteraktionen. Um das zu ermöglichen, findet die Auswertung eines Berechnungsschrittes *Step a* innerhalb der IO-Monade statt. Damit sollte es ohne größere Anpassungen möglich sein, auch Programme mit Seiteneffekten mittels der Laufzeitbibliothek `StrictCurry.hs` zu debuggen. Ebenso ist die Transformation `transform` bereits in der Lage, imperative Programme

```
data Nat = Z | S Nat
inf = S inf
min Z (S _) = return Z
min (S _) Z = return Z
min (S x) (S y) = do
  m' <- min x y
  return (S m')
main = do
  r <- min (S (S (S Z))) inf
  putStrLn ("result: " ++ show r)
```

Abbildung 3: Imperative Variante von `ExMinimum.curry`

korrekt zu transforieren. Einzig die Transformation `stricths` muss hierzu erweitert werden.

Abbildung 3 zeigt eine imperative Variante des Minimalbeispiels aus Abschnitt 3 auf Seite 10. `transform` erzeugt für dieses Programm eine korrekte Orakeldatei. Jedoch schlägt die Transformation `stricths` fehl: das erzeugte Programm ist nicht typkorrekt.

## 4.3 Zur Effizienz

Aus der Funktion *min* aus Abbildung 2 auf Seite 10 erzeugt die Transformation `stricths` die folgende Funktionsdeklaration:[6]

```
min :: Nat -> Nat -> Step Nat
min x1 x2 = (case x1 of
  Z -> return' Z
  S x3 -> (case x2 of
    Z -> return' Z
    S x4 -> min x3 x4 >>>= λx5 -> return' (S x5))
    >>>= return')
  >>>= return'
```

Diese Funktion lässt sich durch Pattern Matching folgendermaßen darstellen:

[6] hier in leicht geglätteter Darstellung und ohne den Aufruf von *traceFunCall*

$$
\begin{aligned}
&\mathit{min}\ Z\ \_ \qquad = \mathit{return}'\ Z >\!\!>\!\!= \mathit{return}' \\
&\mathit{min}\ (S\ x3)\ Z = (\mathit{return}'\ Z >\!\!>\!\!= \mathit{return}') >\!\!>\!\!= \mathit{return}' \\
&\mathit{min}\ (S\ x3)\ (S\ x4) = ((\mathit{min}\ x3\ x4 >\!\!>\!\!= \lambda x5 \rightarrow \mathit{return}'\ (S\ x5)) \\
&\quad >\!\!>\!\!= \mathit{return}') >\!\!>\!\!= \mathit{return}'
\end{aligned}
$$

In der ersten Regel wird zunächst ein Datum $Z$ erzeugt. Dann wird durch die Funktion $(>\!\!>\!\!>\!\!=)$ ein Orakelwert konsumiert und getestet, ob der Wert $Z$ benötigt wird, um das Programm auszuwerten. An dieser Stelle ist jedoch schon klar, dass dieser Wert benötigt wird. Denn würde er nicht gebraucht, dann wäre die Funktion *min* gar nicht aufgerufen worden; schon an der Stelle des Aufrufs wäre der Wert *underscore* anstelle des Funktionswertes geliefert worden.

Ähnlich sieht es in den beiden anderen Regeln aus. Hier werden sogar zwei Aufrufe von $>\!\!>\!\!>\!\!=$ direkt hintereinander ausgeführt. Dabei ist es unmöglich, dass nur das Ergebnis von einem der beiden Aufrufe benötigt wird: Wird das Ergebnis des äußeren Aufrufs benötigt, dann muss die Auswertung des inneren Aufrufs dieses Ergebnis liefern. Wird das Ergebnis des inneren Aufrufs benötigt, dann auch das des äußeren, denn kein anderer als dieser kann das Ergebnis des inneren angefordert haben.

Daher hätte es gereicht, wenn die Funktion *min* in die folgende Funktion transformiert worden wäre – und natürlich bei der Erzeugung des Orakels entsprechend weniger Einträge generiert worden wären:

$$
\begin{aligned}
&\mathit{min}\ Z\ \_ \qquad = \mathit{return}'\ Z \\
&\mathit{min}\ (S\ x3)\ Z = \mathit{return}'\ Z \\
&\mathit{min}\ (S\ x3)\ (S\ x4) = \mathit{min}\ x3\ x4 >\!\!>\!\!>\!\!= \lambda x5 \rightarrow \mathit{return}'\ (S\ x5)
\end{aligned}
$$

Dadurch wäre der zusätzliche Aufwand, den die Erzeugung und Verwendung des Orakels mit sich bringt, auf etwa ein Drittel reduziert worden.

# 5 Fazit

Es wurde der Prototyp eines Debugger-Frontends erstellt, das es allein durch eine Programmtransformation ermöglicht, nicht-strikte Programme in strikter Auswertungsreihenfolge zu debuggen. Dadurch wurde gezeigt, dass das in [Braßel et al. 2007] vorgestellte Prinzip der orakelgesteuerten strikten Auswertung eine geeignete Grundlage bietet, um mit einfachen Mitteln die besonderen Anforderungen, die das Debuggen nicht-strikter deklarativer Sprachen stellt, in den Griff zu bekommen.

Bei der Arbeit an diesem Prototypen haben sich Erweiterungsmöglichkeiten, Designalternativen und Möglichkeiten zur Steigerung der Effizienz gezeigt, die es nahelegen, die Details der Programmtransformation eingehender zu untersuchen.

# Literatur

[Braßel et al. 2007] B. Braßel and S. Fischer and M. Hanus and F. Huch and G. Vidal. Lazy Call-By-Value Evaluation. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming (ICFP'07)*, pages 265 – 276.