Dependent types and program equivalence

Stephanie Weirich, University of Pennsylvania
with Limin Jia, Jianzhou Zhao, and Vilhelm Sjöberg
What are dependent types?

- Types that depend on values of other types
- Used to statically enforce expressive program properties
- Examples:
  - `vec n` – type of lists of length n, static bounds checks
  - Binary Search Tree
  - PADS, data format invariants
  - ASTs that enforce well-typed code
  - CompCert compiler
What about nontermination?

- Treatment of nontermination divides design space
- Affects decidability of type checking, correctness guarantees, and complexity of language
- Independent of type soundness
- Unclear impact on practicality

<table>
<thead>
<tr>
<th>Examples</th>
<th>Only total computation allowed</th>
<th>Types restricted to total computation</th>
<th>No restrictions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coq, Agda2</td>
<td>DML, ATS, Omega, Haskell</td>
<td>Cayenne, Epigram, (\Pi\ \Sigma)</td>
<td></td>
</tr>
<tr>
<td>Type checking</td>
<td>Decidable</td>
<td>Undecidable</td>
<td></td>
</tr>
<tr>
<td>Correctness guarantee</td>
<td>Total correctness</td>
<td>Partial correctness</td>
<td></td>
</tr>
</tbody>
</table>
Program equivalence

- When types depend on programs, type equivalence depends on program equivalence
- Definition of program equivalence is controversial
  - Even when the language is not Turing-complete!
- Many possible definitions
  - Reduce and compare
    - What reduction relation? (evaluation, parallel reduction, eta-reduction?)
  - Type-based equivalence
  - Behavioral equivalence
  - Contextual equivalence
  - Something else?
\(\lambda \approx\): Parameterized program equivalence

- A call-by-value language with an abstract term equivalence relation

Goals for language design
- Simple type soundness proof based on progress and preservation
- Uniformity---program equivalence used by type system must be compatible with CBV

What requirements for equivalence relation?
- Strong enough to prove type soundness
- Weak enough to allow desired definitions

More difficult than we expected
"Pure everywhere" type system - PTS

- No syntactic distinction between types, terms, kinds
  \[ e, \; \tau, \; k \; ::= \; x \mid \lambda x. e \mid e \; e' \mid (x: \tau_1) \to \tau_2 \mid \ast \mid \Box \mid T \mid C \mid \text{case } e \{ \; C_i \; x_i \Rightarrow e_i \; \}\]

- One set of formation rules
  \[ \Gamma \vdash e : \tau \]

- Conversion rule uses beta-equivalence
  \[
  \frac{\Gamma \vdash e : \tau_1 \quad \Gamma \vdash \tau_2 : s \quad \tau_1 \simeq \tau_2}{\Gamma \vdash e : \tau_2}
  \]

- Term equivalence is fixed by type system (and defined to be the same as type equivalence).
\[ \lambda \approx: \text{Parameterized program equivalence} \]

- **Syntactic distinction between terms, types, and kinds**
  
  \[ \begin{align*}
  k & ::= \ast \mid (x:\tau) \rightarrow \ast \\
  \tau & ::= (x:\tau_1) \rightarrow \tau_2 \mid T \mid \tau \ e \mid \text{case } e \langle T \ e' \rangle \text{ of } \{ \ C_i \ x_i \Rightarrow \tau_i \ \}
  \\
  e & ::= x \mid \text{fun } f (x) = e \mid e \ e' \mid C \ e \mid \text{case } e \text{ of } \{ \ C_i \ x_i \Rightarrow e_i \ \}
  \end{align*} \]

- **Key syntactic changes**
  
  - Term language includes non-termination
  - Curry-style, no types in expressions
  - Convenient simplifications
    - Datatypes have one index, data constructors have one argument (unit/products in paper)
    - No polymorphism, no higher-kindred types (future work)
Parameterized term equivalence

- Given an "equivalence context"
  \[ \Delta ::= . \mid \Delta , e_1 = e_2 \]
- Assume the existence of program equivalence predicate
  \[ \text{isEq}(\Delta, e_1, e_2) \]
- Equality is untyped
  - No guarantee that \( e_1 \) and \( e_2 \) have the same type
  - No assumptions about the types of the free variables
- Context may make unsatisfiable assumptions
Type system overview

- Two sorts of judgments
  - Equality for types, contexts, and kinds
    \[ \Delta \vdash \tau_1 \equiv \tau_2 \]
  - Formation for contexts, kinds, types and terms
    \[ \Gamma \vdash e : \tau \]

- Typing context: Equivalence and typing assumptions
  \[ \Gamma ::= . | \Gamma , e_1 = e_2 | \Gamma , x : \tau \]

- All judgments derivable from an inconsistent context
  \[ \text{incon} (\Delta) \text{ if there exist pure terms } C_i w_i \text{ and } C_j w_j \text{ such that } \]
  \[ \text{isEq} (\Delta , C_i w_i , C_j w_j ) \text{ and } C_i \not\equiv C_j \]

- Pure terms
  \[ w ::= x | \textbf{fun} \ f (x) = e | C \ w \]
Type system excerpt

\[
\begin{align*}
\Gamma & \vdash e : \tau & \Gamma^* & \vdash \tau \equiv \tau' & \Gamma & \vdash \tau' : * \\
\hline
\Gamma & \vdash e : \tau'
\end{align*}
\]

\[
\begin{align*}
\Delta & \vdash \tau \equiv \tau' & \text{isEq}(\Delta, e, e') \\
\hline
\Delta & \vdash \tau e \equiv \tau' e'
\end{align*}
\]

\[
\begin{align*}
\text{incon}(\Delta) \\
\hline
\Delta & \vdash \tau \equiv \tau'
\end{align*}
\]

\[
\begin{align*}
\vdash \Gamma & \text{incon}(\Gamma^*) \\
\hline
\Gamma & \vdash e : \tau
\end{align*}
\]
Questions to answer

- What properties of $\text{isEq}$ must hold to show preservation & progress?

- What instantiations of $\text{isEq}$ satisfy these properties?
Necessary assumptions about \texttt{isEq}

- Is an equivalence relation
- Preserved under contextual operations
  - Cut: …
  - Weakening: …
  - Context Conv: …
- Contains evaluation: $e \mapsto e'$ implies isEq ($\Delta$, $e$, $e'$)
- Data constructors are injective for pure arguments
  - isEq ($\Delta$, $C$ $w$, $C$ $w'$) implies isEq ($\Delta$, $w$, $w'$)
- Empty context is consistent
  - $C \neq C'$ implies $\neg$ isEq (., $C$ $w$, $C'$ $w'$)
- Closed under \texttt{pure} substitution
  - isEq ($\Delta$, $e$, $e'$) implies isEq ($\Delta$\{w/x\}, $e$\{w/x\}, $e'$\{w/x\})
  - Preservation of beta
  - Preservation of \texttt{Nat} \equiv \texttt{Bool}
  - Transitivity of $\Delta \vdash \tau_1 \equiv \tau_2$
  - Does not need to hold for arbitrary $e$
Typing rules don't use substitution

**Standard rule**

\[ \Gamma \vdash e_1 : (x : \tau_1) \rightarrow \tau_2 \]

\[ \Gamma \vdash e_2 : \tau_1 \]

\[ \Gamma \vdash e_1 e_2 : \tau_2 \{ e_2/x \} \]

**Our rule**

\[ \Gamma \vdash e_1 : (x : \tau_1) \rightarrow \tau_2 \]

\[ \Gamma \vdash e_2 : \tau_1 \]

\[ \Gamma^* \equiv x \equiv e_2 \vdash \tau_2 \equiv \tau \]

\[ \Gamma \vdash \tau : * \]

\[ \Gamma \vdash e_1 e_2 : \tau \]

- Substitutes an arbitrary expression into the type
- Adds assumption to the context
- x does not escape
Assumptions also for case expression

- Do not need a substitution to type the branches
What satisfies the isEq properties?

- Compare normal forms (ignoring $\Delta$)
  - Only types STLC terms
- Contextual equivalence (ignoring $\Delta$)
  - Only types STLC terms
- RST-closure of evaluation, constructor injectivity, and equivalence assumptions
- CBV Contextual equivalence modulo $\Delta$
- Some strange equalities that identify nonterminating terms with terminating terms
  - Safe to conclude isEq(let $x = \text{loop}$ in 3, 3) as long as we don’t conclude isEq(let $x = \text{loop}$ in 3, loop)
  - Safe to say isEq($\text{loop}$, 3) as long as we don’t say isEq($\text{loop}$, 4)
What about decidable type checking?

- All instantiations of isEq are undecidable
  - Must contain evaluation relation
- Decidable approximations are type safe, but don’t satisfy preservation
  - Any types system that checks strictly fewer terms than a safe type system is safe
- Preservation important for compiler transformations
  - Want to know that inlining always produces safe code
  - Not really an issue: Decidable doesn't mean tractable
What about termination analysis?

- Like most type systems, only get "partial correctness" results:
  - $\Sigma x:t. \ P(x)$ means "If this expression terminates, then it produces a value of type $t$ such that $P$ holds"
  - Implications ($P_1 \rightarrow P_2$) may be bogus
- Termination analysis produces total correctness
- Termination/stage analysis is an optimization
  - permits proof erasure in CBV language
Future work

- Add polymorphism, higher-order types
  - Keep curry-style system for simple specification of isEq
- Annotated external language to aid type checking
  - Similar to ICC* [Barras and Bernardo]
  - Terms contain type annotations, but equality defined for erased terms
  - Type checking still undecidable but closer to an algorithm
- Add control/state effects to computations
  - Should we limit domain of isEq?
  - Non-termination ok in types, but exceptions are not?
- Can we provide type/termination information to strengthen equivalence?
Conclusions – What have we achieved?

- Uniform design
  - Same reasoning for compile time as run time
  - Not easy for CBV!

- Simple design
  - Program equivalence isolated from type system
  - Proved all metatheory in Coq in ~2 weeks (OTT + LNgen)

- General design
  - Program equivalence not nailed down
  - Lots of examples that satisfy preservation, not just type soundness
Type equivalence for case

\[
\text{isEq}(\Delta, e, C_j w) \quad C_j \in \overline{C_i}^{i \in 1..n} \\
C_j : (x_j : \sigma_j) \rightarrow T \quad u_j \in \Sigma_0 \\
\text{isEq}\left(\left(\Delta, w \equiv x_j\right), u, u_j\right) \\
\Delta, w \equiv x_j, e \equiv C_j \ x_j \vdash \tau_j \equiv \tau \\
\Delta \vdash \text{case } e \langle T \ u \rangle \ of \ \{ \overline{C_i \ x_i \Rightarrow \tau_i}^{i \in 1..n} \} \equiv \tau
\]