17 Sprachunterstützung für Algebraische Datentypen

Wie wir in §7 “Datendefinition durch Alternativen und Zerlegung: Algebraische Datentypen” und §9.1 “Rekursive Datentypen” gesehen haben, sind algebraische Datentypen essentiell zur Strukturierung von komplexen Daten. Ähnlich dazu wie Signaturen und Datendefinitionen unterschiedlich gut durch Sprachmittel unterstützt werden können (§16 “Sprachunterstützung für Datendefinitionen und Signaturen”), gibt es auch bei algebraischen Datentypen Unterschiede darin, wie gut diese von der Programmiersprache unterstützt werden.

Wir werden uns vier verschiedene Arten anschauen, wie man algebraische Datentypen ausdrücken kann und diese im Anschluss bewerten.

Zu diesem Zweck betrachten wir folgende Datendefinitionen für arithmetische Ausdrücke:

Beispiel:

```
; An Expression is one of:
; - (make-literal Number)
; - (make-addition Expression Expression)
; interp. abstract syntax of arithmetic expressions
```

Als "Interface" für den Datentyp wollen wir die folgende Menge von Konstruktoren, Selektoren und Prädikaten betrachten:

```
; Number -> Expression
; constructs a literal exression
(define (make-literal value) ...)

; Expression -> Number
; returns the number of a literal
; throws an error if lit is not a literal
(define (literal-value lit) ...)

; [X] X -> Bool
; returns true iff x is a literal
(define (literal? x) ...)

; Expression Expression -> Expression
; constructs an addition expression
(define (make-addition lhs rhs) ...)

; [X] X -> Bool
; returns true iff x is an addition expression
(define (addition? x) ...)

; Expression -> Expression
; returns left hand side of an addition expression
; throws an error if e is not an addition expression
```

```
(define (addition-lhs e) ...)

; Expression -> Expression
; returns right hand side of an addition expression
; throws an error if e is not an addition expression
(define (addition-rhs e) ...)
```

Wir werden nun unterschiedliche Arten betrachten, wie wir diesen Datentyp repräsentieren können.

17.1 ADTs mit Listen und S-Expressions

Wie in §11.4 “S-Expressions” diskutiert, können verschachtelte Listen mit Zahlen, Strings etc. – also S-Expressions – als universelle Datenstruktur verwendet werden. Hier ist eine Realisierung der Funktionen von oben auf Basis von S-Expressions:

```
(define (make-literal n)
  (list 'literal n))

(define (literal-value l)
  (if (literal? l)
      (second l)
      (error 'not-a-literal)))

(define (literal? l)
  (and
   (cons? l)
   (symbol? (first l))
   (symbol=? (first l) 'literal)))

(define (make-addition e1 e2)
  (list 'addition e1 e2))

(define (addition? e)
  (and
   (cons? e)
   (symbol? (first e))
   (symbol=? (first e) 'addition)))

(define (addition-lhs e)
  (if (addition? e)
      (second e)
      (error 'not-an-addition)))

(define (addition-rhs e)
  (if (addition? e)
```

```
      (third e)
      (error 'not-an-addition)))
```

Auf Basis dieses Interfaces können nun Funktionen definiert werden, wie zum Beispiel ein Interpreter für die Ausdrücke:

```
(define (calc e)
  (cond [(addition? e) (+ (calc (addition-lhs e))
                          (calc (addition-rhs e)))]
        [(literal? e) (literal-value e)]))


> (make-addition
   (make-addition (make-literal 0) (make-literal 1))
   (make-literal 2))
'(addition (addition (literal 0) (literal 1)) (literal 2))


> (calc (make-addition
         (make-addition (make-literal 0) (make-literal 1))
         (make-literal 2)))
3
```

Beachten Sie, dass der Code von `calc` in keiner Weise von der Repräsentation der Ausdrücke abhängt sondern lediglich auf Basis des Interfaces definiert wurde.

17.2 ADTs mit Strukturdefinitionen

In dieser Variante verwenden wir Strukturdefinitionen, um das obige Interface zu implementieren. Wir haben die Namen so gewählt, dass sie mit denen, die durch `define-struct` gebunden werden, übereinstimmen, deshalb können wir in zwei Zeilen das gesamte Interface implementieren:

```
(define-struct literal (value))
(define-struct addition (lhs rhs))
```

Auch in dieser Variante können wir nun wieder Funktionen auf Basis des Interfaces implementieren. Die `calc` Funktion aus dem vorherigen Abschnitt funktioniert unverändert mit der `define-struct` Repräsentation von algebraischen Datentypen:

```
(define (calc e)
  (cond [(addition? e) (+ (calc (addition-lhs e))
                          (calc (addition-rhs e)))]
        [(literal? e) (literal-value e)]))
```

Allerdings haben wir durch `define-struct` nun eine neue, komfortablere Möglichkeit, um algebraische Datentypen zu verarbeiten, nämlich Pattern Matching (§10 “Pattern Matching”):

```
(define (calc e)
  (match e
    [(addition e1 e2) (+ (calc e1) (calc e2))]
    [(literal x) x]))


> (make-addition
   (make-addition (make-literal 0) (make-literal 1))
   (make-literal 2))
(addition (addition (literal 0) (literal 1)) (literal 2))


> (calc (make-addition
         (make-addition (make-literal 0) (make-literal 1))
         (make-literal 2)))
3
```

17.3 ADTs mit `define-type`

Wechseln Sie zum Nutzen von `define-type` auf die "Intermediate Student Language"

Eine neue Möglichkeit, um algebraische Datentypen zu repräsentieren, bietet das `2htdp/abstraction` Teachpack mit dem `define-type` Konstrukt. Im Unterschied zu `define-struct` bietet `define-type` direkte Unterstützung für Summentypen, daher kann der Summentyp `Expression` mit seinen unterschiedlichen Alternativen direkt definiert werden. Ein weiterer wichtiger Unterschied zu `define-type` ist, dass zu jedem Feld einer Alternative eine Prädikatsfunktion angegeben wird, die definiert, welche Werte für dieses Feld zulässig sind. Diese Prädikatsfunktionen sind eine Form von dynamisch überprüften Contracts (siehe §16.3 “Dynamisch überprüfte Signaturen und Contracts”).

```
(define-type Expression
  (literal (value number?))
  (addition (left Expression?) (right Expression?)))
```

Analog zu `define-struct` (§6.2 “Strukturdefinitionen”) definiert `define-type` für jede Alternative Konstruktor-, Selektor- und Prädikatsfunktionen, so dass wir Werte dieses Typs auf die gleiche Weise definieren können:

```
> (make-addition
   (make-addition (make-literal 0) (make-literal 1))
   (make-literal 2))
(addition (addition (literal 0) (literal 1)) (literal 2))
```

Allerdings wird bei Aufruf der Konstruktoren überprüft, ob die Felder die dazugehörige Prädikatsfunktion erfüllen. Der folgende Aufruf, der bei der `define-struct` Variante ausgeführt werden könnte, schlägt nun zur Laufzeit fehl:

```
> (make-addition 0 1)
```

```
make-addition: expects an undefined or a Expression, given 0
  in: the 1st argument of
      (->
       (or/c undefined? Expression?)
       (or/c undefined? Expression?)
       addition?)
  contract from: make-addition
  blaming: use
   (assuming the contract is correct)
  at: eval:13.0
```

Passend zu `define-type` gibt es auch eine Erweiterung von `match`, nämlich `type-case`. Mit Hilfe von `type-case` kann die `calc` Funktion nun wie folgt definiert werden:

```
(define (calc e)
  (type-case Expression e
    [literal (value) value]
    [addition (e1 e2) (+ (calc e1) (calc e2))]))


> (calc (make-addition
         (make-addition (make-literal 0) (make-literal 1))
         (make-literal 2)))
3
```

Der wichtigste Unterschied zwischen `match` und `type-case` ist, dass der Typ `Expression`, auf dem wir eine Fallunterscheidung machen möchten, explizit mit angegeben wird. Dies ermöglicht es der DrRacket Umgebung, bereits *vor* der Programmausführung zu überprüfen, ob alle Fälle abgedeckt wurden. Beispielsweise wird die folgende Definition bereits vor der Ausführung mit einer entsprechenden Fehlermeldung zurückgewiesen.

```
> (define (calc2 e)
    (type-case Expression e
      [addition (e1 e2) (- (calc2 e1) (calc2 e2))]))
type-case: syntax error; probable cause: you did not include a case for the literal
variant, or no else-branch was present
```

Der Preis für diese Vollständigkeitsüberprüfung ist, dass `type-case` nur ein sehr eingeschränktes Pattern Matching erlaubt. Beispielsweise ist es nicht erlaubt, Literale, verschachtelte Pattern, oder nicht-lineares Pattern Matching zu verwenden. Im Allgemeinen haben die Klauseln von `type-case` stets die Form `((variant (name-1 ... name-n) body-expression))`, wobei in `body-expression` die Namen `name-1` bis `name-n` verwendet werden können.