4.6.3 Gleichheit und Äquivalenz

Wir haben in Kapitel 4.2 unterschiedliche Gleichheitsbegriffe diskutiert und dann eine konstruktiv berechenbare Gleichheitsdefinition angegeben, die auch beim Rechnen mit unendlichen Datenstrukturen sinnvoll ist: die **strikte Gleichheit** (Def. 4.4). Intuitiv bedeutet dies, dass zwei Ausdrücke strikt gleich sind, wenn beide Ausdrücke zu einem identischen Grundkonstruktorterm reduzierbar sind. Wir haben auch gesehen, dass die strikte Gleichheit als normale Funktion definierbar ist. Wenn wir z.B. mit "`===`" eine boolesche strikte Gleichheitsoperation bezeichnen, dann kann man diese durch folgende schematischen Regeln definieren, wobei das erste Schema für alle 0-stelligen Konstruktoren C, das zweite Schema für alle n-stelligen Konstruktoren D und das dritte Schema für alle Konstruktoren $D \neq E$ verwendet wird:

C `===` C `= True`
$D\ x_1 \ldots x_n$ `===` $D\ y_1 \ldots y_n$ `=` x_1 `===` y_1 `&&` `...` `&&` x_n `===` y_n
$D\ x_1 \ldots x_n$ `===` $E\ y_1 \ldots y_m$ `= False`

Hierbei ist "`&&`" wie in Haskell als sequentielle Konjunktion definiert:

```
True  && x = x
False && _ = False
```

Genau diese Regeln werden durch den Compiler erzeugt, wenn man "`deriving Eq`" bei einer Datentypdeklaration angibt, wobei dann allerdings die Gleichheitsoperation "`==`" heißt. Betrachten wir z.B. die folgende Definition natürlicher Zahlen in Peano-Darstellung:

```
data Nat = Z | S Nat
  deriving Eq
```

In diesem Fall erzeugt der Compiler (Haskell wie auch Curry) die folgenden Instanzdefinition:

```
instance Eq Nat where
  Z   == Z   = True
  S x == S y = x == y
  Z   == S y = False
  S x == Z   = False
```

Die Operation "`==`" kann also verwendet werden, wenn man **testen** will, ob zwei Ausdrücke einen gleichen Wert haben, und je nach Ergebnis etwas anderes zurückliefern möchte. Daher kann man diese Operation auch als **Gleichheitstest** bezeichnen. Z.B. können wir diesen Test verwenden, um festzustellen, ob ein Element in einer Liste vorkommt:

```
elem :: Eq a => a  → [a]  → Bool
elem x []     = False
```

```
elem x (y:ys) = x == y || elem x ys
```

Nun ist aber eine wichtige Eigenschaft von Typklassen, dass Programmierer:innen eigene Instanzen definieren können. Da der Compiler nicht überprüfen kann, ob diese „sinnvoll“ definiert sind, kann man auch nicht intendierte Definitionen angeben. Z.B. könnten für `Nat` an Stelle von “`deriving Eq`” auch die folgende Instanzdefinition angeben:

```
instance Eq Nat where
  x == y = True
```

Hierdurch wäre also jedes Element in einer nichtleeren Liste enthalten:

```
> elem Z [S Z]
True
```

Um solche unbeabsichtigten Instanzdefinitionen auszuschließen, werden häufig noch Gesetze gefordert, die jede Instanzdefinition erfüllen muss. Tatsächlich findet man in vielen Textbüchern und Arbeiten zu Haskell die Aussage, dass “==” die Gleichheitsoperation ist, d.h. die Gleichheit auf Datentermen beschreibt. Tatsächlich ist dies aber nicht so!

Wenn wir uns die Dokumentation der Typklasse `Eq`[5] ansehen, dann wird dort zwar “==” als “equality” bezeichnet, aber es findet sich auch die Bemerkung:

> == is customarily expected to implement an equivalence relationship where two values comparing equal are indistinguishable by “public” functions.

Somit ist es durchaus beabsichtigt, dass e_1 == e_2 zu `True` auswertet auch wenn e_1 and e_2 keine gleichen sondern nur äquivalente (oder gar keine) Werte haben.
Wenn man z.B. komplexe Daten hat, die man schnell vergleichen möchte, könnte man diese Daten mit einem eindeutigen Index versehen und dann nur einen Indexvergleich durchführen. Betrachten wir hierzu den folgenden Datentyp für beliebige Werte, die mit einer eindeutigen Nummer versehen sind:

```
data IVal a = IVal Int a
```

Dann könnten man durchaus statt einer echten Wertgleichheit auch eine Wertäquivalenz definieren:

```
instance Eq a => Eq (IVal a) where
  IVal i1 _ == IVal i2 _  =  i1 == i2
```

Mit dieser Definition erhalten wir aber eventuell unbeabsichtige Ergebnisse:

```
> elem (IVal 1 'b') [IVal 1 'a']
True
```

[5] `http://hackage.haskell.org/package/base-4.12.0.0/docs/Data-Eq.html`

Dieses Ergebnis ist eigentlich nicht beabsichtigt, aber die Definition der Instanzdefinition ist nicht falsch, da hierdurch eine Äquivalenzrelation auf Werten definiert wird.

Ein weiteres Problem entsteht, wenn wir solche Äquivalenzrelationen statt der strikten Gleichheit im Kontext der logisch-funktionalen Programmierung verwenden. Betrachten wir hierzu die schon früher angegebene Möglichkeit des Berechnens des letzten Elementes einer Liste durch Lösen einer Gleichung:

```
last xs | _ ++ [e] == xs = e
  where e free
```

Da die Operation "==" keine strikte Gleichheit, sondern nur eine Äquivalenz festlegt, kann es passieren, dass `last` nicht das tatsächliche letzte Element einer Liste, sondern nur einen Wert zurück gibt, der äquivalent zum letzten Element ist. Z.B. erhalten wir die folgende Antwort, wenn wir das letzte Element einer Liste von `IVal`-Werten berechnen:

```
> last [IVal 1 'a']
IVal 1 _
```

Hier erhalten wir eine allgemeine Repräsentation aller Werte, die zu `IVal 1 'a'` äquivalent sind.

Zur Lösung dieser potenziellen Probleme benötigt man also eine wirkliche strikte Gleichheit statt einer allgemeineren Äquivalenzrelation. Um kompatibel mit Haskell zu sein, enthält Curry neben der Klasse `Eq` für Äquivalenzen auch eine Typklasse `Data` (weitere Details sind in [Hanus/Teegen 20] zu finden):

```
class Data a where
  (===)  :: a  → a  → Bool
  aValue :: a
```

Die Operation "===" ist die strikte Gleichheit, die nach dem obigen Schema definiert ist. Die nichtdeterministische Operation `aValue` liefert jeden Wert eines Typs zurück. Im Gegensatz zu anderen Typklassen gelten für `Data` einige Einschränkungen:

1. Es ist nicht erlaubt, explizite Instanzen für `Data` auf speziellen Typen anzugeben. Dies verhindert die Definition Instanzen mit einer Semantik verschieden von der strikten Gleichheit.

2. Instanzen für `Data` werden automatisch definiert für alle Typen erster Ordnung. Hier ist ein *Typ erster Ordnung*, falls dieser keine Funktionen als Teilkomponenten enthält. Dies bedeutet, dass alle Konstruktoren nicht-funktionale Argumenttypen haben und sonst nur Typen erster Ordnung verwenden.

Wenn wir also in der Definition von `last` die strikte Gleichheit verwenden, dann verhält es sich wie erwartet, d.h. man erhält tatsächlich immer genau das letzte Element einer Liste:

```
last :: Data a => [a]  → a
last xs | _ ++ [e] === xs = e
  where e free
```

Beachtet werden sollte, dass die Typsignatur impliziert, dass `last` nicht auf Listen von Funktionen angewendet werden kann, da man für Funktionen keine `Data`-Instanzen einfach definieren kann (dazu müsste man ein konstruktives Verfahren zur Feststellung der Gleichheit von Funktionen realisieren). Dies ist hier auch deswegen sinnvoll, weil wir zum Lösen von Gleichungen Werte für logische Variablen raten müssen, aber wir nicht alle Funktionen einfach aufzählen können.

Die Basistypen wie `Bool`, `Char`, `Int`, `Float` und `Ordering` sowie die Typkonstruktoren wie `Maybe`, `Either`, Listen und Tupel besitzen `Data`-Instanzen (Typkonstruktoren wie `Maybe` natürlich nur, wenn der Typparameter auch eine `Data`-Instanz hat). Damit können wir z.B. nicht-deterministisch alle Werte eines Typs aufzählen, indem wir die Operation `aValue` mit einem entsprechenden Typ annotieren:

```
> aValue :: Maybe Bool
Nothing
Just False
Just True
```

Da durch `aValue` jeder beliebige Wert eines Typs erzeugt wird und eine freie Variable für einen festen, aber unbekannten Wert steht, könnte man im Prinzip jede freie Variable durch `aValue` ersetzen, d.h.

```
... where x free
```

durch

```
... where x = aValue
```

ersetzen. Eine solche Ersetzung freier Variablen ist tatsächlich die Grundlage der Curry-Implementierung KiCS2, die Curry-Programme nach Haskell übersetzt [Braßel *et al.* 11]. Wie wir aber unten noch sehen werden, kann man durch eine direkte Verwendung freier Variablen eine effizientere Auswertung erreichen.

Freie Variablen

Freie Variablen werden durch Narrowing instanziiert, wenn sie als (verlangte) Argumente von Funktionen vorkommen, die ausgewertet werden. Betrachten wir z.B. eine Additionsfunktion auf natürlichen Zahlen (in Peano-Darstellung):

```
add :: Nat  → Nat  → Nat
add Z     n = n
add (S m) n = S (add m n)
```

Wenn "`add x (S Z)`" ausgewertet wird, wird die freie Variable `x` mit einem der Konstruktoren `Z` und `S` instanziiert, um dann eine Regel anzuwenden. Da Funktionen als Muster nicht vorkommen, können freie Variablen nicht mit Funktionen instanziiert werden. Um also eine unbeabsichtigte Verwendung freier Variablen zu vermeiden, wird der Typ freier Variablen immer mit einen `Data`-Kontext eingeschränkt, d.h. *es existieren keine rein polymorphen freie Variablen.*

Diese Einschränkung ist auch deswegen sinnvoll, weil wir ja auch schon diskutiert haben, dass der Applikationsoperator rigide ist, d.h. dass der Ausdruck "`x 2`" kann nicht ausgewertet werden, sondern er suspendiert. Da freie Variablen immer einen `Data`-Kontext haben, aber keine `Data`-Instanzen für Funktionen existieren, werden solche Ausdrücke schon zur Übersetzungszeit zurückgewiesen.

Wie wir schon gesehen habe, können wir mittels Narrowing Gleichungen lösen. Da die strikte Gleichheit "`===`" eine ganz normale Funktionsdefinition ist, können wir Gleichungen lösen, indem wir diese mittels Narrowing einfach ausrechnen. Wir können wir z.B. die Differenz zweier Zahlen (hier: `1` und `2`) durch Lösen einer Gleichheit mit Addition berechnen:

```
> add x (S Z) === S (S Z)  where x free
{x = Z} False
{x = S Z} True
{x = S (S Z)} False
{x = S (S (S _a))} False
```

Wenn wir an den negativen Resultaten nicht interessiert sind, können wir die Operation `solve` verwenden:

```
> solve $ add x (S Z) === S (S Z) where x free
{x = S Z} True
```

Zu beachten ist, dass diese Verwendung der strikten Gleichheit nur für Datentypen möglich ist, für die alle Konstruktoren durch eine `data`-Deklaration eingeführt wurden und somit mittels des obigen Schemas Regeln für die strikte Gleichheit existieren, die dann mittels Narrowing verwendet werden. Bei Standarddatentypen wie z.B. `Int`, `Float` oder `Char` ist dies nicht möglich, da man z.B. bei der Gleichung `x===3` entweder `x` an `3` bindet und `True` ausgibt, oder man bindet `x` an alle möglichen Zahlen verschieden von `3` und gibt `False` zurück. Da letzteres nicht so einfach möglich ist, suspendiert z.B. die Curry-Implementierung PAKCS in dieser Situation (ebenso beim Vergleich zweier Variablen). Dagegen enthält die Curry-Implementierung KiCS2 intern eine Peano-artige Darstellung ganzer Zahlen, sodass KiCS2 hierauf nicht suspendiert (dafür wird in KiCS2 das Residuation-Prinzip nicht unterstützt). Wenn man allerdings weiß, dass man nur positive Lösungen berechnen will, dann kann man die Gleichheit effizienter durch Unifikation lösen.

Gleichheitsconstraints und Unifikation

Neben der strikten Gleichheit "`===`" unterstützt Curry, ähnlich wie Prolog, auch Unifikation von Datentermen. Hierzu existiert ein **Gleichheitsconstraint** "`=:=`", welches eine Kombination der Auswertung zu Datentermen (strikte Gleichheit) und Unifikation ist und den folgenden Typ hat:

```
(=:=) :: Data a => a → a → a
```

Der Typ ist also genau wie bei der strikten Gleichheit, aber im Gegensatz zu "`===`" werden beim Gleichheitsconstraint nur `True`-Werte berechnet. Dafür werden Variablen nicht nur instanziiert, sondern es werden auch Variablen an andere Variablen oder Datenterme gebunden, ohne diese zu instanziieren, wodurch der Suchraum eingeschränkt werden kann. Das Gleichheitsconstraint kann also als *Optimierung der strikten Gleichheit* aufgefasst werden, welches man immer benutzen kann, wenn man nur `True` berechnen will, z.B. in Bedingungen von Regeln oder bei `solve`. Betrachten wir z.B. die Gleichung

```
> solve $ S x === S y where x,y free
```

Diese wird dadurch gelöst, dass `x` und `y` an `Nat`-Datenterme gebunden werden, wodurch wir unendlich viele Lösungen erhalten:

```
{x = Z, y = Z} True
{x = (S Z), y = (S Z)} True
{x = (S (S Z)), y = (S (S Z))} True
{x = (S (S (S Z))), y = (S (S (S Z)))} True
⋮
```

Diese unendlich vielen Lösungen werden durch das Gleichheitsconstraint "`=:=`" dadurch vermieden, dass einfach eine Variable an die andere gebunden wird, ohne einen konkreten Wert zu raten:

```
> S x =:= S y where x,y free
{x=y} True
```

Die Benutzung von `solve` ist hierbei nicht notwendig, weil durch "`=:=`" nur positive Lösungen berechnet werden, d.h. "`=:=`" entspricht der strikten Gleichheit aus Definition 4.4. Im Gegensatz zu "`===`" ist aber die genaue operationale Bedeutung des Gleichheitsconstraint "`=:=`" nicht als Standardfunktion definierbar, sodass wir die folgende „Metadefinition" zur Auswertung von "e_1 `=:=` e_2" angeben:

1. Werte e_1 und e_2 zur Kopfnormalform aus.
2. Unterscheide nun die folgenden Fälle:
   - `x =:= y`: binde die Variable `x` an `y`, d.h. das Ergebnis ist die Substitution $\{\texttt{x} \mapsto \texttt{y}\}$ und der Wert `True`.

- `x =:=` $C\ t_1 \ldots t_n$: Falls die Variable `x` nur in Funktionsaufrufen in $t_1, \ldots, t_n$ vorkommt (sonst Fehlschlag wegen "occur check"), binde `x` an $C\ y_1 \ldots y_n$ (wobei $y_1, \ldots, y_n$ neue Variablen sind) und werte den Ausdruck

  y_1`=:=`t_1 `&` ... `&` y_n`=:=`t_n

  aus.
- $C\ t_1 \ldots t_n$ `=:= x`: Berechne `x =:=` $C\ t_1 \ldots t_n$.
- $C\ s_1 \ldots s_n$ `=:=` $C\ t_1 \ldots t_n$: Werte den Ausdruck

  s_1`=:=`t_1 `&` ... `&` s_n`=:=`t_n

  aus.
- Sonst: Fehlschlag

Das Gleichheitsconstraint "`=:=`" kann also als Optimierung der strikten Gleichheit "`===`" betrachtet werden, weil hierdurch möglicherweise unendlich viele Alternativen durch eine Variablenbindung zusammengefasst werden. Z.B. führt die Auswertung von

```
> solve $ x===y && add x y === S Z where x,y free
```

zu einem unendlichen Suchraum (ohne Lösungen), weil `x` und `y` an unendlich viele `Nat`-Datenterme gebunden werden. Dagegen führt die Auswertung von

```
> x=:=y && add x y =:= S Z where x,y free
```

zu einem endlichen Suchraum, weil zunächst `x` and `y` gebunden wird und dann der Ausdruck

```
add x x =:= S Z
```

durch Raten der zwei Bindungen `x=Z` und `x=(S _)` (Narrowing von `add`) zu Fehlschlägen ausgewertet wird.

Es sollte noch erwähnt werden, dass die Curry-Implementierungen PAKCS und KiCS2 automatisch "`===`" durch "`=:=`" ersetzen,[6] wenn sichergestellt ist (hierzu wird eine Programmanalyse verwendet), dass nur positive Lösungen (d.h. `True`) berechnet werden sollen. Somit erhält man hier nur eine Bindung als Lösung:

```
> solve $ S x === S y where x,y free
{x=y} True
```

und der Ausdruck

```
> solve $ x===y && add x y === S Z where x,y free
```

[6] Dies ist erst ab Version 3.3.0 der Fall. In älteren Versionen wird "`==`" durch "`=:=`" ersetzt, was allerdings bei unbeabsichtigten Instanzen nicht korrekt ist.

hat einen endlichen Suchraum, weil in beiden Fällen "`===`" durch "`=:=`" ersetzt wird. Man muss sich in der Regel also nicht um den Unterschied zwischen "`===`" und "`=:=`" kümmern, sondern kann einfach die strikte Gleichheit "`===`" verwenden.
Man kann diese Optimierung durch Angabe der Option

```
-Dbindingoptimization=no
```

beim Aufruf von PAKCS oder KiCS2 abschalten.