On Extensions to Definitional Equality in Agda
Or: Making Agda See More

Andreas Abel

Department of Computer Science
Ludwig-Maximilians-University Munich

10th Agda Implementors’ Meeting
Gothenburg, Sweden
15 September 2009
A Problem on the Agda List

- Conor `foo` type-checks, but `boo` does not.
  
  ```agda
data Unit : Set where
    unit : Unit

T : Unit -> Set
T unit = Nat

foo : (x : Unit) -> T x
foo unit = 0

boo : (x : Unit) -> T x
boo x = 0
```

- Agda does not see that `x = unit`.
- But this is so obvious that beginners are confused...
Eta-Expansion for the Unit Type

- Expand neutral terms of type \( \text{Unit} \) to \( \text{unit} \).

\[
\uparrow^{\text{Unit}} x = \text{unit}
\]

\[
\text{boo} : (x : \text{Unit}) \rightarrow T \times
\]

\[
\text{boo} \ x = 0
\]

- We need to check

\[
\vdash x \mapsto 0 : (x : \text{Unit}) \rightarrow T \times
\]

- Introducing new variable \( x \) into the context

\[
x : \text{Unit} \vdash 0 : T(\uparrow^{\text{Unit}} x)
\]

\[
x : \text{Unit} \vdash 0 : T \text{unit}
\]

\[
x : \text{Unit} \vdash 0 : \text{Nat}
\]
Eta-Expansion for Data Types with One Constructor

- **Data types** with one constructor are non-dependent record types.

```haskell
data Sigma (A : Set)(B : A -> Set) : Set where
  pair : (fst : A) -> (snd : B fst) -> Sigma A B
```

- **Destructors** `fst` and `snd` are generated:

  ```haskell
  fst : {A : Set}{B : A -> Set}(p : Sigma A B) -> A
  snd : {A : Set}{B : A -> Set}(p : Sigma A B) -> B (fst{A}{B}p)
  ```

- **Eta**: `Sigma` is only inhabited by **pairs**.

  \[ \uparrow^{\Sigma A B} x = \text{pair} \{A\} \{B\} \quad (\uparrow^A (\text{fst} \{A\} \{B\} x)) \quad (\uparrow^B (\text{fst} \{A\} \{B\} x) (\text{snd} \{A\} \{B\} x)) \]
Eta-Expansion for the Empty Type

- Empty types have no inhabitants, so any two inhabitants of an empty type are equal.

```haskell
data Empty : Set where
```

- Introduce an internal, inconstructible constant $\ast$.

```haskell
\uparrow^{\text{Empty}} x = \ast
```

- Does not compromise consistency, since there is already an $x : \text{Empty}$. See [Abel/Coquand/Pagano, TLCA’09].
Propositional Equality

- Complication: Non-linearity.

\[
\text{data } \text{Id} \{A : \text{Set}\}(a : A) : A \to \text{Set} \text{ where }
\]

\[
\text{refl} : \text{Id} a a
\]

\[
\text{refl} : \{A : \text{Set}\}(a : A) \to \text{Id} \{A\} a a
\]

- \(\text{Id}\{T\} tt'\) is inhabited by \(\text{refl}\{T\}\{t\}\) if \(t\) definitionally equal to \(t'\), otherwise empty.

\[
\uparrow\text{Id}\{T\} tt' = \begin{cases} 
\text{refl}\{T\}\{t\} & \text{if } \vdash t = t' : T \\
\ast & \text{otherwise}
\end{cases}
\]

- Paper: [Abel, NBE’09].
Trust Me

- **Subsumes**
  
  \[
  \text{postulate}
  \]
  
  \[
  \text{trustMe} : \{A \to \text{Set}\} (a b : A) \to \text{Id} \{A\} a b
  \]

- **Binds name** `trustMe` **to value**

  \[
  \uparrow \{A : \text{Set}\} (a : A) (b : A) \to \text{Id} \{A\} a b \text{ trustMe}
  \]

- **Behaves like a `\(\lambda\)**

  \[
  (\uparrow (x : A) \to B x \ r) s = \uparrow^B s (r s)
  \]

- **trustMe \(t\ \ t\)** **evaluates to** `refl`. 
Non-Overlapping Pattern Inductive Types

Vectors

data Vec (A : Set) : Nat -> Set where
  vnil : Vec A zero
  vcons : {n : Nat}(hd : A)(tl : Vec A n) -> Vec A (suc n)

Need to termination check data declaration!

data REmpty : Set where
  bla : REmpty -> REmpty
Smart Case Example: The Filinski Test

\[
\text{tripleF} \, : \, (f \, : \, \text{Bool} \to \text{Bool}) \, (x \, : \, \text{Bool}) \to f \, (f \, (f \, x)) \, \Rightarrow \, f \, x
\]

\[
\text{tripleF} \, f \, \text{true} \, \text{with} \, f \, \text{true}
\]

... | true = ?

... | false \, \text{with} \, f \, \text{false}

... | true = ?

... | false = ?

\[
\text{tripleF} \, f \, \text{false} \, \text{with} \, f \, \text{false}
\]

... | false = ?

... | true \, \text{with} \, f \, \text{true}

... | true = ?

... | false = ?
Sorted Lists

Another problem from the Agda list:

data SList : Nat -> Set where
  snil : SList zero
  scons : {n : Nat}(m : Nat) -> True (n <= m) -> SList n -> SList m

max : Nat -> Nat -> Nat
max n m with n <= m
...  | true   = m
...  | false  = n
Sorted Lists, Insertion

- **Naive try:**

  ```agda
  insert : {n : Nat}(m : Nat) -> SList n -> SList (max n m)
  insert m snil = scons m tt snil
  insert m (scons n p l) with n <= m
  insert m (scons n p l) | true =
    scons m ? (scons n p l)
  insert m (scons n p l) | false =
    scons n (maxLemma p (notLe ?)) (insert m l)
  ```

- Intuitively, the holes have trivial proofs.
- In practice, Agda does not know \( n <= m = \text{true} \).
Smart Case in MiniAgda and PiSigma-1.0

- When checking \( e \) in `case \( v \) of \( p \rightarrow e \)`, add rewrite \( v \rightarrow p \) to evaluator.
- Make sure \( v \) is in normal form, also w.r.t. to rewrites.
- Detect inconsistencies, like \( v \rightarrow \text{true} \), \( v \rightarrow \text{false} \).
- When adding new rewrite, apply it to old rewrites.
- Implemented prototypically in MiniAgda and old PiSigma (Thorsten Altenkirch, Nicolas Oury).
- Termination? Completeness?
Discussion

- Time is ripe for eta!
- Smart case should be thoroughly researched.
- Prototypical Agda implementation!?
- More rewriting?