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 Int, 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 Π:

```
(x : A) → B x
```

  Codomain B can depent on argument value x.

- Dependent record type Σ:

```
record Seq (A : Set) : Set where
  field
    length   : ℕ
    elements : Vec A length
```

  The type of a field can depend on the value of a previous field.

- Full dependent types:

```
Sum : ℕ  → Set
Sum 0       = ℕ
Sum (n + 1) = ℕ  → Sum n
```

  Shape of type can depend on value.

Curry-Howard-Correspondence

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

Logic		Programming	
Proposition		Type	
Proof		Program	
Implication	$\Rightarrow$	Function type	$\rightarrow$
Conjunction	$\wedge$	Cartesian product	$\times$
Disjunction	$\vee$	Disjoint sum	$+$
Truth	$\top$	Biggest type (unit)	$\top$
Absurdity	$\bot$	Least type (empty)	$\bot$

- 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)

Logic		Programming	
Proposition		Type	
Proof		Program	
Universal q.	$\forall x : \tau.\ P\,x$	Depend. function t.	$(x : A) \to B\,x$
Existential q.	$\exists x : \tau.\ P\,x$	Depend. pair t.	$(x : A) \times B\,x$
Predicates	$P\,t$	Indexed type	$A\,t$
Type	τ	Type	A
Term	t	Program	t

- 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

```
loop : ⊥
loop = 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

```
_∘_ : {A B C : Set} → (B → C) → (A → B) → A → C
g ∘ f = λ x → g (f x)

_∘_ {A} {B} {C} g f = λ x → g (f x)
```

- Reconstruction: Omit arguments, solve by unification
- Higher-order unification = solve equations over lambda-terms
- Undeciable (Hilbert's 10th problem) $\Rightarrow$ 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 true = true
X x x  = x + 1
```

- Postpone constraints, try to gather more information
- Unification for records:

  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) → 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!`