Dependent Types

Stephanie Weirich

University of Pennsylvania

Compose 2015

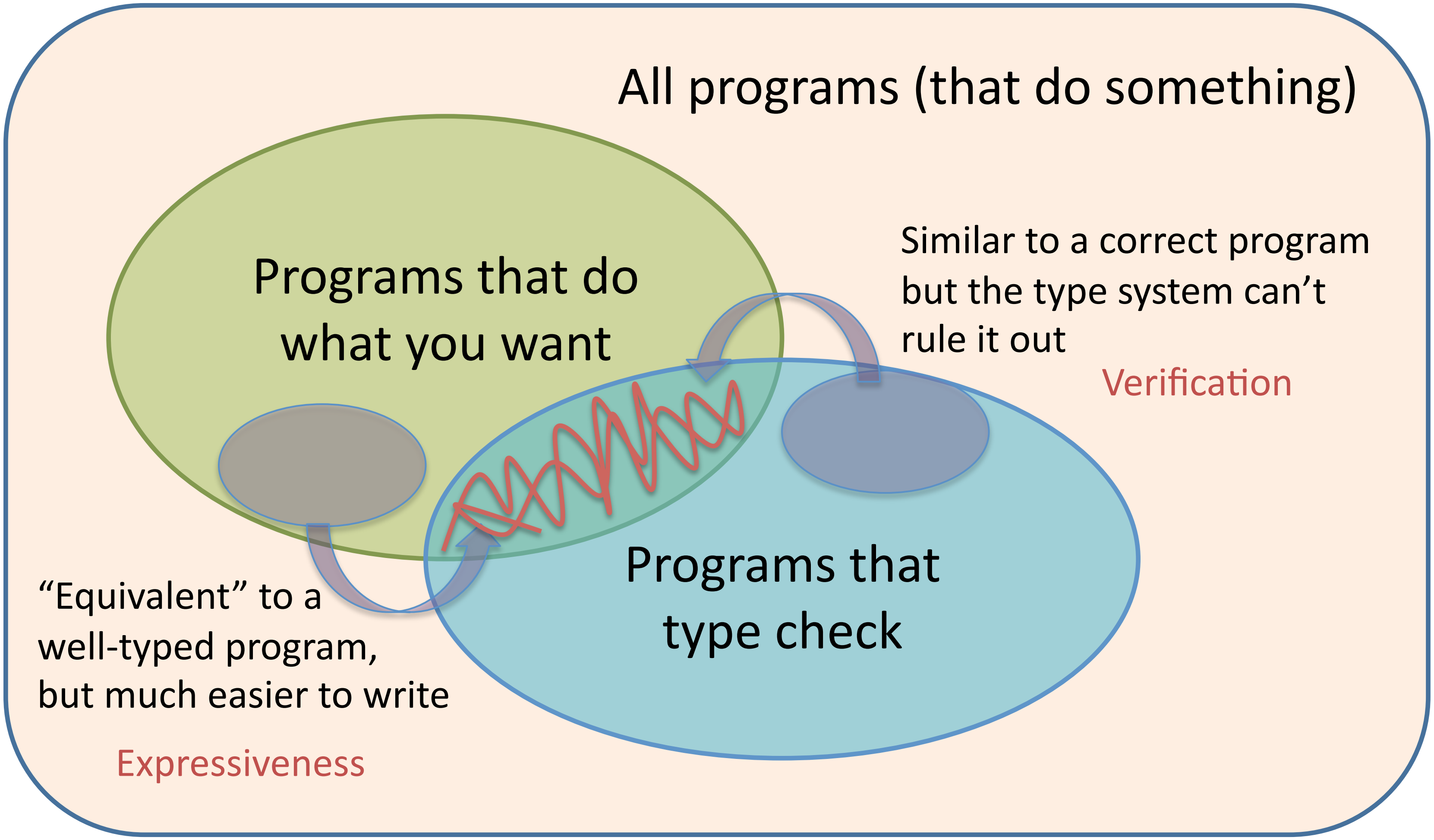
Talk Plan

- Part 1: What are dependent types all about?
  - Examples of and motivation for dependently-typed programming
  - Overview of current research topics, projects and directions
- 10-min Break

Please ask questions!

- Part 2: How do you implement a type checker for a dependently-typed language?
  - Play along: https://github.com/sweirich/pi-forall
  - Caveat: pi-forall is missing many important features

Type systems Research



Why Dependent Types?

- *Verification*: Dependent types express application-specific program invariants that are beyond the scope of existing type systems
- *Expressiveness*: Dependent types enable flexible interfaces, of particular importance to generic programming and metaprogramming.
- *Uniformity*: The same syntax and semantics is used for computations, specifications and proofs

Program verification is “just programming”

Dependent types and verification

- Haskell prelude function, only defined for non-empty lists:

```
head :: [a] -> a
head (x : xs) = x
head [] = error “no head”
```

- Is “`head z`” a correct program? Haskell’s type checker can’t tell.

With dependent types (pi-forall)

- Datatype that tracks the length of the list at compile time

```
data Nat : Type where
   Zero
   Succ of Nat
```

Indexed datatype

```
data Vec (A : Type) (n : Nat) : Type where
   Nil  of [n = Zero]
   Cons of [m:Nat] (A) (Vec A m) [n = Succ m]
```

```
head : [A : Type] -> [n:Nat] -> Vec A (Succ n) -> A
head = \ [A][n] x.  case x of
     Cons [m] y ys -> y
    -- Nil case is impossible, because Zero /= Succ n
```

If "`head z`" typechecks, then z must be non-nil.

Extended example

Lambda.pi

What are the research problems in designing dependently-typed languages?

Effective program development

- How can we make it easier to create and work with dependently-typed programs?
  - Specifications and proofs can be long... sometimes longer than the programs themselves
- Research directions:
  - Embedded domain-specific languages
  - Tactics (special purpose language to generate programs)
  - Type/proof inference
  - Theorem provers (SMT solvers, etc.)
  - IDE support: view development as interactive
  - Integration with testing

Efficient Compilation

- Consider this function:

```
safe_head :(x:List a) -> non_empty x -> a
safe_head (Cons x xs) _ = x
```

Proof argument

- How do we divide computational arguments from specificational arguments?
  - Idris/Epigram – let the compiler figure it out
  - GHC (and many others) – syntactically distinguish them
  - Coq – type system sort distinction (Prop / Set)
  - pi-forall, ICC* (and others) – type system marks irrelevance

Non-termination

- Consistency proofs for logic require all programs to terminate
- Programmers don't
- What to do?
  - Require proofs to be values (Haskell, Cayenne)
  - Nontermination monad (model infinite computation via coinduction)
  - Partial type theories (Nupri)
- Zombie research language
  Chris Casinghino, Vilhelm Sjöberg,
  Stephanie Weirich

Semantics

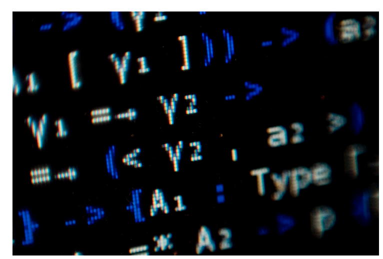
- Type checking requires deciding type equality....

  ....and types contain programs
- When are two programs equal?
  - When they are beta equal? (\x.x) 3 = 3
  - When they are beta/eta equal? (\x y. plus x y) = plus
  - When their relevant parts are equal?
  - When they are provably equal?
  - When they are both proofs of the same thing?
    p1, p2 : A = B implies p1 = p2
  - Univalence: still more yet...
- Many other semantic issues
  - Predicativity vs. Impredicativity
  - Inductive datatypes & termination

How to get started?

Pick a language and play with it

- **Agda:** See wiki for tutorials, invited talk from ICFP 2012 (McBride)
- **Coq:** *Certified Programming with Dependent Types* (Chlipala) *Software Foundations* (Pierce et al.)
- **Idris:** Tutorials and videos at http://www.idris-lang.org/ (Brady)
- **F-star:** Security-focus, compiles to Javascript and F# (Swamy et al.)
- **GHC:** Singletons (Eisenberg & Weirich) and Hasochism (Lindley & McBride)



Implement your own language!

- We are still learning about the role of dependent types in programming
  - There is plenty still to learn by experimenting!

- Don’t have to start from scratch
  - Löh, McBride, Swierstra. “A Tutorial Implementation of a Dependently Typed Lambda Calculus.” *Fundamenta Informaticae,* 2001
  - Andrej Bauer, “How to implement dependent type theory”
  - Lectures on implementing Idris (www.idris-lang.org)
  - 2nd half of talk: pi-forall

More examples

GHC: Flow sensitive typing

```
data Expr a where
   CB    :: Bool -> Expr Bool
   CI    :: Int -> Expr Int
   If    :: Expr Bool -> Expr a -> Expr a -> Expr a
   BinOp :: Op (a -> b -> c)
          -> Expr a -> Expr b -> Expr c
```

```
eval :: Expr a -> a
eval (CB b) = b
eval (CB i) = i
eval (If x y z) =
   if eval x then eval y else eval z
eval (Binop b x y) =
   b (eval x) (eval y)
```

In each branch, the data constructor determines 'a'

Indexed datatypes encode proofs

```
Inductive is_redblack : tree → color → nat →
  Prop :=
| IsRB leaf: ∀c, is_redblack E c 0
| IsRB_r: ∀tl k tr n,
    is_redblack tl Red n →
    is_redblack tr Red n →
    is_redblack (T Red tl k tr) Black n
| IsRB_b: ∀c tl k tr n,
    is_redblack tl Black n →
    is_redblack tr Black n →
    is_redblack (T Black tl k tr) c (S n)
```

Red nodes must have Black parents

Black nodes can have arbitrary parents

Appel. “Efficient Verified Red-Black Trees.” 2011