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

Types that contain computation

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

	Only total computation allowed	**Types restricted to total computation**	**No restrictions**
Examples	Coq, Agda2	DML, ATS, Ωmega, Haskell	Cayenne, Epigram, Π Σ
Type checking	Decidable		Undecidable
Correctness guarantee	Total correctness	Partial correctness	

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?

λ≈: 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

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

- One set of formation rules

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

- Conversion rule uses beta-equivalence

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

> τ_1 and τ_2 are beta-convertible

- Term equivalence is fixed by type system (and defined to be the same as type equivalence).

λ≈: Parameterized program equivalence

- Syntactic distinction between terms, types, and kinds

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

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

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

- 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

- 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
  - $\Gamma ::= . \mid \Gamma \,,\ e_1 = e_2 \mid \Gamma,\ x{:}\tau$

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

Type system excerpt

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

Extract equivalence context

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

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

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

Questions to answer

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

- What instantiations of $\mathbf{isEq}$ satisfy these properties?

Necessary assumptions about **isEq**

- Is an equivalence relation
- Preserved under contextual operations
  - **Cut: …**
  - **Weakening: …**
  - **Context Conv: …**
- Contains evaluation: $e \mapsto e'$ implies $\textbf{isEq}\ (\Delta,\ e,\ e')$
- Data constructors are injective for pure arguments
  - $\textbf{isEq}\ (\Delta,\ C\ w,\ C\ w')$ implies $\textbf{isEq}\ (\Delta,\ w,\ w')$
- Empty context is consistent
  - $C \neq C'$ implies $\neg\textbf{isEq}(\ .\ ,\ C\ w,\ C'\ w')$
- Closed under ***pure*** substitution
  - $\textbf{isEq}\ (\Delta,\ e,\ e')$ implies $\textbf{isEq}\ (\Delta\{w/x\},\ e\{w/x\},\ e'\{w/x\})$

Preservation
$e_1 e_2 \mapsto e_1 e'_2$

Transitivity of
$\Delta \vdash \tau_1 \equiv \tau_2$

$\not\vdash \text{Nat} \equiv \text{Bool}$

Preservation of beta

Does not need to hold for arbitrary e

Typing rules don't use substitution

Standard rule

$$\frac{\Gamma \vdash e_1 : (x : \boldsymbol{\tau}_1) \rightarrow \boldsymbol{\tau}_2 \qquad \Gamma \vdash e_2 : \boldsymbol{\tau}_1}{\Gamma \vdash e_1\, e_2 : \boldsymbol{\tau}_2 \{e_2/x\}}$$

Substitutes an arbitrary expression into the type

Our rule

$$\frac{\Gamma \vdash e_1 : (x{:}\boldsymbol{\tau}_1) \rightarrow \boldsymbol{\tau}_2 \qquad \Gamma \vdash e_2 : \boldsymbol{\tau}_1 \qquad \Gamma^*, x \cong e_2 \vdash \boldsymbol{\tau}_2 \equiv \boldsymbol{\tau} \qquad \Gamma \vdash \boldsymbol{\tau} : *}{\Gamma \vdash e_1\, e_2 : \boldsymbol{\tau}}$$

Adds assumption to the context

x does not escape

Assumptions also for case expression

- Do not need a substitution to type the branches

Type check scrutinee

Lookup data constructors in signature

$$
\frac{
\begin{array}{ll}
\Gamma \vdash e : T\,u & \mathsf{CtrOf}(T) = \overline{C_i}^{\,i \in 1..n} \\
\Gamma \vdash \tau : * & \overline{C_i : (x_i{:}\tau_i) \rightarrow T\,u_i \in \Sigma_0}^{\,i \in 1..n} \\
\multicolumn{2}{l}{\overline{\Gamma, x_i : \tau_i\,,\ u \cong u_i\,,\ e \cong C_i\,x_i \vdash e_i : \tau}^{\,i \in 1..n}}
\end{array}
}{
\Gamma \vdash \mathbf{case}\ e\ \mathbf{of}\ \{\ \overline{C_i\,x_i \Rightarrow e_i}^{\,i \in 1..n}\ \} : \tau
}
$$

Pattern variables don't escape

Data constructor pattern

Data type index

What satisfies the isEq 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:
  - $\Sigma x{:}\mathrm{t}.\ \mathrm{P(x)}$ means “If this expression terminates, then it produces a value of type t such that P holds”
  - Implications $(\mathrm{P1} \rightarrow \mathrm{P2})$ 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

Type equivalence for case

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