Introduction to Dependent Types

Chris Warburton

Monday 19th August, 2013

Outline

Motivation

Mechanics

Examples

Discussion

Why Types?

- Programmers make mistakes
- Software evolves, specifications drift
- Enforce interfaces
  - Esp. for libraries and higher-order functions
- Keep track of tedious properties
  - Escaping, overflow, resources (memory, files, sockets), etc.
- Expose structure and properties of problems and algorithms

Why not tests?

- Types and tests are orthogonal
  - We should use both!
- Tests look for bugs in a depth-first way
  - Check detailed properties of specific inputs
- Types look for bugs in a breadth-first way
  - Check limited properties of *every input*

Why Dependent Types?

- Allow *arbitrary* types
  - Almost Turing-complete
- *Simple* foundation
  - Extends Lambda Calculus
- Seamlessly combines *programming* and *theorem-proving*
  - Curry-Howard: Types are theorems, programs are their proofs
- Incremental
  - Focus on properties we care about (eg. code injection, time-bounds)

Type Systems

- Assigns at least one type to every value
  - Dynamic types are just large sums
- Can only *restrict* code
  - "I can already do that in $LANG!"
- Purely syntactic
  - $1 + 1 \mathrel{!=} 2$
- Only exist at compile-time
- Consistent
  - Necessarily *incomplete*: Some correct programs won't type-check

Dependent Type Systems

- One language
  - Types are first-class values, just like everything else
  - We can compute our types as part of our program
- Types can *depend* on values:
  - *Dependent* functions

    ```
    id : (t : Type) -> t -> t
    ```

  - *Dependent* pairs:

    ```
    (t : Type ** t)
    ```

Types Are Terms

```
intOrChar : Bool -> Type
intOrChar True  = Int
intOrChar False = Char

data (=) : a -> b -> Type where
  refl : (x : a) -> (x = x)

unitTestCheck : (allUnitTestsPass = True)
unitTestCheck = refl allUnitTestsPass
```

Dependent Functions

- Result *type* can contain argument *value*
- No specific values, so use *universal quantification* **forall**

Dependent Pairs

- Second value's *type* can contain first *value*
- Each pair can differ, so we get *existential quantification*
  there exists

Demo

Applications

- Theorem proving (esp. Coq, Agda)
- Tricky datastructures/algorithms
- Security
  - Handling malicious input (eg. PDF)
  - Proof-carrying code
- Inductive programming

Drawbacks

- Consistent type systems must be *total*
  - Defined for all inputs
  - Must terminate or co-terminate
- Library problem: damages code re-use
- Verbose
  - Dependent pattern-matching, views, etc.

On-going Research

- Library problem
  - Ornaments
  - Observational equality
  - Higher-dimensional Type Theory
- Automation
  - Theorem proving
  - Type inference
  - Termination checking
- UI
  - More informative types can inform our IDEs

Summary

- Pros:
  - Verifications of arbitrary properties
  - Incremental safety
  - Theorem-proving
- Cons:
  - Verbose
  - Totality-checking
  - Limited code re-use

Thanks

Questions?