16 Sprachunterstützung für Datendefinitionen und Signaturen

In einem gewissen Sinne ist es egal, welche Programmiersprache man verwendet: Jede Berechnung und jeder Algorithmus lassen sich in jeder Programmiersprache ausdrücken. Man kann jede Programmiersprache und jedes Programmiersprachenkonzept in jeder anderen Programmiersprache simulieren.

Dies gilt zumindest für alle sogenannten "Turing-vollständigen" Sprachen. Fast alle gängigen Programmiersprachen sind Turing-vollständig.

Die Entwurfstechniken, die wir Ihnen beibringen, lassen sich in allen Programmiersprachen verwenden. Allerdings unterscheiden sich Programmiersprachen darin, wie gut die Entwurfstechniken durch die Sprache unterstützt werden. Wenn Sie beispielsweise in Assembler-Programmiersprachen programmieren, gibt es nichts, was unserem `define-struct` entspricht; ein Assembler Programmier muss sich daher selber überlegen, wie er mehrere Werte zusammenfasst und in einem linearen Speicher anordnet.

Auf der anderen Seite gibt es auch Programmiersprachen, in denen unsere Entwurfstechniken besser unterstützt werden als in BSL.

In diesem Kapitel wollen wir darüber reden, wie Sprachen eines unserer Kernkonzepte unterstützen können, nämlich das der Datendefinitionen und Signaturen. Datendefinitionen dienen zur Strukturierung und Klassifikation der Daten in unserem Programm. Signaturen dienen der Beschreibung der Erwartungen einer Funktion an ihre Argumente und der Beschreibung der Garantien bezüglich des Ergebnisses.

Um Sprachen und Sprachfeatures bezüglich ihrer Unterstützung für Datendefinitionen und Signaturen zu bewerten, benötigen wir Kriterien, die qualitativ diese Unterstützung messen. Wir wollen die folgenden Kriterien betrachten:

- Ist sichergestellt, dass alle Werte und Funktionen in einer Weise verwendet werden, die zur Information/Berechnung, die dieser Wert/Funktion repräsentiert, passt?
- Zu welchem Zeitpunkt werden Fehler gefunden? Im Allgemeinen möchte man, dass Fehler möglichst früh auftreten; am besten schon bevor das Programm startet, aber zumindest zu dem Zeitpunkt, an dem der Programmteil, der für den Fehler verantwortlich ist, ausgeführt wird. Am schlechtesten ist, wenn das Programm selber niemals einen Fehler meldet und stattdessen möglicherweise unbemerkt falsche Ergebnisse produziert.
- Wie modular sind die Fehlermeldungen? Wenn ein Fehler auftritt, so möchte man wissen, welcher Programmteil daran "schuld" ist. Dieses sogenannte "Blame Assignment" ist äußerst wichtig, um Fehler effektiv lokalisieren und beheben zu können. Bei Fehlern sollte es stets einen klar benennbaren Programmteil geben, der der Verursacher dieses Fehlers war.
- Wie ausdrucksstark ist die Signatursprache? Kann man die Bedingungen an Eingaben und Ausgaben präzise darin ausdrücken?
- Gibt es sinnvolle Programme, die nicht mehr ausgeführt werden können?

- Wird die Aufrechterhaltung der Konsistenz zwischen Datendefinitionen/Signaturen und dem Programmverhalten unterstützt?

Im folgenden wollen wir die wichtigsten Klassen von Programmiersprachen beschreiben, die sich in wichtigen Punkten bezüglich der oben angeführten Kriterien unterscheiden.

16.1 Ungetypte Sprachen

Ungetypte Sprachen zeichnen sich dadurch aus, dass jede Operation auf alle Arten von Daten angewendet werden kann, unabhängig davon ob es einen Sinn ergibt oder nicht. Beispielsweise ist es nicht sinnvoll, einen String und eine Zahl miteinander zu addieren.

Assembler-Sprachen sind typischerweise ungetypt. Alle Arten von Daten werden als (32 oder 64 bit) Zahlen repräsentiert. Auch Strings, boolsche Werte, und alle anderen Daten werden durch solche Zahlenwerte repräsentiert. Addiert man nun zwei Werte, so werden die Zahlenwerte addiert, egal ob das aus Sicht dessen, was diese Werte repräsentieren, einen Sinn ergibt.

Bezüglich des ersten Punkts aus der Liste oben bieten ungetypte Sprachen daher keinerlei Unterstützung; es liegt vollständig in der Verantwortung des Programmierers, diese Eigenschaft sicherzustellen.

Fehler, also Verstöße gegen diese Eigenschaft, werden in Assembler-Programmen dementsprechend sehr spät gefunden, denn das Programm läuft ja einfach immer weiter, auch wenn die Daten, die gerade berechnet wurden, völlig unsinnig sind.

Da es keine von der Sprache unterstützten Signaturen oder Datendefinitionen gibt, gibt es auch keine Einschränkungen bezüglich der Ausdrucksstärke der Signaturen/Datendefinitionen und es gibt keine Einschränkungen der Art, dass bestimmte Programme nicht ausgeführt werden können. Allerdings gibt es auch keinerlei Unterstützung für die Aufrechterhaltung der Konsistenz; dies liegt allein in der Verantwortlichkeit des Programmierers.

16.2 Dynamisch getypte Sprachen

In dynamisch getypten Sprachen wird jedem Wert ein Typ zugeordnet und das Laufzeitsystem repräsentiert Werte so, dass der Typ eines Wertes während der Ausführung jederzeit abgefragt werden kann. Ein Typ ist daher eine Art Markierung für Werte, die darüber Auskunft gibt, was für eine Art von Wert es ist. Typischerweise gibt es (je nach Sprache unterschiedliche) fest eingebaute ("primitive") Typen sowie vom Benutzer zu definierende Typen. Die Beginning Student Language, in der wir bisher programmiert haben, ist eine dynamisch getypte Sprache. Fest eingebaute Typen sind beispielsweise Boolean (`boolean?`), Number (`number?`), String (`string?`) und Symbole (`symbol?`). Neue Typen können mittels `define-struct` definiert werden.

Die dynamischen Typen werden verwendet, um sicherzustellen, dass nur solche primitiven Operationen auf die Werte angewendet werden, die auch für diese Werte definiert sind. Wenn wir beispielsweise `(+ x y)` auswerten, so prüft das Laufzeitsystem, dass `x` und `y` auch tatsächlich Zahlen sind. Wenn `x` hingegen beispielsweise ein

boolscher Wert ist, so wird dieser, anders als bei ungetypten Sprachen, nicht einfach irgendwie als Zahl interpretiert.

Allerdings gilt diese Eigenschaft nicht für vom Programmierer selber definierte Funktionen. Es ist äußerst sinnvoll, jede Funktionsdefinition mit einer Signatur zu versehen, so wie wir es ja auch gemacht haben, doch es wird nicht geprüft, ob die Signatur auch von der Funktion und den Aufrufern der Funktion eingehalten wird.

Da die dynamischen Typen jedoch nicht alle Informationen umfassen, die wir in Datendefinitionen festhalten, kann es dennoch zu Fehlbenutzungen von Werten kommen. Beispielsweise macht es keinen Sinn, eine Temperatur und eine Länge zu addieren. Falls beide jedoch durch den Typ Number repräsentiert werden, kann das Laufzeitsystem diesen Fehler nicht feststellen.

Schauen wir uns mal an einigen Beispielen an, wie und wann in dynamische Typsystemen Typfehler auftauchen. Betrachten Sie folgende Funktion:

```
; Number (list-of Number) -> (list-of Number)
; returns the remainder of xs after first occurence of x, or
empty otherwise
(define (rest-after x xs)
  (if (empty? xs)
      empty
      (if (= x (first xs))
          (rest xs)
          (rest-after x (rest xs)))))
```

Die Funktion gibt den Rest der Liste nach dem ersten Vorkommen des Elements `x` zurück (und `empty` falls das Element nicht vorkommt). Hier zwei Beispiele dazu:

```
> (rest-after 5 (list 1 2 3 4))
'()
> (rest-after 2 (list 1 2 3 4))
'(3 4)
```

Was passiert jedoch, wenn wir die Signatur verletzten?

```
> (rest-after 2 (list "eins" "zwei" "drei"))
=: expects a number as 2nd argument, given "eins"
```

Wir sehen, dass wir in diesem Fall einen Laufzeitfehler erhalten. Allerdings tritt nicht bei jeder Verletzung der Signatur (sofort) ein Laufzeitfehler auf:

```
> (rest-after 2 (list 1 2 "drei" "vier"))
'("drei" "vier")
```

In diesem Beispiel wird eine Liste übergeben, die nicht nur Zahlen enthält, aber da die hinteren Elemente der Liste nicht verwendet werden gibt es auch keinen Laufzeitfehler.

Betrachten wir jetzt einmal den Fall, dass nicht der Aufrufer der Funktion sondern die Funktion selber die Signatur verletzt. In diesem Beispiel gibt die Funktion einen String statt einer leeren Liste zurück, falls `x` in der Liste nicht vorkommt.

```
; Number (list-of Number) -> (list-of Number)
; returns the remainder of xs after first occurence of x, or
empty otherwise
(define (rest-after x xs)
  (if (empty? xs)
      "not a list"
      (if (= x (first xs))
          (rest xs)
          (rest-after x (rest xs)))))
```

Diese Beispiele illustrieren, dass es keinen Fehler ergibt, wenn wir die Funktion mit einem `x` aufrufen, welches in der Liste enthalten ist.

```
> (rest-after 2 (list 1 2 3 4))
'(3 4)
```

Selbst wenn wir ein `x` auswählen, welches nicht in der Liste enthalten ist, ergibt die Ausführung keinen Laufzeitfehler.

```
> (rest-after 5 (list 1 2 3 4))
"not a list"
```

Erst wenn wir das Ergebnis verwenden und damit rechnen kommt es zu einem Laufzeitfehler.

```
> (cons 6 (rest-after 5 (list 1 2 3 4)))
cons: second argument must be a list, but received 6 and "not a list"
```

Allerdings ist es im Allgemeinen sehr schwer, herauszufinden, wer denn "Schuld" an diesem Fehler ist, denn die Stelle an der der Fehler auftritt ist möglicherweise weit von der Stelle entfernt, die den Fehler verursacht. Wenn Sie sich die Fehlermeldung anschauen, sehen Sie auch nichts, das darauf hindeutet, dass die Ursache des Fehlers in der Implementierung der `rest-after` Funktion zu finden ist. Daher sind Fehlermeldungen in dynamisch getypten Sprachen nicht sehr modular.

16.3 Dynamisch überprüfte Signaturen und Contracts

Um Fehler früher zu finden und Fehlermeldungen modularer zu machen, können Signaturen und Datendefinitionen auch als Programme definiert werden. Diese Programme können dann verwendet werden, um Signaturen zu überprüfen während das Programm läuft.

Eine Datendefinition wie:

```
; a list-of-numbers is either:
;  - empty
;  - (cons Number list-of numbers)
```

kann beispielsweise durch folgendes Programm repräsentiert werden:

```
; [X] (list-of X) -> Boolean
; checks whether xs contains only numbers
(define (list-of-numbers? xs)
  (if (empty? xs)
      true
      (and (number? (first xs))
           (list-of-numbers? (rest xs)))))
```

Diese "ausführbaren" Datendefinitionen können dann, zusammen mit den vordefinierten Prädikaten wie `number?` verwendet werden, um eine dynamisch geprüfte Variante der `rest-after` Funktion zu definieren:

```
; Number (list-of Number) -> (list-of Number)
; dynamically checked version of rest-after
(define (rest-after/checked x xs)
  (if (number? x)
      (if (and (list? xs)
               (list-of-numbers? xs))
          (if (list-of-numbers? (rest-after x xs))
              (rest-after x xs)
              (error "function must return list-of-numbers"))
          (error "second arg must be list-of-numbers"))
      (error "first arg must be a number")))
```

Diese Funktion verhält sich genau wie `rest-after` sofern sich die Funktion und ihre Aufrufer an die Signatur halten:

```
> (rest-after/checked 2 (list 1 2 3 4))
'(3 4)
```

Im Fehlerfall gibt es jedoch viel früher eine Fehlermeldung und diese Fehlermeldung ist modular (sie tritt an der Stelle auf, die auch die Ursache des Fehlers ist).

```
> (rest-after/checked "x" (list 1 2 3 4))
first arg must be a number
```

Allerdings werden nun auch Programme mit einer Fehlermeldung abgebrochen, die, wie wir oben gesehen haben, vorher ohne Fehler durchgelaufen sind:

```
> (rest-after/checked 2 (list 1 2 "drei" 4))
second arg must be list-of-numbers
```

Dennoch ist es sinnvoll, diese Programme mit einem Fehler abzubrechen, denn im Allgemeinen wird früher oder später doch noch ein (dann nicht mehr modularer)

Fehler auftreten. In jedem Fall ist ein Verstoss gegen die Signatur ein Hinweis auf einen Programmierfehler, unabhängig davon ob er tatsächlich letzten Endes zu einem Fehler führen würde.

Wie wir sehen, ist es allerdings aus Programmierersicht relativ mühselig und fehleranfällig, auf diese Art Signaturen und Datendefinitionen zu überprüfen. Deshalb gibt es einige Sprachen, die die dynamische Prüfung von Signaturen und Datendefinitionen direkt und komfortabel unterstützen.

Dies trifft beispielsweise auf die Sprache Racket zu, die auch von der DrRacket Umgebung unterstützt wird und die, bis auf kleine Abweichungen, die Beginning Student Language als Teilsprache unterstützt. In Racket können dynamische Prüfungen von Signaturen – in dem Kontext auch *Contracts* genannt – an der Grenze zwischen Modulen definiert werden. Module sind abgeschlossene Programmeinheiten, die in Racket meistens mit Dateien assoziiert sind, also jede Datei ist ein Modul.

Hier sehen Sie die Definition eines Moduls welches die `rest-after` Funktion von oben implementiert. In der `provide` Klausel des Moduls wird dieser Funktion ein *Contract*, also eine ausführbare Signatur, zugeordnet. Wir speichern das Modul in einer Datei "heinz.rkt", um zu illustrieren, dass vielleicht der Entwickler Heinz dieses Modul programmiert hat. Wie Sie sehen, hat Heinz den gleichen Fehler in die Implementierung eingebaut, den wir schon oben betrachtet haben.

`"heinz.rkt"`

```
#lang racket

(provide
 (contract-out
      [rest-after (-> number? (listof number?) (listof number?))]))
(define (rest-after x xs)
  (if (empty? xs)
      "not a list"
      (if (= x (first xs))
          (rest xs)
          (rest-after x (rest xs)))))
```

Betrachten Sie nun ein Modul vom Entwickler Elke, welche diese Funktion benutzen möchte und daher das Modul von Heinz über eine `require` Klausel "importiert". Über diese Klausel wird deutlich gemacht, dass das Modul von Elke von dem Heinz Modul abhängt und dessen Funktionen verwenden möchte.

`"elke.rkt"`

```
#lang racket

(require "heinz.rkt")

(rest-after "x" (list 1 2 3 4))
```

Elke hat allerdings im Aufruf der Funktion gegen den Contract verstossen und hat als erstes Argument einen String übergeben. Wenn wir versuchen, dieses Programm auszuführen, so erhalten wir folgende Fehlermeldung:

```
rest-after: contract violation
 expected: number?
 given: "x"
 in: the 1st argument of
     (->
      number?
      (listof number?)
      (listof number?))
 contract from: /Users/klaus/heinz.rkt
 blaming: /Users/klaus/elke.rkt
  (assuming the contract is correct)
 at: /Users/klaus/heinz.rkt:3.24
```

Sie sehen, dass nicht nur der Aufruf der Funktion direkt als fehlerhaft erkannt wurde. Die Fehlermeldung sagt auch klar, wer an diesem Fehler Schuld ist, nämlich Elke.

Elke korrigiert also ihren Fehler. Nun kommt jedoch der Fehler, den Heinz in die Funktion eingebaut hat, zum Tragen. Dieser Fehler wird jedoch sofort gefunden und es wird korrekt Heinz die Schuld daran zugewiesen.

`"elke.rkt"`

```
 #lang racket

 (require "heinz.rkt")

 (rest-after 5 (list 1 2 3 4))
```

```
rest-after: broke its contract
  promised: "list?"
  produced: "not a list"
  in: the range of
      (->
       number?
       (listof number?)
       (listof number?))
  contract from: /Users/klaus/heinz.rkt
  blaming: /Users/klaus/heinz.rkt
   (assuming the contract is correct)
  at: /Users/klaus/heinz.rkt:3.24
```

Wie diese Beispiele illustrieren, ist der Hauptvorteil von dynamisch überprüften Signaturen und Contracts, das Fehler früher gefunden werden und die Fehlermeldungen modular sind und es bei Verletzungen einen klar benennbaren "Schuldigen" gibt. Wenngleich Fehler hierdurch früher gefunden werden, so werden die Fehler dennoch erst während der Programmausführung gefunden. Da es im Allgemeinen unendlich viele verschiedene Programmausführungen für ein Programm gibt, kann man sich nie sicher sein, dass nicht doch noch Contract-Verletzungen zur Laufzeit auftreten könnnen.

Ein wichtiger Nachteil von Contracts ist, dass man nur solche Contracts ausdrücken kann, die auch tatsächlich berechnet werden können. Eine Signatur wie

```
; [X] (list-of X) -> (list-of X)
```

erfordert es beispielsweise, dass man zur Überprüfung dieses Contracts ein Prädikat benötigt, welches überprüft, ob ein Listenelement ein X ist. Dieses Prädikat muss gegebenenfalls im Programm mit übergeben und gegebenenfalls über große "Entfernungen" durch das Programm "durchgeschleift" werden.

Außerdem können Contracts offensichtlich nur Restriktionen überprüfen, für die die relevanten Informationen auch als Daten zur Verfügung stehen. Eine Datendefinition wie

```
; A temperature is a number that is larger than -273.15.
; interp. temperature in degrees Celsius
```

läßt sich nicht überprüfen, weil wir eine Zahl nicht ansehen können, ob sie eine Temperatur repräsentiert. Allerdings können wir durch eine Strukturdefinition ein passendes Tag dazu definieren, welches dann auch zur Laufzeit überprüfbar ist:

```
(define-struct temperature (d))
; A temperature is: (make-temperature Number) where the number
is larger than -273.15
; interp. a temperature in degrees celsius
```

Ein pragmatischer Nachteil von dynamischen Überprüfungen ist, dass diese die Laufzeit eines Programms stark negativ beeinflussen können. Deshalb gibt es in einigen Sprachen die Möglichkeit, die dynamische Überprüfung abzustellen.

16.4 Statisch getypte Sprachen

Die letzte Variante, um Signaturen und Datendefinitionen durch die Sprache zu unterstützen, ist die Idee eines statischen Typsystems. In einem statischen Typsystem wird jedem Programmteil *vor der Ausführung* ein Typ zugeordnet, und zwar so, dass der Typ eines zusammengesetzten Programmteils nur von den Typen seiner Komponenten abhängt (sogenannte *Kompositionalität*).

Beispielsweise kann dem Ausdruck `(+ e-1 e-2)` der Typ `Number` zugeordnet werden unter der Voraussetzung, dass `e-1` und `e-2` ebenfalls diesen Typ haben.

Statische Typsysteme zeichnen sich durch zwei wichtige Eigenschaften aus: 1) Falls ein Programm den Typchecker durchläuft ("wohlgetypt" ist), so wird in allen (i.A. unendlich vielen) möglichen Programmausführungen kein Typfehler auftreten. In dem Sinne sind statische Typen viel mächtiger als Tests, weil diese immer nur eine kleine Zahl unterschiedlicher Programmausführungen überprüfen können. 2) Es gibt stets Programme, die vom Typchecker abgelehnt werden, obwohl sie eigentlich ausgeführt werden könnten, ohne das ein Typfehler auftritt. Diese Eigenschaft ist eine direkte Konsequenz des sogenannten "Theorem von Rice", welches aussagt, dass nichttriviale Eigenschaften des Verhaltens von Programmen nicht entscheidbar sind.

DrRacket unterstützt eine getypte Variante von Racket, Typed Racket. Hier ist unser Beispiel von oben in Typed Racket:

```
(: rest-after (-> Integer (Listof Integer) (Listof Integer)))
(define (rest-after x xs)
  (if (empty? xs)
      empty
      (if (= x (first xs))
          (rest xs)
          (rest-after x (rest xs)))))
```

Wie wir sehen, gibt es in Typed Racket eine formale Syntax für die Signatur von Funktionen. Das Typsystem von Typed Racket ist so gestaltet, dass die Konsistenz der Funktion zur angegebenen Signatur überprüft werden kann, ohne das Programm bzw. einen Test auszuführen. Es kann also einmal, "once and for all", überprüft werden, dass `rest-after` die angegebene Signatur einhalten wird, und zwar für alle Parameter die den angegebenen Typen genügen.

Diese Funktion kann nun wie in der Beginning Student Language aufgerufen werden:

```
> (rest-after 2 (list 1 2 3 4))
- : (Listof Integer)
'(3 4)
```

Allerdings gibt es einen wichtigen Unterschied: Der Funktionsaufruf wird ebenfalls vor dem Auruf auf Konsistenz mit der Funktionssignatur überprüft:

```
> (rest-after "x" (list 1 2 3 4))
eval:5:0: Type Checker: type mismatch
  expected: Integer
  given: String
  in: 4


> (rest-after 2 (list 1 2 "drei" 4))
eval:6:0: Type Checker: type mismatch
  expected: (Listof Integer)
  given: (List Integer Integer String Integer)
  in: 4
```

Dass diese Überprüfung schon vor dem Aufruf stattfindet, erkennt man daran, dass die Typprüfung eines Aufrufs auch dann gelingt, wenn der tatsächliche Aufruf einen Laufzeitfehler generieren würde.

```
> (:print-type (rest-after (/ 1 0) (list 1 2 3 4)))
(Listof Integer)
```

Auch die Prüfung der Funktion selber findet statt ohne die Funktion auszuführen. Ein Verstoss gegen die angegebene Signatur wird sofort angezeigt.

```
(: rest-after (-> Integer (Listof Integer) (Listof Integer)))

(define (rest-after x xs)
  (if (empty? xs)
      "not a list"
      (if (= x (first xs))
          (rest xs)
          (rest-after x (rest xs)))))
```
eval:9:0: Type Checker: type mismatch
expected: (Listof Integer)
given: String
in: xs

Der grosse Vorteil statischer Typprüfung ist, dass diese schon vor der Programmausführung (beispielsweise beim Entwickler und nicht beim Kunden) gefunden werden und ein wohlgetyptes Programm niemals Typfehler generieren wird. In der Theorie formalisiert man diese Eigenschaft häufig so, dass die Reduktionssemantik für die getypte Sprache stets die Wohlgetyptheit erhält, also wenn ein Programm vor der Reduktion wohlgetypt hat ist es das auch nach der Reduktion (sogenanntes "Preservation" oder "Subject Reduction" Theorem) und wohlgetypte Programme, die keine Werte sind, können stets reduziert werden (sogenanntes "Progress" Theorem).

Der größte Nachteil statischer Typprüfung ist, dass es stets Programme gibt, die vom Typchecker abgelehnt werden, obwohl ihre Ausführung keinen Fehler ergeben würde.

Hier ein kleines Programm, welches in BSL ohne Typfehler ausgeführt wird:

```
> (+ 1 (if (> 5 2) 1 "a"))
2
```

Das gleiche Programm wird in Typed Racket abgelehnt:

```
> (+ 1 (if (> 5 2) 1 "a"))
```
eval:10:0: Type Checker: type mismatch
expected: Number
given: (U String One)
in: "a"

Der Entwurf von Typsystemen, mit denen möglichst viele Programme überprüft werden können, ist ein sehr aktiver Forschungszweig in der Informatik.