Doing dependent types wrong without going wrong

Stephanie Weirich, University of Pennsylvania
Work in Progress with Limin Jia, Jianzhou Zhao, and Vilhelm Sjöberg

What are dependent types?

Types that depend on elements of other types.

- Examples:
  - vec n – type of lists of length in
  - Generalized tries
  - PADS
  - Type of ASTs that represent well-typed code
- Statically enforce expressive program properties
  - BST ops preserve BST invariants
  - CompCert compiler

Two sorts

Full Spectrum	Phase-sensitive
Types indexed by actual computations	Types indexed by a pure language, separate from computations
Easier to connect type system to actual computation, harder to extend computation language	Index language may have minimal similarity to computation language
Includes "strong eliminators" if x=3 then Bool else Int	May or may not not include strong eliminators
Examples: Cayenne, Coq, Epigram, Agda2, Guru	Examples: DML, ATS, Ω mega, Haskell

Let's do it wrong…

- Cayenne is *only* language that deliberately allows nonterminating terms in types
  - Nothing proved about it!
- Primary Goal: prove *type soundness* for a language with impure computations in types.
  - Note: type checking may be undecidable
- Secondary Goals:
  - CBV language
  - "Modular" metatheory

Full spectrum: Pure type system

- No distinction between types and terms

```
s,t,A,B,k ::=  x | \x.t | s t | (x:A) -> B | T
             | * | [] | c | case s { c x => t }
```

- One set of formation rules

$$\Gamma \ \vdash t : A$$

- Conversion rule uses type equivalence

$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash B : s \qquad A \sim B}{\Gamma \vdash t : B}$$

A and B are beta-convertible

- Term equivalence is fixed by type system (and defined to be the same as type equivalence).

New vision

- Syntactic distinction between terms and types, but still full spectrum

```
k ::=  * | (x:A) -> k
A ::= (x:A1) -> A2 | T | A w | let x = t in A
   | case t of { c x => A }
t ::= x | \x. t | t w | let x = t in t
   | c w | case t of { c x => t }
   | fix f(x). t
w ::= x | \x. t | fix f(x). t | c w
```

- Key changes:
  - Term language explicitly includes non-termination
  - CBV – only pure terms (w) substituted for variables
  - Type system parameterized by term equality

Parameterized term equality

- Given a list of equality assumptions about terms:
  - `Δ ::= . | Δ , t1 = t2`
- Assume the existence of two functions:
  - `con (Δ) in { maybe, false }`
  - `isEq (Δ, t1, t2)  in { true, maybe }`

- Equality is untyped
  - No guarantee that `t1` and `t2` have the same type
  - No assumptions about the types of the free variables
  - Types don't require terms appearing in them to be well-typed

Type equivalence (excerpt)

$$\frac{\texttt{con}\ (\Delta) = \texttt{false}}{\Delta \mid\!- \texttt{A1} = \texttt{A2}}$$

$$\frac{\Delta \mid\!- \texttt{A1} = \texttt{A2} \qquad \texttt{isEq}\ (\Delta,\ \texttt{w1 w2}) = \texttt{true}}{\Delta \mid\!- \texttt{A1 w1} = \texttt{A2 w2}}$$

$$\frac{\texttt{isEq}\ (\Delta,\ \texttt{t},\ \texttt{ci wi}) = \texttt{true}}{\Delta \mid\!- \texttt{case t of}\ \{\ \texttt{ci xi} \Rightarrow \texttt{Ai}\ \} = \texttt{Ai}\ \{\ \texttt{wi} / \texttt{xi}\ \}}$$

$$\frac{\Delta, x = t \mid\!- A = B \qquad x \text{ notin } \Delta, B}{\Delta \mid\!- \texttt{let x = t in A = B}}$$

Typing rules (excerpt)

```
 Γ Δ |- t : (x:A) -> B     Γ Δ |- w : A
-----------------------------------------
        Γ Δ |- t w : B { w / x }

 Γ Δ |- t1 : A   Γ,x:A  Δ,x=t1 |- t2 : B
-----------------------------------------
      Γ Δ |- let x = t1 in t2 : B

      Γ Δ |- t : T t'     Δ |- B : *
         ci : (xi : Ai) -> T ti'
Γ, xi:Ai  Δ, t = ci xi, ti' = t' |- ti : B
-----------------------------------------
  Γ Δ |- case t of { ci xi => ti } : B

Γ Δ |- t : A    Δ |- A = B      Δ |- B : *
-------------------------------------------
              Γ Δ |- t : B
```

Questions to answer

- What properties of `isEq` & `Con` must we assume to show preservation & progress?
- What instantiations of `isEq` & `Con` satisfy these properties?

Necessary assumptions (con)

- Don't start inconsistent

  `con( . ) = maybe`

- Once inconsistent, stay inconsistent through weakening, substitution, cut and conversion

  - `con (Δ) = false => con (Δ Δ') = false`
  - `con (Δ) = false => con (Δ {w/x} ) = false`
  - `con (Δ (e1 = e2) Δ') = false & isEq (Δ, e1, e2) => con (Δ Δ') = false`
  - `con(Δ) = false & (Δ = Δ') => con(Δ') = false`

Necessary assumptions (isEq)

- `isEq` is an equivalence class
- Holds for evaluation: If `e -> e'` then `isEq (Δ, e, e')`
- Constructors are injective, for (possibly) consistent contexts
  `con(Δ) = maybe & isEq(Δ, ci e1, cj e2) =>`
  `isEq(Δ, e1, e2) & i=j`
- Preserved by substitution
  `isEq(ΔΔ', e1, e2) => isEq(Δ, w, w') =>`
  `isEq (ΔΔ'{w/x}, e1{w/x}, e2{w'/x})`
- Preserved under contextual operations (weakening, cut, conversion)
  `isEq (Δ (e = e') Δ', e1, e2)  & isEq(Δ, e, e') =>`
  `isEq (Δ Δ',  e1, e2)`

What satisfies these properties?

- Compare normal forms, ignoring equalities in the context
  - Above plus equalities in the context
- Contextual equivalence
  - Contextual equivalence modulo Δ
- Some strange equalities that identify nonterminating terms with terminating terms
  - Sound to conclude isEq(let x = loop in 3, 3) as long as we don't conclude isEq(let x = loop in 3, loop)
  - Sound to say isEq(loop,3) as long as we don't say isEq(loop, 4)

What about termination?

- Termination analysis not required for type soundness
  - Decidable approximation of `isEq` is type sound, but doesn't satisfy preservation
    - Any types system that checks strictly fewer terms than a sound type system is sound.
- However, like most type systems, only get partial correctness results:
  - "If this expression terminates, then it produces a value of type t"
- Termination analysis permits proof erasure

More questions

- Is untyped equivalence strong enough?
  - Have we accomplished anything?
- Can we give more information about typing to `Con` and `isEq`?
  - For now, we want to make axiomatization of `isEq` independent of the type system, but does that buy us anything?
- Can we add a predicate to control what expressions are compared for equality?
  - Limit domain of isEq for stronger properties
- What about more computational effects: state/control effects?
  - Can we use effect typing to strengthen equivalence?