Adventures in Dependently-Typed Metatheory

Work in Progress

Joint work 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
  - vec n m – type of n x m matrices
  - Type of trees that satisfy binary search tree invariant
  - Type of ASTs that represent well-typed code
- Statically enforce expressive program properties
  - BST ops preserve BST invariants
  - tagless, staged interpreters
  - CompCert compiler

Dependent types today

- Umbrella term for many languages that permit expressive static checking

Full Spectrum	Phase-sensitive
Types indexed by actual computations	Types indexed by a pure language, separate from computations
Type checking involves deciding program equivalence	Easier to decide type equality, as only pure expressions are involved
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 not include strong eliminators
Examples: Cayenne, Coq, Epigram, Agda2, Guru	Examples: DML, ATS, Ωmega, Haskell

Full spectrum: Lambda Cube

- One syntactic class, no distinction between types and terms

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

- One set of formation rules:

```
G |- t : A
```

- Conversion rule to decide type equivalence

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

Full-spectrum types

- Problem: full-spectrum type systems do not interact well with full programming languages
  - Single definition of equivalence for types and terms
- Type soundness depends on properties of equivalence that be must proven early in the development
  - Need to know int != bool
- Additions to the programming language requires significant restructuring of the definition of equivalence (fix, state, effects, etc.)

New vision

- Syntactic distinction between terms and types (computations)
- Still full spectrum, types depend on computation

```
k    ::=  * | (x:A) -> k

A, B ::= (x:A) -> B | T | A t

      |  case t of {c x => A}

t    ::= x | \x. t |  t u | c

      | case t of {c x => t} | fix x:A . t
```

Key changes:

term language explicitly includes non-termination

different definitions of equality for types and terms

Parameterized term equality

Given a list of equality assumptions about terms:

```
D ::= . |  D (t1 = t2)
```

Assume the existence of two (partial) functions:

```
con (D) in { true, maybe, false }

isEq (D, t1, t2)  in { true, maybe }
```

Type equivalence depends on parameterized term equivalence

$$\frac{\texttt{con (G*) = false}}{\texttt{G |- t1 = t2 : k}}$$

$$\frac{\texttt{G |- A1 = A2 : (x:B) -> k} \qquad \texttt{isEq (G*, t1 t2)=true}}{\texttt{G |- A1 t1 = A2 t2 : k}}$$

$$\frac{\texttt{isEq (G*, t, c t1)=true} \qquad \texttt{G, t = c t1 |- t2 : k}}{\texttt{G |- case t of \{ c x => t2 \} = t2 \{ t1 / x \} : k}}$$

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)

- Start consistent
  `con( . ) = true`

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

  - `con (D) = false => con (D D’) = false`

  - `con (D) = false => con (D {e/x} ) = false`

  - `con (D (e1 = e2) D’) = false & isEq (D, e1, e2) =>`
    `con (D D’) = false`

  - `con(D) = false & (D = D’) => con(D’) = false`

Necessary assumptions (isEq)

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

What satisfies these properties?

- Trivial equality that only compares normal forms, ignoring equalities in the context
  - This is the weakest (finest) equality that satisfies the assumptions
- Above plus equalities in the context
- Version that erases "irrelevant" information before normalization
- Coarser equalities that identify more terms, cf. contextual equivalence

What about termination?

- Termination analysis not required for type soundness
  - Decidable version of `isEq` is type sound, but doesn't satisfy preservation
  - Progress requires CBV semantics
- 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
  - Otherwise, must run proofs to make sure they are not bogus

More questions

- 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?
- Useful to add `inDom` predicate to control what expressions are compared for equality?
- What about more computational effects: state/control effects?
  - Can we use effect typing to strengthen equivalence?

Conclusion

- Metatheory for full-spectrum dependently-typed languages is complex, highly entangled
  - Canonical forms lemmas require deep reasoning about program equivalence
  - Our current definitions are algorithmic to permit inversion lemmas
- Parameterizing term equality allows us to reuse results
  - Don't fix decision procedure for program equivalence a priori
  - The fundamental structure of the type soundness proof shouldn't change when new features are added to the computation language