Why should you care about dependent types?

Stephanie Weirich

University of Pennsylvania

PLMW 2014

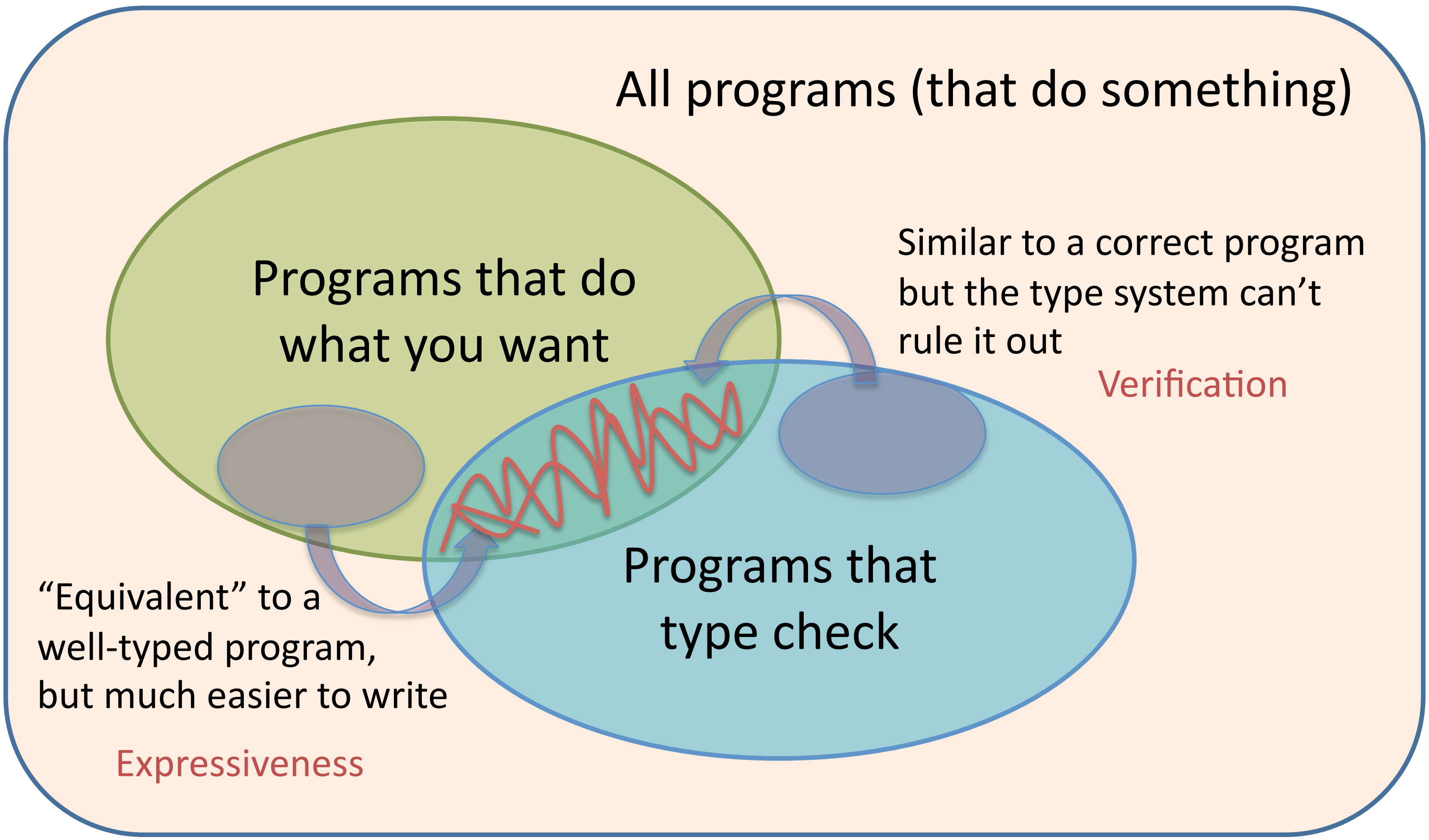
Why I care about dependent types

Stephanie Weirich

University of Pennsylvania

PLMW 2014

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 list -> 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

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

Indexed datatype

```
data Nat = O | S Nat
data Vec (a : *) (n : Nat) where
  nil  : Vec a O
  cons : a -> Vec a n -> Vec a (S n)

head :: Π (x:Nat). Vec a (S x) -> a
head (cons x xs) = x
-- head nil case impossible!
```

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

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

Expressiveness

- What about programs that do what you want, but don’t type check?

- Generic programming
  - Types can be calculated by programs
  - Program execution can depend on types

- Embedded Domain-Specific Languages
  - Application-specific type checking
  - Building a programming language is hard!
  - Dependently-typed *meta*-languages

Generic Programming

- User-defined generic traversals
  - Operations defined over representations of the type structure, in a type-preserving way
  - Eliminates boilerplate code. Aids development & refactoring
- Examples:

```
children (BinOp Plus e1 e2) == [e1; e2]
freshen (If (Var "x") (Var "y") (Var "z")) ==
   (If (Var "x0") (Var "y0") (Var "z0"))
```

`arbitrary / shrink` for random test generation

Embedded Domain Specific Languages

- EFFECT, embedded in Idris
  - algebraic model of effects in types
  - alternative to Monad Transformers
  - extensible to new effects
- Bedrock & VST, embedded in Coq
  - low-level safe C-like language
    for safe systems programming
  - tactics for generating proofs about
    memory safety

- Ivory, embedded in Haskell
  - low-level safe C-like language
    for safe systems programming
  - generates C, linked with RTOS and loaded onto quadcopter

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 (cf. Ulf Norell ICFP 2013)
  - Incremental development
    - Once you have stated a program property, why not use it for testing first?

Efficient Compilation

- Consider this function:

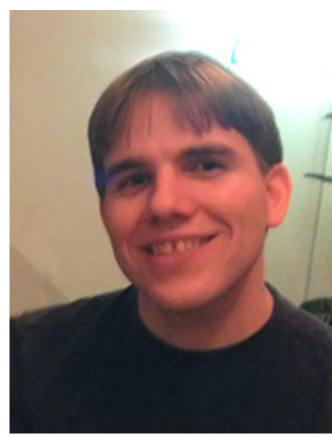
```
safe_head : Π(x:list a). non_empty x -> a
safe_head (cons x xs) _ = x
```

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)
  - Trellys, ICC* (and others) – type system analysis

Non-termination

- Consistency proofs for logic require all programs to terminate
- Programmers don't
- What to do?
  - Require proofs to be values
  - Nontermination monad (model infinite computation via coinduction)
  - Partial type theories (Nuprl, Zombie)
- Chris Casinghino, Vilhelm Sjöberg, Stephanie Weirich. "Combining Proofs and Programming in a Dependently-Typed Language", Session 1a tomorrow



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
  - (See Richard's talk on Closed Type Families, Friday)
  - When they are beta/eta equal? (\x y. plus x y) = plus
  - When they are both proofs of the same thing?

    p1, p2 : A = B implies p1 = p2
  - When their relevant parts are equal?
  - Univalence: still more yet…

- Many other semantic issues
  - Predicativity vs. Impredicativity
  - Inductive datatypes & termination

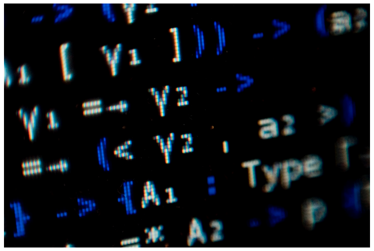
How to get started?

Reading List

- Per Martin Löf. *Constructive mathematics and computer programming*, 1982
- Nordstrom, Petersson, and Smith. *Programming in Martin-Löf's Type Theory,* 1990
- Barendregt. "*Lambda Calculi with Types.*" Handbook of Logic in Computer Science II, 1992
- Harper, Honsell, Plotkin. *"A Framework for Defining Logics."* *JACM* 1993
- Aspinall and Hoffman. "Dependent types." *ATTAPL*, 2004
- Sørensen and Urzyczyn*, Lectures on the Curry-Howard Isomorphism*, 2006
- *Homotopy Type Theory: Univalent Foundations of Mathematics*, 2013

Pick a language and play with it

- **Agda:** See wiki for tutorials, watch invited talks from ICFP 2012 (McBride) & 2013 (Norell)
- **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
  - Lectures on implementing Idris (www.idris-lang.org)
  - My OPLSS 2013 lectures & pi-forall github repository