Programming and Program Verification with Dependent Types in Agda

Andreas Abel

Department of Computer Science
Ludwig-Maximilians-University Munich

Seminar des Lehrstuhls 8, Theoretische Informatik
Friedrich-Alexander-Universität Erlangen
8 May 2012
Type Systems

- Basic types \texttt{Int}, \texttt{Word} from machine (CPU)
- Data types (record, objects, variants) describe structures
- Higher types (function, polymorphism, monads) describe computations
- Strong typing: check at compile-time
  1. Enable code optimizations
  2. Catch errors early
  3. Document code and libraries
  4. Allow (partial) verification
Types for verification

- Types for physical units (F#)
- Polymorphic types: A more general type has less inhabitants
- Security types: Control access to resources
- Resource types: Control complexity (time/space)
- **Dependent types**
  - Express arbitrary (logical) properties of a function in its type
  - Pre- and postconditions
- **Vision**: dependent types as universal type system
- Lightweight incorporation of security, resource, ... types
Dependent types

- Dependent function type $\Pi$:
  \[(x : A) \rightarrow B \ x\]
  Codomain $B$ can depend on argument value $x$.

- Dependent record type $\Sigma$:
  record Seq (A : Set) : Set where
    field
    length : $\mathbb{N}$
    elements : Vec A length
  The type of a field can depend on the value of a previous field.

- Full dependent types:
  \[
  \begin{align*}
  \text{Sum} &: \mathbb{N} \rightarrow \text{Set} \\
  \text{Sum} 0 &= \mathbb{N} \\
  \text{Sum} (n + 1) &= \mathbb{N} \rightarrow \text{Sum} n
  \end{align*}
  \]
  Shape of type can depend on value.
Curry-Howard-Correspondence

- Constructive propositional logic corresponds to simply-typed lambda-calculus

<table>
<thead>
<tr>
<th>Logic</th>
<th>Programming</th>
</tr>
</thead>
<tbody>
<tr>
<td>Proposition</td>
<td>Type</td>
</tr>
<tr>
<td>Proof</td>
<td>Program</td>
</tr>
<tr>
<td>Implication</td>
<td>Function type</td>
</tr>
<tr>
<td>Conjunction</td>
<td>Cartesian product</td>
</tr>
<tr>
<td>Disjunction</td>
<td>Disjoint sum</td>
</tr>
<tr>
<td>Truth</td>
<td>Biggest type (unit)</td>
</tr>
<tr>
<td>Absurdity</td>
<td>Least type (empty)</td>
</tr>
</tbody>
</table>

- One language for types and specifications
- One language for programming and proving
Curry-Howard for Predicate Logic

- Constructive predicate logic corresponds to dependently-typed lambda-calculus (Martin-L"of Type Theory)

<table>
<thead>
<tr>
<th>Logic</th>
<th>Programming</th>
</tr>
</thead>
<tbody>
<tr>
<td>Proposition</td>
<td>Type</td>
</tr>
<tr>
<td>Proof</td>
<td>Program</td>
</tr>
<tr>
<td>Universal q.</td>
<td>Depend. function t.</td>
</tr>
<tr>
<td>Existential q.</td>
<td>Depend. pair t.</td>
</tr>
<tr>
<td>Predicates</td>
<td>Indexed type</td>
</tr>
<tr>
<td>Type</td>
<td>Type</td>
</tr>
<tr>
<td>Term</td>
<td>Program</td>
</tr>
</tbody>
</table>

- Unification of concepts $\Rightarrow$ lean language.
- Program extraction: discard proofs
Some Dependently Typed Languages

- **NuPrl**: Extensional Type Theory (ETT)
  Cornell, US (Bob Constable, ...) 1980s–

- **Coq**: Impredicative Intensional Type Theory (ITT)
  INRIA, France (Huet, Coquand, ...) 1980s–

- **Epigram**: Predicative ITT
  McKinna, McBride 2000s–

- **Agda**: Predicative ITT
  Chalmers, Sweden 1990s–: Alf, Alfa, Agda, AgdaLight
  Agda 2 (Norell, ...) 2006–
Agda 2

- Haskell-like syntax +
  - mixfix identifiers
  - Parametrized modules
- People
  - Implemented by Ulf Norell
  - Library by Nils Anders Danielsson (Nisse)
  - Maintained by Ulf, Nisse, and me
  - Contributions by 20 more...
- Compiles to
  - Haskell (M. Takeyama)
  - C (via Epic) (Gustafsson)
  - JavaScript (A. Jeffrey)
- Agda is implemented in Haskell
  
cabal install Agda
Agda tour (interactive)

- Numbers
- Vectors
- Logic
- Sorted lists
Theory of Agda

- Type checking = proof checking
- Termination
- Automation
  - Higher-order unification
  - Program synthesis = proof search
  - Overloading via instance search (type classes)
- Program extraction
  - Identify computationally irrelevant terms (proofs)
  - Eliminate uniquely determined terms (singletons)
  - Exploit rich type information for optimizations
Termination

- Looping programs prove anything
  \[
  \text{loop} : \bot
  \]
  \[
  \text{loop} = \text{loop}
  \]
- Agda accepts only terminating programs
- Current termination checker searches for lexicographic structural descent
- Plan: certify termination by sized types
  - Type-based termination (Pareto 1996, Giménez 1998, ...)
  - My PhD thesis: non-dependent language + sized types
  - MiniAgda: study sized types and pattern matching
- Coinduction: non-terminating, but productive programs
  - Induction: consume input in each iteration
  - Coinduction: produce output in each iteration
Consistency

- Logical consistency proved by model
- Term model: Model types by sets of terminating programs
- Proves also decidability of type checking
- No model for all of Agda published
  - Dependently-typed lambda-calculus
  - Inductive types
  - Indexed types
  - Recursive functions and pattern matching
  - Universes
- Current status: extrapolation from models that consider a subset of features
- Plan: simpler model through sized types
Unification

- Type-checking decidable, but infeasible
- Excessive amount of (repetitive) type arguments

\[
_\circ_ : \{A \ B \ C : \text{Set}\} \rightarrow (B \rightarrow C) \rightarrow (A \rightarrow B) \rightarrow A \rightarrow C
\]

\[
g \circ f = \lambda x \rightarrow g (f \ x)
\]

\[
_\circ_ \{A\} \{B\} \{C\} \ g \ f = \lambda x \rightarrow g (f \ x)
\]

- Reconstruction: Omit arguments, solve by unification

- Higher-order unification = solve equations over lambda-terms
- Undecidable (Hilbert’s 10th problem) \implies \text{constraints}
  - Agda: Sound and complete constraint simplification
  - Only finds unique solutions
  - Coq: more solving by first-order unification, but also wrong solutions
Higher-Order Unification

- Some constraints have no unique solutions
  \[
  X \text{ true} = \text{ true} \\
  X \ x \ x \ = \ x \ + \ 1
  \]

- Postpone constraints, try to gather more information

- Unification for records:
  \[\text{Abel, Pientka, TLCA 2011}\]

- Beware of type checking modulo constraints:
  If constraints are unsolvable, term might be ill-typed/looping

- Satisfying theoretical account lacking!
Computational irrelevance

- Declare some programs as proofs by typing
  \[ f : \, .\,(x : A) \to B \, x \]

  \( f \) and \( B \) do not computationally depend on \( x \)

- \( A \) is “squash \( A \)”.

- Irrelevant arguments can be erased during program extraction.
Conclusion

- Agda: Unified proof/programming language via Curry-Howard
- Lean syntax, but complicated theory
- Vision: universal type system
- Subsume the next 700 type systems
- Need: more automation for boilerplate!
- Tactics!
Get involved!

- Try Agda!
- Formalize your papers with Agda!
- Write libraries!
- Write documentation (Agda wiki)!
- Fix bugs code.google.com/agda/issues/
- Contribute features!
- Contribute theory!