Adventures in Dependentely-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

<table>
<thead>
<tr>
<th>Full Spectrum</th>
<th>Phase-sensitive</th>
</tr>
</thead>
<tbody>
<tr>
<td>Types indexed by actual computations</td>
<td>Types indexed by a pure language, separate from computations</td>
</tr>
<tr>
<td>Type checking involves deciding program equivalence</td>
<td>Easier to decide type equality, as only pure expressions are involved</td>
</tr>
<tr>
<td>Easier to connect type system to actual computation, harder to extend computation language</td>
<td>Index language may have minimal similarity to computation language</td>
</tr>
<tr>
<td>Includes &quot;strong eliminators&quot; if x=3 then Bool else Int</td>
<td>May not include strong eliminators</td>
</tr>
<tr>
<td>Examples: Cayenne, Coq, Epigram, Agda2, Guru</td>
<td>Examples: DML, ATS, Ωmega, Haskell</td>
</tr>
</tbody>
</table>
Full spectrum: Lambda Cube

- One syntactic class, no distinction between types and terms
  \[ s, t, A, B, k \ ::= \ x \mid \lambda x. t \mid s \ t \mid (x : A) \rightarrow B \]
  \[ \mid * \mid [] \mid c \mid \text{case } s \{ c \ x \Rightarrow t \} \]

- One set of formation rules:
  \[ G \vdash t : A \]

- Conversion rule to decide type equivalence
  \[ G \vdash t : A \quad G \vdash B : s \quad A \sim B \]
  \[ G \vdash t : B \]
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 must be proven early in the development
  - Need to know \( \text{int} \neq \text{bool} \)

- 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

\[
k ::= * \mid (x:A) \rightarrow k
\]

\[
A, B ::= (x:A) \rightarrow B \mid T \mid A \ t
\]

\[
| \ case \ t \ of \ \{ c \ x =\Rightarrow \ A \}
\]

\[
t ::= x \mid \backslash x. \ t \mid t \ u \mid c
\]

\[
| \ case \ t \ of \ \{ c \ x =\Rightarrow \ t \} \mid \text{fix} \ x:A . \ t
\]

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:

\[
D ::= . \mid D \ (t_1 = t_2)
\]

Assume the existence of two (partial) functions:

\[
\text{con} \ (D) \ \text{in} \ \{ \text{true, maybe, false} \}
\]
\[
\text{isEq} \ (D, t_1, t_2) \ \text{in} \ \{ \text{true, maybe} \}
\]
Type equivalence depends on parameterized term equivalence

\[
\text{con } (G^*) = \text{false} \\
G \vdash t_1 = t_2 : k
\]

\[
G \vdash A_1 = A_2 : (x:B) \rightarrow k \quad \text{isEq } (G^*, t_1 \ t_2)=\text{true} \\
G \vdash A_1 \ t_1 = A_2 \ t_2 : k
\]

\[
\text{isEq } (G^*, t, c \ t_1)=\text{true} \quad G, t = c \ t_1 \vdash t_2 : k \\
G \vdash \text{case } t \text{ of } \{ \ c \ x \Rightarrow t_2 \ \} = t_2 \ \{ \ t_1 / x \ \} : k
\]
Questions to answer

- What properties of \texttt{isEq/Con} must we assume to show preservation & progress?
- What instantiations of \texttt{isEq/Con} satisfy these properties?
Necessary assumptions (con)

- Start consistent
  \( \text{con}( . ) = \text{true} \)

- Once inconsistent, stay inconsistent through weakening, substitution, cut and conversion
  - \( \text{con} (D) = \text{false} \Rightarrow \text{con} (D \ D') = \text{false} \)
  - \( \text{con} (D) = \text{false} \Rightarrow \text{con} (D \ {e/x} ) = \text{false} \)
  - \( \text{con} (D \ (e1 = e2) \ D') = \text{false} \ & \ \text{isEq} (D, e1, e2) \Rightarrow \text{con} (D \ D') = \text{false} \)
  - \( \text{con}(D) = \text{false} \ & \ (D = D') \Rightarrow \text{con}(D') = \text{false} \)
Necessary assumptions (isEq)

- **isEq** is an equivalence class

- Holds for evaluation: If \( e \rightarrow e' \) then isEq \((D, e, e')\)

- Constructors are injective, for (possibly) consistent contexts
  \[
  \text{con}(D) \neq \text{false} \& \text{isEq}(D, c_i e_1, c_j e_2) \Rightarrow \text{isEq}(D, e_1, e_2) \& i=j
  \]

- Preserved by substitution
  \[
  \text{isEq}(D, e_1, e_2) \Rightarrow \text{isEq}(D\{e/x\}, e_1\{e/x\}, e_2\{e/x\})
  \]

- Preserved under contextual operations (weakening, cut, conversion)
  \[
  \text{isEq}(D(\text{e = e'}) D', e_1, e_2) \& \text{isEq}(D, e, e') \Rightarrow \text{isEq}(D, D', e_1, e_2)
  \]
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 $\text{isEq}$ 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 \texttt{Con} and \texttt{isEq}?
  - For now, we want to make axiomatization of \texttt{isEq} independent of the type system, but does that buy us anything?

- Useful to add \texttt{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.