Dependent types and program equivalence

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

Doing dependent types wrong without going wrong

Stephanie Weirich, University of Pennsylvania
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 in practice today

Pure everywhere	Pure types only
Types indexed by actual computations, everything is pure (terminating)	Types indexed by a pure language, separate from impure computations
• Decidable type checking • Easy to connect type system to actual computation • Uniform reasoning independent of phase • Total correctness	• Decidable type checking • Expressive computation language, including nontermination, state & control effects, etc
• Not really a programming language	• Index language may have minimal similarity to computation language, both in syntax and semantics • "Partial" Correctness
Examples: Coq, Epigram, Agda2	Examples: DML, ATS, Ωmega, Haskell

Let’s do it wrong…

- What about languages that are impure everywhere?
  - Deliberately allow nonterminating terms in types
  - Type:Type [Cardelli 86], Cayenne [Augustsson 98]
- What does a *type soundness proof* for such a language look like?
  - Note: type checking undecidable
- Advantages
  - Linguistic uniformity, reasoning does not depend on phase
  - Programming language, not a logic
- Disadvantages
  - How to type check?
  - Partial correctness

What else do we want?

- Syntactic type soundness proof
  - Easily extensible
- Strong eliminators
  - "If x = true then int else bool"
  - Important for expressivity, refinements, etc.
- Call-by-value language
  - If we have an impure language, we must fix the evaluation order
  - CBV has better treatment of control effects
- "Modular" metatheory
  - Program equivalence is *hard*. Let's not commit to a particular definition.

"Pure everywhere" type system - PTS

- No distinction between types, terms, kinds

$$e,\ \tau,\ k\ ::=\ x \mid \lambda x.e \mid e\ e' \mid (x{:}\tau_1) \rightarrow \tau_2 \mid * \mid \square$$
$$\mid\ \mathrm{T} \mid C \mid case\ e\ \{\ \overline{C_i\ x_i \Rightarrow e_i}\ \}$$

- One set of formation rules

$$\Gamma \vdash e : \tau$$

- Conversion rule uses beta-equivalence

$$\frac{\Gamma \vdash e : \tau_1 \qquad \Gamma \vdash \tau_2 : s \qquad \tau_1 \sim \tau_2}{\Gamma \vdash e : \tau_2}$$

τ_1 and τ_2 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, types, and kinds

$$k ::= * \mid (x{:}\tau) \rightarrow *$$

$$\tau ::= (x{:}\tau_1) \rightarrow \tau_2 \mid T \mid \tau\ e \mid \text{case } e\ \langle T\ e' \rangle\ \{\ \overline{C_i\ x_i \Rightarrow \tau_i}\ \}$$

$$e ::= x \mid \text{fun } f\,(x) = e \mid e\ e' \mid C\ e \mid \text{case } e\ \{\ \overline{C_i\ x_i \Rightarrow e_i}\ \}$$

- 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-kinded types (future work)

Parameterized term equivalence

- Given an "equivalence context"
  - $\Delta ::= . \mid \Delta , e_1 = e_2$
- Assume the existence of program equivalence predicate
  - $\mathbf{isEq}\ (\Delta, e_1, e_2)$
- Equality is untyped
  - No guarantee that e_1 and e_2 have the same type
  - No assumptions about the types of the free variables
- Rules do not use substitution, add to equivalence context instead

Type system

- Two sorts of judgments
  - Equality for type, contexts, and kinds

$$\Delta \vdash \tau \equiv \tau'$$

  - Formation for contexts, kinds, types and terms

$$\Gamma \vdash e : \tau$$

- All judgments derivable from an inconsistent context
  - **incon** (Δ) if there exist pure terms $C_i\, w_i$ and $C_j\, w_j$ such that **isEq** $(\Delta\,,\ C_i\, w_i,\ \ C_j\, w_j)$ and $C_i \neq C_j$
- Pure terms
  - $w ::= x \mid \text{fun}\, f\,(x) = e \mid C\, w$

Typing rules (excerpt)

Extract equivalence context

$$\frac{\Gamma \vdash e : \tau \quad \Gamma^{\star} \vdash \tau \equiv \tau' \quad \Gamma \vdash \tau' : *}{\Gamma \vdash e : \tau'}$$

$$\frac{\vdash \Gamma \quad \textbf{incon}\,(\Gamma^{\star})}{\Gamma \vdash e : \tau}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : (x{:}\tau_1) \rightarrow \tau_2 \quad \Gamma \vdash e_2 : \tau_1 \\ \Gamma^{\star}, x \cong e_2 \vdash \tau_2 \equiv \tau \quad \Gamma \vdash \tau : *\end{array}}{\Gamma \vdash e_1\, e_2 : \tau}$$

Typing rules for case

$$\frac{\begin{array}{c} C : (x{:}\sigma) \rightarrow T\,u \in \Sigma_0 \quad \Gamma \vdash e : \sigma \\ \Gamma^\star, x \cong e \vdash T\,u \equiv \tau \quad \Gamma \vdash \tau : * \end{array}}{\Gamma \vdash C\,e : \tau}$$

$$\frac{\begin{array}{l} \Gamma \vdash e : T\,u \quad \mathsf{CtrOf}(T) = \overline{C_i}^{\,i \in 1..n} \\ \Gamma \vdash \tau : * \quad \overline{C_i : (x_i{:}\tau_i) \rightarrow T\,u_i \in \Sigma_0}^{\,i \in 1..n} \\ \overline{\Gamma, x_i : \tau_i\,,\, u \cong u_i\,,\, e \cong C_i\,x_i \vdash e_i : \tau}^{\,i \in 1..n} \end{array}}{\Gamma \vdash \textbf{case}\; e\; \textbf{of}\; \{\, \overline{C_i\,x_i \Rightarrow e_i}^{\,i \in 1..n} \,\} : \tau}$$

Type equivalence (excerpt)

$$\frac{\textbf{incon}\,(\,\Delta\,)}{\Delta \vdash \tau \equiv \tau'}$$

$$\frac{\Delta \vdash \tau \equiv \tau' \quad \textbf{isEq}\,(\,\Delta\,,\ e\,,\ e'\,)}{\Delta \vdash \tau\, e \equiv \tau'\, e'}$$

$$\frac{\begin{array}{l}\textbf{isEq}\,(\,\Delta\,,\ e\,,\ C_j\ w\,) \quad C_j \in \overline{C_i}^{\,i\in 1..n} \\ C_j : (x_j{:}\sigma_j) \to T\ u_j \in \Sigma_0 \\ \textbf{isEq}\,(\,(\,\Delta\,,\ w \cong x_j\,)\,,\ u\,,\ u_j\,) \\ \Delta\,,\ w \cong x_j\,,\ e \cong C_j\ x_j \vdash \tau_j \equiv \tau\end{array}}{\Delta \vdash \textbf{case}\, e\, \langle\, T\, u\,\rangle\, \textbf{of}\, \{\, \overline{C_i\ x_i \Rightarrow \tau_i}^{\,i\in 1..n}\,\} \equiv \tau}$$

Questions to answer

- What properties of **isEq** must hold to show preservation & progress?

- What instantiations of **isEq** satisfy these properties?

Necessary assumptions about isEq

- Is an equivalence class
- Contains evaluation: $e \mapsto e'$ implies isEq (Δ, e, e')
- Constructors are injective for pure arguments
  - **isEq** $(\Delta, C\,w, C\,w')$ implies **isEq** (Δ, w, w')
- Empty context is consistent
  - $C \neq C'$ implies **isEq**$(., C\,w, C'\,w')$ does not hold
- Closed under ***pure*** substitution
  - **isEq** (Δ, e, e') implies **isEq** $(\Delta\{w/x\}, e\{w/x\}, e'\{w/x\})$
- Preserved under contextual operations
  - **isEq** $((\Delta, e = e', \Delta'), e_1, e_2)$ and **isEq**(Δ, e, e') implies **isEq** $(\Delta\,\Delta', e_1, e_2)$
  - **isEq** $(\Delta\,\Delta'', e_1, e_2)$ implies **isEq** $(\Delta\,\Delta'\,\Delta'', e_1, e_2)$
  - **isEq** (Δ, e_1, e_2) and $\Delta = \Delta'$ implies **isEq** (Δ', e_1, e_2)

What satisfies these properties?

- Compare normal forms (ignoring Δ)
  - Only types STLC terms
- Contextual equivalence (ignoring Δ)
  - Only types STLC terms
- RST-closure of evaluation, constructor injectivity, and equivalence assumptions
- CBV Contextual equivalence modulo Δ
- Some strange equalities that identify nonterminating terms with terminating terms
  - Safe to conclude isEq(let x = loop in 3, 3) as long as we don't conclude isEq(let x = loop in 3, loop)
  - Safe to say isEq(loop,3) as long as we don't say isEq(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 ($P1 \rightarrow P2$) 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