Eine Demo der abhängig getypten Programmiersprache

Agda: Mit starken Typen abhängen

Joachim Breitner*

7. Juni 2012
Gulaschprogrammiernacht 12†, Karlsruhe

Andrew Hunt und David Thomas empfehlen

> „Lerne mindestens eine neue Programmiersprache pro Jahr."

Ganz in diesem Sinne widmen wir uns heute der Programmiersprache Agda. Agda ist zum einen eine funktionale Programmiersprache wie Haskell oder ML. Das alleine wäre jetzt nicht so spannend, aber Agda hat ein sehr mächtiges Typsystem mit sogenannten abhängigen Typen (*dependent types*), die dem Programmierer erlauben sehr viele Aussagen über sein Programm als Typen anzugeben und vom Compiler prüfen zu lassen. Das geht sogar so weit dass Agda auch ein Beweisprüfer wie Coq oder Isabelle ist, mit dem sich (konstruktive) Mathematik betreiben lässt.

Dass das geht, und vor allem dass das ziemlich gut geht, liegt an folgender „verhält-sich-zu-wie-Relation": Programme zu Typen wie Beweise zu Aussagen. Eine Aussage wie

> Wenn aus α β folgt, dann folgt, wenn β nicht gilt, auch dass α nicht gilt.

hat einen Beweis, etwa

> Es gelte β nicht. Beweis per Widerspruch. Angenommen, es gelte α. Dann folgt aus der Voraussetzung, dass β gilt. Das ist ein Widerspruch dazu, dass β nicht gilt, womit die Aussage gezeigt ist. ■

Der Aussage entspricht der Typ

```
(a → b) → ¬ b → ¬ a
```

und dem Beweis entspricht das Programm

*mail@joachim-breitner.de, http://www.joachim-breitner.de/. Diese Arbeit wurde durch ein Promotionsstipendium der Deutschen Telekom Stiftung gefördert.
†http://entropia.de/GPN12

λ f x a. x (f a).

Ein Typ, zu dem es ein Programm gibt, entspricht damit einer bewiesenen Aussage und Beweisen ist plötzlich nichts anderes mehr als Programmieren. Wir sind als auf der Gulasch-beweisnacht! Das Ganze ist übrigens bekannt als der Curry-Howard-Isomorphismus.

In diesem Vortrag werden wir beide Aspekte von Agda sehen, als Beispiel dient hier ein kleines Spiel:

> Auf dem Tisch liegt eine Anzahl Murmeln. Abwechselnd nimmt jeder Spieler ein paar Murmeln; mindestens eine und höchstens sechs. Wer die letzte Murmel nimmt, verliert.

1 Der erste Versuch

Unsere Aufgabe ist es nun einen Simulator für dieses einfache Spiel zu schreiben. Dazu legen wir eine neue Agda-Datei an und importieren erstmal ganz viele Module, die wir jetzt oder später brauchen werden:

```
module Pebbels1 where

open import Data.Nat
open import Data.List
open import Data.Bool
open import Data.Nat.DivMod
open import Data.Nat.Properties
open import Data.Fin using (Fin; toℕ; zero; suc)
open import Data.Fin.Props
open import Relation.Binary.PropositionalEquality
open import Function
open import Relation.Binary
open import Data.Empty
open import Relation.Nullary.Negation
open ≡-Reasoning
open DecTotalOrder decTotalOrder using () renaming (refl to ≤-reflv)
import Algebra
open Algebra.CommutativeSemiring commutativeSemiring using (+-comm)

open import DivModUtils
```

Das letzte Modul ist ein Hilfsmodul für diesen Vortrag mit ein paar arithmetischen Lemmas, die wir hier nicht widergeben wollen.

Als nächstes definieren wir ein paar Typen für unsere Aufgabe. Ein Spielzug ist eine natürliche Zahl, diese Paken wir in einen eigenen Datentypen um nicht zum beispiel Spielzüge und Anzahl der Murmlen auf dem Tisch durcheinander zu bringen. Eine Spielstrategie ist eine

Funktion, die jeder natürlichen Zahl einen Spielzug zuweist. Für diese UTF8-Sonderzeichen gibt es im Emacs im Agda-Modus spezielle Eingabefolgen, etwa `\bn` für ℕ und `\to` für →.

```
data Move : Set where
  pick : ℕ → Move
picked : Move → ℕ
picked (pick k) = k
Strategy = ℕ → Move
```

Nun definiern wir die Simulation, erstmal wie in einer stink-normalen funktionalen Programmiersprache wie Haskell. Neu ist hierbei höchstens das **with**-Konstrukt, es entspricht einem case ... of, nur dass in der Fallunterscheidung alle Parameter der Funktion erneut wiederholt werden können – das wird später noch wichtig. Da wir die Parameter nicht ändern Der Unterstricht steht für eine Variable, die uns nicht

```
play : ℕ → Strategy → Strategy → List ℕ
play 0 _ _ = []
play n p1 p2 with p1 n
... | p = n :: play (n ∸ picked p) p2 p1
```

Nun können wir zwei Spieler mit recht einfachen Strategien definieren; der erste nimmt immer zwei Murmeln, der zweite immer zwei:

```
player1 : Strategy
player1 _ = pick 1
player2 : Strategy
player2 _ = pick 2
```

Schon können wir mit dem Code herumspielen. Erst wird er mittels Strg C L geprüft und kompiliert. Mittels Strg C N könen wir einen Ausdruck auswerten, etwa `play 5 player1 player2`, das wertet zur Liste `5 :: 4 :: 2 :: 1 :: []` aus (das ist [5,4,3,2,1] in Haskell-Syntax).

Wir sehen, dass hier Spieler eins gewinnt, aber immer von Hand zu zählen ist uns zu mühsam, also definieren wir eine Funktion die prüft, ob eine Liste eine gerade Anzahl von Einträgen hat. Diese ist polymorph, also funktioniert mit beliebigen Listen-Elementen A, aber der Parameter ist implizit (da in geschweiften Klammern) und muss beim Aufruf von evenList nicht angegeben werden.

```
evenList : {A : Set} → List A → Bool
evenList [] = true
evenList (_ :: xs) = not (evenList xs)
```

Zuletzt können wir die Funktion definieren, die sagt, ob der erste Spieler Gewinnt.

```
winner : ℕ → Strategy → Strategy → Bool
winner n p1 p2 = evenList (play n p1 p2)
```

Damit wertet `play 5 player1 player2` zu `true` aus.

2 Bescheißerei!

Leider lässt sich dieser Code ganz schön an der Nase herumführen. Nehmen wir folgenden Spieler:

```
player0 : Strategy
player0 _ = pick 0
```

Offensichtlich wird dieser Spieler nie verlieren, wie wir mit Strg C N stichprobenhaft nachprüfen können. Oder, vielleicht weniger offensichtlich, folgenden Spieler:

```
playerN : Strategy
playerN n = pick (n ∸ 1)
```

Dieser lässt seinem Gegenüber immer genau eine Murmel übrig und kann damit auch nicht verlieren.

In herkömmlichen Programmiersprachen würde man jetzt vermutlich in play eine Abfrage einfügen, ob der Spieler versuchte, keine Murmel zu nehmen, und dann z.B. eine Exception schmeißen. Es geht aber auch eleganter: Wir wollen dass der Compiler ungültige Spielzüge schon beim Kompilieren ausschließt.

Wir müssen nun also sichergehen dass die Anzahl der genommenen Zahlen mindestens 1 und höchstes 6 ist. Eine Möglichkeit wäre es, einen Aufzählungstypen mit 6 Elemente nzu definieren, aber das wäre hässlich, weil wir damit nicht mehr schön rechnen können, außerdem möchten wir den Code später vielleicht verallgemeinern und die Anzahl der Murmeln, die man nehmen darf, konfigurierbar machen.

Statt dessen betreten wir jetzt die Welt der abhängigen Typen. Das heißt dass in den Typen auch Werte auftauchen können. Insbesondere gibt es Typen für (fast) beliebige Aussagen über Werte, etwa „der Vektor xs hat n Elemente“ oder, hier relevanter, „die Zahl n ist kleiner als die Zahl m“. Werte von diesem Typ sind dann Beweise, dass die Aussage stimmt. Wenn eine Funktion jetzt neben einem Wert auch einen solchen Beweis erwartet, dann kann ich sie nur aufrufen, wenn ich auch einen Beweis angebe. Die Funkion selbst wird vermutlich den Beweis nicht anschauen, aber kann sich darauf verlassen, dass die Aussage stimmt – sonst hätte ich keinen Beweis konstruieren können und die Funktion auch nicht aufrufen können.

Wir möchten sichergehen dass ein Wert vom Typ Move immer auch ein gültiger Zug ist. Dazu erwarten wir dass ein Move nicht nur aus einer natürlichen Zahl besteht, sondern auch aus einem Beweis dass die Zahl größer 0 und kleiner 7 ist:

```
data Move : Set where
  pick : (n : ℕ) → 1 ≤ n → n ≤ 6 → Move


picked : Move → ℕ
picked (pick k _ _) = k
```

Beachte dass wir beim ersten Parameter nicht nur den Typ (ℕ) angeben, sondern ihm auch einen Namen geben, den wir im *Typ* des zweiten Parameters wieder verwenden können.

Da wir nur per picked auf den Move zugriefen, muss die play-Funktion nicht geändert werden. Allerdings müssen die Spieler jetzt Beweise mit liefern. Agda kann uns helfen, diese zu finden, wir geben also erstmal nur ein Loch an:

```
player1 : Strategy
player1 _ = pick 1 ? ?
player2 : Strategy
player2 _ = pick 1 ? ?
```

Nach dem Laden (Strg C L) werden daraus Löcher, die man inspizieren kann, so kann man sich mit Strg C , das aktuelle Ziel anzeigen, also den Typ, der an diesem Loch erwartet wird.

In einfachen Fällen kann Agda sogar selbst herausfinden, welcher Code hier reinmuss, dazu probieren wir Strg C A (A für „auto“) und erhalten:

```
player1 : Strategy
player1 _ = pick 1 (s≤s z≤n) (s≤s z≤n)
player2 : Strategy
player2 _ = pick 2 (s≤s z≤n) (s≤s (s≤s z≤n))
```

Was heißt das? Anscheinend gibt es Funkionen namens z≤n und s≤s, mit denen man sich Kleiner-Gleich-Beweise zusammenbasteln kann. Mit der mittleren Maustaste kann man direkt zur Definition einer Funktion springen. Dort sehen wir den folgenden Code:

```
data _≤_ : Rel ℕ Level.zero where
  z≤n : ∀ {n} → zero ≤ n
  s≤s : ∀ {m n} (m≤n : m ≤ n) → suc m ≤ suc n
```

In Worten: z≤n ist ein Beweis dass Null kleiner-gleich jeder Zahl ist, und dass ich einen Vergleich zweier Zahlen auf ihren Nachfolger übertragen kann. Also hat s≤s z≤n) den Typ suc zero ≤ suc m für jede natürliche Zahl m. In unserem Fall heißt das, wie gewünscht 1 ≤ 6.

So, was haben wir jetzt davon? Wir können nun weder player0 noch playerN schreiben, denn es wird uns nicht gelingen 1 ≤ 0 oder n ≤ 6 für beliebige n zu beweisen. Damit haben wir jetzt ungültige Spieler vollständig ausgeschlossen!

3 Der optimale Spieler

Zum Abschluss möchten wir jetzt noch einen besseren Spieler implementieren, nämlich einen, der die optimale Strategie fährt:

> Nimm immer so viele Murmeln, dass danach ein Vielfaches von 7 plus eine weitere Murmel liegen bleiben.

Das geht immer, es sei denn, es liegt bereits eine mehr als ein Vielfaches von 7 Murmeln auf dem Tisch. Wenn wir also z.B. mit 100 Murmeln beginnen und der erste Spieler diese Strategie fährt, gewinnt er auf jeden Fall.

Implementieren wir also diesen Spieler. Der Ausdruck k mod 7 gibt einen Wert vom Typ Fin 7 zurück; das sind natürliche Zahlen kleiner als 7. (Wir hätten also Fin 6 auch selbst für Move verwenden können; aber so wars lehrreicher). Die Funktion **to**ℕ macht daraus wieder eine normale natürliche Zahl, während bounded r den Beweis **to**ℕ r ≤ 6 liefert.

```
opt : Strategy
opt k with pred k mod 7
... | zero = pick 1 (s≤s z≤n) (s≤s z≤n)
... | (suc r) = pick (toℕ (suc r)) (s≤s z≤n) (bounded r)
```

Nun wollen wir beweisen, dass die optimale Strategie wirklich die optimale ist. Wie würde wohl so ein Lemma aussehen? Vermutlich so, wobei 1' eine eins vom Typ Fin 7 ist.

```
opt-is-opt : ∀ n s → n mod 7 ≢ 1' → winner n opt s ≡ true
```

Aber um dort anzukommen brauchen wir ein Hilfslemma für den anderen Spieler.

```
opt-is-opt2 : ∀ n s → n mod 7 ≡ 1' → winner n s opt ≡ false
```

Beginnen wir mit dem zweiten Fall. Generell kommt man beim Beweisen am besten voran, wenn man die Struktur des Programms, über das man den Beweis führt, nachvollzieht.

```
opt-is-opt2 0 _ ()
opt-is-opt2 (suc n) s eq with s (suc n)
opt-is-opt2 (suc n) s eq | pick k 1≤k k≤6 = cong not $
  opt-is-opt (suc n ∸ k) s (lem-sub-p n k eq 1≤k k≤6)
```

Das Lemma lem-sub-p habe ich bereits vorbereitet (Modul DivModUtils):

```
lem-sub-p : ∀ n p → (suc n mod 7 ≡ 1') → 1 ≤ p → p ≤ 6 → ((suc n ∸ p) mod 7 ≢ 1')
```

Nun zum Beweis des zweiten Falls. Der wird sehr ähnlich aussehen, wieder brauchen wir ein Lemma analog zu dem bereits vorbereiteten, diesmal für opt. Das sieht dann so aus:

```
lem-opt : ∀ n → suc n mod 7 ≢ 1' → (suc n ∸ picked (opt (suc n))) mod 7 ≡ 1'
lem-opt n neq with n divMod 7
lem-opt .(q * 7) neq | result q zero = ⊥-elim (neq (mod-lemma q 6 1'))
lem-opt .(1 + toℕ r + q * 7) neq | result q (suc r) = begin
  (1 + toℕ r + q * 7 ∸ toℕ r) mod 7
    ≡⟨ cong (λ y → (1 + y ∸ toℕ r) mod 7) $ +-comm (toℕ r) (q * 7) ⟩
  (1 + q * 7 + toℕ r ∸ toℕ r) mod 7
    ≡⟨ cong (λ y → y mod 7) $ m+n∸n≡m (1 + q * 7) (toℕ r) ⟩
  (1 + q * 7) mod 7
    ≡⟨ mod-lemma q 6 1' ⟩
  1' ■
```

Das Lemma -mod-lemma ist auch aus DivModUtils\:

```
mod-lemma : ∀ x d (r : Fin (suc d)) → (toℕ r + x * suc d) mod suc d ≡ r
```

Der zweite Fall sieht wiederum dem ersten Fall ähnlich. Entscheident ist, wo wir das lem-opt einbauen: Nach dem Aufruf von **with** opt (suc n) wird der Zusammenhang zwischen pick k 1≤k k≤6 und opt (suc n) vergessen sein, den brauchen wir allerdings um lem-opt anwenden zu können. Daher müssen wir auf beides *gleichzeitig* matchen.

```
opt-is-opt 0 _ _ = refl
opt-is-opt (suc n) s neq with opt (suc n) | lem-opt n neq
opt-is-opt (suc n) s neq | pick k 1≤k k≤6 | eq = cong not $
  opt-is-opt2 (suc n ∸ k) s eq
```

Damit ist gezeigt dass unser Spieler immer gewinnt, wenn er gewinnen kann.

4 Termination

Das hat im Vortrag leider keinen Platz mehr gehabt, aber der Vollständigkeit halber will ich hier noch darauf eingehen:

Bisher waren die Funktion play und die Beweise, die darauf aufbauen, rot markiert. Damit zeigt Agda dass es nicht weiß ob play terminiert, also garantiert in keine Endlosschleife läuft. Warum ist das wichtig? Weil man mit Endlosschleifen beliebige Aussagen beweisen kann:

```
unsinn : 42 < 7
unsinn = unsinn
```

Damit wäre es jetzt doch wieder möglich, den Betrüger zu implementieren.

```
n≤6 : {n : ℕ} → n ≤ 6
n≤6 = n≤6

1≤n : {n : ℕ} → 1 ≤ n
1≤n = 1≤n

playerN : Strategy
playerN n = pick (n ∸ 1) 1≤n n≤6
```

Nun erkennt Agda in einfachen Fällen die Termination, etwa in der Funktion evenList. Hier ruft evenList (x :: xs) im rekursiven Fall evenList xs auf, das Argument ist also ein Teil des Parameters und Agda ist sich sicher dass diese Funktion irgendwann fertig ist.

Bei play ist das zwar auch der Fall, aber Agda sieht das nicht und wir müssen ihm auf die Sprünge helfen. Leider müssen wir diesen Beweis direkt in der Definition von play führen; das erfordert zusätzliche Parameter und damit größere Änderungen am Code und den Beweisen:

```
open import Induction.WellFounded
open import Induction.Nat

play : (n : ℕ) → Acc _<_ n → Strategy → Strategy → List ℕ
play 0 _ _ _ = []
play (suc n) (acc rec) p1 p2 with p1 (suc n)
... | pick 0 () _
... | pick (suc k) _ _ = (suc n) :: play (suc n ∸ suc k) (rec _ (s≤s (n∸m≤n k n))) p2 p1

winner : ℕ → Strategy → Strategy → Bool
winner n p1 p2 = evenList (play n (Subrelation.well-founded ≤⇒≤' <-well-founded n) p1 p2)

opt-is-opt : ∀ n s → n mod 7 ≢ 1' → winner n opt s ≡ true

opt-is-opt1 : ∀ n a s → n mod 7 ≢ 1' → evenList (play n a opt s) ≡ true
opt-is-opt2 : ∀ n a s → n mod 7 ≡ 1' → evenList (play n a s opt) ≡ false

opt-is-opt2 0 _ _ ()
opt-is-opt2 (suc n) (acc a) s eq with s (suc n)
... | pick 0 () _
... | pick (suc k) 1≤k k≤6 = cong not $
  opt-is-opt1 (suc n ∸ suc k) _ s (lem-sub-p n (suc k) eq 1≤k k≤6)


lem-opt : ∀ n → suc n mod 7 ≢ 1' → (suc n ∸ picked (opt (suc n))) mod 7 ≡ 1'
lem-opt n neq with n divMod 7
lem-opt .(q * 7) neq | result q zero = ⊥-elim (neq (mod-lemma q 6 1'))
lem-opt .(1 + toℕ r + q * 7) neq | result q (suc r) = begin
  (1 + toℕ r + q * 7 ∸ toℕ r) mod 7
    ≡⟨ cong (λ y → (1 + y ∸ toℕ r) mod 7) $ +-comm (toℕ r) (q * 7) ⟩
  (1 + q * 7 + toℕ r ∸ toℕ r) mod 7
    ≡⟨ cong (λ y → y mod 7) $ m+n∸n≡m (1 + q * 7) (toℕ r) ⟩
  (1 + q * 7) mod 7
    ≡⟨ mod-lemma q 6 1' ⟩
  1' ■

opt-is-opt1 0 _ _ _ = refl
opt-is-opt1 (suc n) (acc a) s neq with opt (suc n) | lem-opt n neq
... | pick 0 () _ | _
... | pick (suc k) 1≤k k≤6 | eq = cong not $
  opt-is-opt2 (suc n ∸ (suc k)) _ s eq

opt-is-opt n s = opt-is-opt1 n _ s
```