Why Dependent Types Matter

Thorsten Altenkirch

School of Computer Science
University of Nottingham

October 5, 2011
The Greek-ASCII dichotomy

- Programs are (were ?) written in ASCII . . .
- Papers in theoretical Computer Science use greek letters . . .
- Programmers don’t do proofs . . .
- Theoreticians don’t write programs . . .
- Can we bridge the gap?
Another observation

- Compilers don’t read comments . . .
- Sometimes they should!
- How can we make informal statements formal
- and checkable by our software tools?
A clarification

- Formal specifications cannot be the starting point of software development.
- The early stages are exploratory steps involving prototypes.
- In the beginning we don’t know much about the software we are developing.
- Exploratory steps / Consolidation steps.
- Specifications are one of the outputs of consolidation steps.
- Would like to guarantee that specifications and code fit together.
Proofs

- Unrealistic to hope that all relevant properties are decidable.
- Need proofs as formal objects which provide evidence that an assertion holds.
- Replace oracles (decision procedures which answer either yes or no) . . .
- . . . with evidence producing decision procedures.
- Potential of an economy of proofs (who is too blame?).
But how do we do it?

Programming Language + Logic
- Separation of language for programming and reasoning
- Possible for (almost) any programming language
- Conventional logic (1st order, classical)
- Geared to posthoc verification

Dependently Typed Programming
- Functional language with an expressive type system
- Reasoning emerges due to the Curry-Howard principle
- Intuitionistic logic
- Integration of reasoning and programming
Introduction

From Per to Ulf

Per Martin-Löf

Introduced Type Theory
As a new constructive foundation of Mathematics
since the mid 1970ies

Ulf Norell

Implemented the current Agda system
A functional programming language
and an interactive proof assistant
based on Type Theory
in his PhD in 2005
Rest of the talk

- A taste of Agda
- The Curry-Howard Principle
- Classical logic
- Recursion and induction
- Families of types
- Coinduction
- Design challenges
A taste of Agda
Safe lookup

Define an operation which extracts the nth element out of a list.

_ !!_ : List A → ℕ → A
xs !! n = ?
1st attempt

_!!_: List A → ℕ → A
[] !! n = ?
(x :: xs) !! zero = x
(x :: xs) !! suc n = xs !! n

- We cannot complete this program.
- Agda only allows complete pattern.
- A could be empty.
2nd attempt (use monad)

\[
_ \text{!!}_ : \text{List } A \rightarrow \mathbb{N} \rightarrow \text{Maybe } A \\
[] \text{!! } n = \text{nothing} \\
(a :: as) \text{!! } \text{zero} = \text{just } a \\
(a :: as) \text{!! } \text{suc } n = as \text{!! } n
\]

- We use the \textit{Maybe} monad.
- In Haskell (and other languages) this is built-in.
- Runtime errors may arise at any time.
From *Nat* and *List*

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

data List (A : Set) : Set where  
  [] : List A  
  _::_ : (x : A) (xs : List A) → List A
```
To \textit{Fin} and \textit{Vec}

\begin{verbatim}
data Fin : \mathbb{N} \to Set where
  zero : Fin (suc n)
  suc : (i : Fin n) \to Fin (suc n)
\end{verbatim}

\begin{verbatim}
data Vec (A : Set) : \mathbb{N} \to Set a where
  [] : Vec A zero
  _ :: _ : (x : A) (xs : Vec A n) \to Vec A (suc n)
\end{verbatim}
3rd attempt (use dependent types)

\[ \_ \_ !\_ !\_ : Vec A n \to Fin n \to A \]
\[ [] !\! () \]
\[ (x :: xs) !\! zero = x \]
\[ (x :: xs) !\! suc i = xs !\! i \]

- We have replaced \textit{List} with \textit{Vec} and \textit{Nat} with \textit{Fin}.
- No runtime errors.
- Using dependent types we can eliminate runtime errors
- But what if we read the index from external sources?
- We need to check but only once.
- Runtime errors are clearly localized.
The Curry-Howard principle
The Curry-Howard principle

- We can express certain constraints using dependent types.
- What are the limits of this technology?
- We can encode any logical formula as a dependent type.
- We assign to a logical formula the set of its proofs.

\[
\begin{align*}
\text{prop} & : \text{Set}_1 \\
\text{prop} & = \text{Set}
\end{align*}
\]

- Proving = constructing a program of this type.
Propositional Logic

Implication \( P \rightarrow Q \) is given by the type of functions from \( P \) to \( Q \).

Conjunction \( P \land Q \) is given by the type of pairs of elements of \( P \) and \( Q \).

\[
\text{data } \_ \land \_ (P \text{ Q : prop}) : 	ext{prop where}
\]
\[
\_, \_ : P \rightarrow Q \rightarrow P \land Q
\]

Disjunction \( P \lor Q \) is given by the disjoint union of elements of \( P \) and \( Q \).

\[
\text{data } \_ \lor \_ (P \text{ Q : prop}) : 	ext{prop where}
\]
\[
\text{left} : P \rightarrow P \lor Q
\]
\[
\text{right} : Q \rightarrow P \lor Q
\]
How to prove?

\[ P \land (Q \lor R) \iff (P \land Q) \lor (P \land R) \]
Write a program!

\[
distrib : P \land (Q \lor R) \rightarrow (P \land Q) \lor (P \land R)
\]
\[
distrib (p, \text{left } q) = \text{left } (p, q)
\]
\[
distrib (p, \text{right } r) = \text{right } (p, r)
\]

- Observe that the program is invertible.
- Hence we can prove \(\iff\).
- This provides a different explanation than the truth table.
- More accessible to programmers?!
Predicate logic

universal quantification  The set of proofs of $\forall x : A. P x$ is the set of dependent function $(x : A) \rightarrow P x$.

existential quantification  The set of proofs of $\exists x : A. P x$ is the set of dependent pairs:

\[
\text{data } \exists (A : \text{Set}) (P : A \rightarrow \text{prop}) : \text{prop where } \\
\_ : (a : A) \rightarrow P a \rightarrow \exists A P
\]
How to prove?

\[ \forall x : A. P x \rightarrow Q \iff (\exists x : A. P x) \rightarrow Q \]
Write a program!

\[
\text{curry} : ((\exists A \ P) \rightarrow Q) \rightarrow (a : A) \rightarrow P \ a \rightarrow Q
\]
\[
\text{curry} \ x = \lambda \ a \ x' \rightarrow x \ (a, x')
\]
\[
\text{curry}' : ((a : A) \rightarrow P \ a \rightarrow Q) \rightarrow ((\exists A \ P) \rightarrow Q)
\]
\[
\text{curry}' \ x \ (a, y) = x \ a \ y
\]

- Generalized form of currying.
  \[(P \land Q \rightarrow R) \iff (P \rightarrow Q \rightarrow R)\]
- Not just a logical equivalence . . .
- but an isomorphism.
- Not all equivalences are isomorphisms.
Classical logic
What about the excluded middle?

- We cannot prove:
  
  \[ \text{tnd} : \{ P : \text{prop} \} \rightarrow P \lor \neg P \]

  and other classical principles.

- Because our logic is intuitionistic and constructive.
The classical Babelfish

Classical reasoner says:  |  Babelfish translates to:

<table>
<thead>
<tr>
<th>$A \lor B$</th>
<th>$\neg(\neg A \land \neg B)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\exists x : S. Px$</td>
<td>$\neg \forall x : S. \neg Px$</td>
</tr>
</tbody>
</table>

**Negative translation**

$A \lor \neg A$ is translated to $\neg(\neg A \land \neg \neg A)$ which is constructively provable.

A classical reasoner is somebody who is unable to say anything positive.

However, while the axiom of choice is provable (easily)

$(\forall a : A. \exists b : B. R a b) \rightarrow \exists f : A \rightarrow B. \forall a : A. R a (f a)$

its translation is not:

$(\forall a : A. \neg \forall b : B. \neg R a b) \rightarrow \neg \forall f : A \rightarrow B. \neg \forall a : A. R a (f a)$
Recursion and induction
How to prove?

\[ \forall i j k : \mathbb{N}. (i + j) + k = i + (j + k) \]

where

\[ - + _{-} : \mathbb{N} \to \mathbb{N} \to \mathbb{N} \]

\[ \text{zero} + n = n \]

\[ \text{suc} \ m + n = \text{suc} \ (m + n) \]
Recursion and induction

Equality

The only proof that $a = b$ is \textit{refl} if $a$ and $b$ are identical.

\begin{verbatim}
data _ ≡ _ (x : A) : A → Set where
  refl : x ≡ x
\end{verbatim}

We can prove that every function respects equality using pattern matching:

\begin{verbatim}
cong : (f : A → B) \{ a b : A \} → a ≡ b → f a ≡ f b
cong f refl = refl
\end{verbatim}
Write a program!

\[ \text{assoc} : (i \; j \; k : \mathbb{N}) \rightarrow (i + j) + k \equiv i + (j + k) \]

\[ \text{assoc zero} \; j \; k = \text{refl} \]

\[ \text{assoc} \; (\text{suc} \; i) \; j \; k = \text{cong suc} \; (\text{assoc} \; i \; j \; k) \]

- This is a recursive program!
- Induction = primitive recursion
- What is the result of \( \text{assoc} \; 2 \; 7 \; 3 \)?
Proof irrelevance

• Indeed \textit{assoc} always returns \textit{refl}.
• There is no point in running \textit{assoc}.
• However, it is important to know that it exists.
• Is this always the case?
Deciding equality

- Equality for 1st order datatypes (like \(\mathbb{N}\)) is decidable.

  - This is witnessed by a boolean function:

    \[
    _ \equiv_? _ : \mathbb{N} \to \mathbb{N} \to \text{Bool}
    \]

    \[
    \text{zero } \equiv_? \text{ zero } = \text{ true}
    \]

    \[
    \text{zero } \equiv_? \text{ suc n } = \text{ false}
    \]

    \[
    \text{suc n } \equiv_? \text{ zero } = \text{ false}
    \]

    \[
    \text{suc n } \equiv_? \text{ suc m } = \text{ n } \equiv_? \text{ m}
    \]

- How do we know that this function decides equality?
Decidability

- To decide a proposition means we can show there is a proof . . .
- or there cannot be one.

```haskell
data Dec (P : Set) : Set where
  yes : (p : P) → Dec P
  no : (¬p : ¬P) → Dec P
```

- A predicate is *decidable*, if each instance can be decided.
- To say that equality is decidable amounts to
  \((m \ n : \mathbb{N}) \rightarrow Dec (m \equiv n)\)
Deciding equality . . .

\[ \equiv \, m n : \mathbb{N} \rightarrow \text{Dec} \, (m \equiv n) \]

zero \equiv zero = yes refl

zero \equiv suc n = no (\lambda ())

suc n \equiv zero = no (\lambda ())

suc n \equiv suc m \text{ with } n \equiv m

suc n \equiv suc m \mid yes p = yes (\text{cong suc } p)

suc n \equiv suc m \mid no np =

\quad no (\lambda q \rightarrow np (\text{cong pred } q))

- Similar structure as the boolean function.
- Instead of returning true or false . . .
- \equiv returns yes or no and evidence that this is the correct answer.
- Indeed \equiv’s type already completely specifies its behaviour.
Families of types
How to define \( \leq \) ?

\[
\text{data } _\leq _\text{: } \mathbb{N} \rightarrow \mathbb{N} \rightarrow \text{Set where}
\]

\[
\text{le0 : zero } \leq n
\]

\[
\text{leS : } m \leq n \rightarrow \text{suc } m \leq \text{suc } n
\]

- \( m \leq n \) is the set of derivation trees showing that \( m \) is less or equal \( n \).
- E.g. \( \text{leS (leS le0)} : 2 \leq 4 \)
- How to prove transitivity?
Write a program!

```latex
\textbf{data} \_ \leq \_ : \mathbb{N} \to \mathbb{N} \to \text{Set} \quad \text{where}
 \begin{align*}
  \text{le0} : \text{zero} & \leq n \\
  \text{leS} : m & \leq n \to \text{suc} \ m \leq \text{suc} \ n
\end{align*}

\begin{align*}
\text{trans} : \forall \{l\ m\ n\} \to l & \leq m \to m \leq n \to l \leq n \\
\text{trans} \ \text{le0} \ p & = \text{le0} \\
\text{trans} (\text{leS} \ p) \ (\text{leS} \ q) & = \text{leS} (\text{trans} \ p \ q)
\end{align*}
```
How to define provability?

\textbf{data} \_ \vdash \_ : \textit{Context} \to \textit{Formula} \to \textit{Set} \ 	extbf{where}

- \textit{ass} : \Gamma \cdot A \vdash A
- \textit{weak} : \Gamma \vdash A \to \Gamma \cdot B \vdash A
- \textit{app} : \Gamma \vdash A \Rightarrow B \to \Gamma \vdash A \to \Gamma \vdash B
- \textit{abs} : \Gamma \cdot A \vdash B \to \Gamma \vdash A \Rightarrow B

- Minimal propositional logic.
- $\Gamma \vdash A$ is the set of derivation trees proving $A$ from $\Gamma$.
- This is a natural deduction style definition.
- Corresponds to typed $\lambda$-calculus with de Bruijn variables.
- Define typed terms directly, not untyped terms and typing relation.
Combinatory logic

```haskell
data ⊢ sk : Context → Formula → Set where
  ass : Γ · A ⊢ sk A
  weak : Γ ⊢ sk A → Γ · B ⊢ sk A
  app : Γ ⊢ sk A ⇒ B → Γ ⊢ sk A → Γ ⊢ sk B
  K : Γ ⊢ sk A ⇒ B ⇒ A
  S : Γ ⊢ sk (A ⇒ B ⇒ C) ⇒ (A ⇒ B) ⇒ A ⇒ C
```

- Can prove equivalence
  \[ Γ ⊢ A ⇔ Γ ⊢ sk A \]
  by recursion / induction over derivation trees.
- Key lemma: \[ Γ · A ⊢ sk B → Γ ⊢ sk A ⇒ B \]
Coinduction
Streams

- While *List* $A$ represents the set of finite sequences.
- *Stream* $A$ is the set of infinite sequences.

\[
data \text{Stream} \ (A : \text{Set}) : \text{Set} \ \text{where} \nonumber\]
\[
\_ :: \_ : A \rightarrow \infty (\text{Stream} \ A) \rightarrow \text{Stream} \ A 
\]

- To define *Stream* $A$ we exploit the notion of lifted types $\infty A$.
- *Delay* : $\# : A \rightarrow \infty A$
- *Force* : $\flat : \infty A \rightarrow A$
Computations on streams

- Define the sequence of numbers starting with $n$:

$$f_{\text{rom}} : \mathbb{N} \to \text{Stream } \mathbb{N}$$
$$f_{\text{rom}} \ n = n :: \#(f_{\text{rom}} (\text{suc } n))$$

- Can we prove?

$$\text{mapStream } \text{suc } (f_{\text{rom}} \ n) \approx f_{\text{rom}} (\text{suc } n)$$

where

$$\text{mapStream} : (A \to B) \to \text{Stream } A \to \text{Stream } B$$
$$\text{mapStream } f \ (a :: as) = f \ a :: \#(\text{mapStream } f \ (\text{♭} as))$$
Infinite proofs

- Since proofs = programs
- proofs over infinite datastructures
- can be infinite datastructures themselves.
- Extensional equality of streams (bisimilarity).

\[
\text{data } _\sim _\approx \{ A \} : (xs \ ys : \text{Stream } A) \rightarrow \text{Set where } \\
_\approx : \forall x \{ xs \ ys \} \ (xs \approx : \infty (♭xs \approx ♭ys)) \rightarrow x :: xs \approx x :: ys
\]

- Can construct an infinite proof:

\[
nthLem : (n : \mathbb{N}) \rightarrow \text{mapStream } suc \ (\text{from } n) \approx \text{from } (\text{suc } n) \\
nthLem n = \text{suc } n :: ♭nthLem \ (\text{suc } n)
\]
Design challenges
Design challenges

Termination checking

- Need to ensure programs are total.
- Agda termination checker verifies structural recursion / guardedness.
- Non-structural / non-guarded total programs can be implemented...
- ...but the effort is considerable.
- Need extensible but safe termination checker.
- Reduction to total core language instead of external checker?
Efficient implementation of IDEs

- Interactive program development creates new challenges.
- Symbolic evaluation.
- Typechecking incomplete programs.
- Issues with scaling to larger sized developments.
- Agda: problems with records due to $\eta$-expansion.
Efficent compilation

- Naive compilation creates considerable overhead.
- Many expressions don’t need to be computed, no computational content.
- See Edwin Brady’s work on compilation of dependently typed languages.
- Dependent type provide ample opportunities for novel optimisations (e.g. exploiting finiteness)
Interfacing the real world

- Monads provide a clear interface to effectful programming.
- Haskell’s IO monad is opaque.
- How to reason about it?
- What happens when I/O expression appear in dependent types?
- See Wouter Swierstra’s work on functional specification of I/O.
Proof automatisation

- Want to create proofs (semi) automatically.
- Instead of providing a tactic language . . .
- exploit reflection!
- Use Agda to write tactics.
- E.g. see the recent work of Struth and Foster.
Reusability

- Finer types
- reduce reusability
- E.g. instead of lists we have vectors, sorted lists, contexts, . . .
- Hard to implement a useful library.
- Use generic programming to derive datatypes
- and share common structure
- Topic of an ongoing research project (Nottingham, Oxford, Strathclyde)
Tricky datatypes

- Agda allows very flexible mutual definitions.
- induction-recursion.
- induction-induction.
- which are not well understood semantically.
- Topic of an ongoing research project (Nottingham, Swansea, Strathclyde).
Extensionality

- The principle of extensionality is not provable in Agda
  \[ ext : (f \ g : A \rightarrow B) \rightarrow ((a : A) \rightarrow f \ a \equiv g \ a) \rightarrow f \equiv g \]
- Lack of quotient types.
- New proposal: identify types upto isomorphism (Voevodsky)
- Don’t want to add axioms
- because they destroy the computational structure of the theory.
- Can these principles be eliminated?
Conclusions

- DTP: new perspective on certified program development.
- Reasoning emerges from a rich type discipline.
- Covers the whole spectrum from programming to verification.
- Allows a pay-as-you go approach to certification.
- New challenges . . .
- . . . but many of them seem to be unavoidable.