(Dependent) Types for Proofs and Programs

Marcin Benke

MIM UW

May 8, 2010
The result type of a function can depend on the value of the argument.

Theorems and specifications are types

\[
sort : \forall (xs : \text{List } A) \rightarrow \exists (ys : \text{List } A).\text{Sorted}(xs, ys)
\]

Proofs are programs and programs are proofs

\[
\text{lemma} : \forall x \ y \rightarrow x \cdot y \cdot x^{-1} \approx y
\]

\[
\text{lemma } x \ y = \begin{align*}
x \cdot y \cdot x^{-1} & \approx \langle \text{comm } \_ \_ \langle \cdot \text{-cong } \rangle \text{refl } \rangle \\
y \cdot x \cdot x^{-1} & \approx \langle \text{assoc } \_ \_ \_ \rangle \\
y \cdot (x \cdot x^{-1}) & \approx \langle \text{refl } \langle \cdot \text{-cong } \rangle \text{proj}_2 \text{ inverse } \_ \rangle \\
y \cdot \varepsilon & \approx \langle \text{proj}_2 \text{ identity } \_ \rangle \\
y & \square
\end{align*}
\]

(in case you wonder only: \( \forall = \) belong to the language — the rest is user-defined)

Proofs can be verified automatically via typechecking.
The Holy Grail of Correctness

- We want to know our programs are correct.
- Alas, verification is hard and expensive.
- “Verification is hard, let’s go shopping”... NOT.
- Make formulating and proving program properties as easy as possible (but not easier).
- C: arbitrary crashes, buffer overruns.
- Java: some protection, but still null pointer exceptions occur even in stock trading programs.
- No null pointers, no buffer overruns, types limit what particular function can do (e.g. a function $f : \text{Int} \to \text{Int}$ cannot launch missiles.
- But it’s by far not the whole story...
Why Dependent Types Matter?

(Altenkirch, McBride, McKinna)

- Conventional type systems validate programs with respect to a fixed set of criteria.
- Dependent types realize a continuum of precision from the basic assertions up to a complete specification of the program’s behaviour.
- Dependently typed programs are, by their nature, proof carrying code.
- In Java or Haskell, you can swap the then and else branches of conditionals and the compiler won’t complain.
- These type systems don’t care that pieces of data mean things.
- Dependent types do care. That is why dependent types matter.
Martin-Löf Type Theory

- Several versions by Per Martin-Löf 1972–84; extensions by Coquand, Dybjer et. al
- Original aim: foundations of constructive mathematics
- Fringe benefits:
  - proof assistants: Alf, Agda, Alfa, Agda2,…
  - programming with dependent types: Cayenne, Agda/Alonzo
  - linguistics: Grammatical Framework
- Propositions as sets: a proposition is interpreted as the set of its proofs.
- A proof assistant for constructive mathematics
- Interactive proof development
- A functional programming language (with dependent types)
  - Types help represent and enforce specification
  - Proof-carrying code
  - Interactive program development
- Closing the semantic gap between programs and their properties
Judgments

A is a type:

\[ A \text{ type} \]

\(a\) is an object of type \(A\):

\[ a : A \]

\(a\) and \(b\) are the same object of type \(A\):

\[ a = b : A \]

two types are identical:

\[ A = B \]
The Logical Framework

There is one primitive type of sets:

\[ \vdash \text{Set type} \]

Variables:

\[ x : A \in \Gamma \quad \Gamma \vdash \]
\[ \Gamma \vdash x : A \]

We stipulate that its elements are types:

\[ \Gamma \vdash A : \text{Set} \]
\[ \Gamma \vdash A \text{ type} \]

Note: assuming \( \text{Set} : \text{Set} \) (or type type) would lead to Girard’s paradox.
Dependent product:

\[ \frac{\Gamma \vdash A \text{ type} \quad \Gamma, (x : A) \vdash B \text{ type}}{\Gamma \vdash (x : A) \to B \text{ type}} \]

Introduction (function):

\[ \frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x : A. M : (x : A) \to B} \]

Elimination (application):

\[ \frac{\Gamma \vdash M : (x : A) \to B \quad \Gamma \vdash N : A}{\Gamma \vdash MN : B[N/x]} \]

Computation (beta-conversion):

\[ \frac{x : A \vdash M : B \quad N : A}{(\lambda x : A. M)N = M[N/x] : B[N/x]} \]
Extending the framework

Within this framework we can define a new sets by specifying:

- How is it formed from components (if any) — formation rules:

\[
\frac{A : Set \quad B : Set}{A \cup B : Set}
\]

- What are the canonical elements of the set — introduction rules:

\[
\frac{a : A}{in_1 \ a : A \cup B : Set}
\]
\[
\frac{b : B}{in_2 \ b : A \cup B : Set}
\]

- Elimination rules (reasoning, induction):

\[
C : (A \cup B \to Set) \quad c_1 : (a : A) \to C(in_1 \ a) \quad c_2 : (b : B) \to C(in_2 \ b)
\]

\[
E_{\cup} C \ c_1 \ c_2 \ x : C \ x
\]

- Equality rules (computation):

\[
E_{\cup} C \ c_1 \ c_2 \ (in_1 \ a) = c_1 a
\]
\[
E_{\cup} C \ c_1 \ c_2 \ (in_2 \ b) = c_2 b
\]
Disjoint sum in Agda

...or, in Agda notation:

data _⊎_ (A : Set) (B : Set) : Set where
  inj₁ : (a : A) → A ⊎ B
  inj₂ : (b : B) → A ⊎ B

E⊎ : {A B : Set} → (C : A ⊎ B → Set)
  → ((a : A) → C (inj₁ a))
  → ((b : B) → C (inj₂ b))
  → (x : A ⊎ B) → C x

E⊎ C c₁ c₂ (inj₁ a) = c₁ a
E⊎ C c₁ c₂ (inj₂ b) = c₂ b
**Natural numbers**

```
data ℕ : Set where
    zero : ℕ
    suc : (n : ℕ) → ℕ
```

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

```
_≤?_ : Decidable _≤_

zero  ≤? _ = yes z≤n
suc m  ≤? zero = no λ()
suc m  ≤? suc n with m  ≤? n
... | yes m≤n = yes (s≤s m≤n)
... | no m≰n = no (m≰n ∘ ≤-pred)
```
The empty set and consistency

The empty set has no canonical elements:

data \bot : Set where

\bot\text{-elim} : \forall \{w\} \{\text{Whatever : Set } w\} \to \bot \to \text{Whatever}
\bot\text{-elim} ()

We must ensure that the empty set remains uninhabited, e.g. by proving that every expression reduces to its canonical form (strong normalization).

data D where

lam : (D \to D) \to D

or

bad : \bot
bad = bad

would breaks consistency and Agda does not allow that.

But there are many traps: excluded middle or the axiom of choice can lead to trouble in some settings.
Equality

The identity type lets us reason about equality:

\[
\text{data } \equiv \{A : \text{Set}\} (x : A) : A \rightarrow \text{Set} \text{ where }
\]
\[
\text{refl} : x \equiv x
\]

\[
\text{data } \cong \{A : \text{Set}\} (x : A) : \{B : \text{Set}\} \rightarrow B \rightarrow \text{Set} \text{ where }
\]
\[
\text{refl} : x \cong x
\]

\[
\text{proof-irrelevance} : \{A B : \text{Set}\} \{x : A\} \{y : B\}
\]
\[
(p \ q : x \cong y) \rightarrow p \equiv q
\]
\[
\text{proof-irrelevance refl refl} = \text{refl}
\]

Note: with dependent product, \(\bot\), \(\uplus\) and \(\equiv\) are enough to define basically any Agda type (via reflection aka universes).
Universes

A universe consists of

- a set of codes for datatypes: $\text{Sig} : \text{Set}$
- a decoding function: $\text{T} : (\Sigma : \text{Sig}) \rightarrow \text{Set}$

A universe for indexed families indexed by $I$

$\text{Sig}_I = [\text{Arity}_I]$  
$\text{Arity}_I = \text{data} \ \text{Nil} (i : I) \mid \text{NonRec} (A : \text{Set}) (A \rightarrow \text{Sig}_I) \mid \text{Rec} I \text{Sig}_I$

Example: vectors

\[
\begin{align*}
\frac{n \in \mathbb{N}}{\text{Vec} \ n \ A \in \text{Set}} & \quad \frac{A \in \text{Set}}{\text{Vnil} \in \text{Vec} \ 0 \ A} & \quad \frac{x \in A \quad xs \in \text{Vec} \ n \ A}{\text{Vcons} \ x \ xs \in \text{Vec} \ (n + 1) \ A}
\end{align*}
\]

The signature for vectors:

$[\text{Nil} \ 0, \ \text{NonRec} \ A \lambda_. \ \text{NonRec} \ \mathbb{N} \ \lambda n. \ \text{Rec} \ n \ \text{Nil} \ (S \ n)]$
Specifications in Agda: AVL-trees

\[
data \sim : \mathbb{N} \rightarrow \mathbb{N} \rightarrow \text{Set}
where
\sim + : \forall \{n\} \rightarrow n \sim 1 + n
\sim 0 : \forall \{n\} \rightarrow n \sim n
\sim - : \forall \{n\} \rightarrow 1 + n \sim n
max : \forall \{m n\} \rightarrow m \sim n \rightarrow \mathbb{N}
\]

\[
data \mathsf{Tree} : \mathbb{N} \rightarrow \text{Set}
where
\phantom{0} \mathsf{leaf} : \mathsf{Tree} 0
\phantom{0} \mathsf{node} : \forall \{h^l h^r\} (\mathsf{bal} : h^l \sim h^r)
\phantom{0} (l : \mathsf{Tree} h^l) (k : \mathsf{KV}) (r : \mathsf{Tree} h^r)
\phantom{0} \rightarrow \mathsf{Tree} (1 + \text{max \, bal})
\]

\[
\mathsf{insert} : \forall \{h\} \rightarrow (k : \mathsf{Key}) \rightarrow \mathsf{Value} \, k \rightarrow \mathsf{Tree} \, h \rightarrow
\exists \lambda \, i \rightarrow \mathsf{Tree} \, (i \oplus h)
\]

Insert is now guaranteed to produce balanced trees.
Grammatical Framework

Grammatical Framework (Ranta et al.) is a (dependent types-based) environment for natural language processing, supporting:

- specification of abstract and concrete syntaxes,
- linearization (abstract to concrete syntax)
- parsing (concrete to abstract)
- translation (concrete syntax of one language to another)

The application of dependent types is quite natural; consider a rule for building a sentence from a Noun Phrase and Verb Phrase:

\[
\frac{a : NP \quad b : VP(a)}{a + b : S}
\]

where the choice of VP (number, gender, etc) depends on the choice of NP.

GF resource grammar library for 22 languages including Polish (by our student, Adam Slaski).