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

Conversion rule uses type equivalence

A and B are betaconvertible

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

- 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:

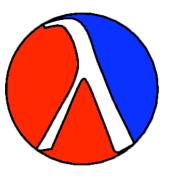
```
\triangle ::= . | \triangle , t1 = t2
```

Assume the existence of two functions:

- \triangleright con (\triangle) 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)

con
$$(\Delta)$$
 = false
 Δ I - A1 = A2

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

isEq (
$$\Delta$$
, t, ci wi) = true
 Δ |- case t of { ci xi => Ai } = Ai { wi / xi }

$$\Delta$$
, x = t |- A = B x notin Δ , B
 Δ |- let x = t in A = B

Typing rules (excerpt)

```
\Gamma \Delta I - t : (x:A) \rightarrow B \qquad \Gamma \Delta I - w : A
            \Gamma \Delta I - tw : B \{ w / x \}
 \Gamma \Delta I - t1 : A \Gamma, x : A \Delta, x = t1 I - t2 : B
          \Gamma \Delta I - let x = t1 in t2 : B
         ΓΔ l- t : T t' Δ l- B : *
             ci : (xi : Ai) -> T ti'
\Gamma, xi:Ai \Delta, t = ci xi, ti' = t' |- ti : B
    \Gamma \Delta I - case t of { ci xi => ti } : B
 \Gamma \Delta I - t : A \Delta I - A = B \Delta I - B : *
                    ΓΔ I- 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 (\Delta) = false => con (\Delta \Delta') = false
```

- con (Δ) = false => con $(\Delta \{w/x\})$ = false
- con (Δ (e1 = e2) Δ ') = false & isEq (Δ , e1, e2) => con (Δ Δ ') = false
- \cdot con(Δ) = false & ($\Delta = \Delta$ ') => con(Δ ') = false

Necessary assumptions (isEq)

- isEq is an equivalence class
- ▶ Holds for evaluation: If $e \rightarrow e'$ then is Eq (Δ , e, e')
- Constructors are injective, for (possibly) consistent contexts

```
con(\Delta) = maybe \& isEq(\Delta, ci e1, cj e2) => isEq(\Delta, e1, e2) \& i=j
```

Preserved by substitution

```
isEq(\Delta\Delta', e1, e2) \Rightarrow isEq(\Delta, w, w') \Rightarrow isEq(\Delta\Delta'\{w/x\}, e1\{w/x\}, e2\{w'/x\})
```

 Preserved under contextual operations (weakening, cut, conversion)

```
isEq (\Delta (e = e') \Delta', e1, e2) & isEq(\Delta, e, e') => isEq (\Delta \Delta', e1, e2)
```

What satisfies these properties?

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