Dependent Types and Program Equivalence

Limin Jia   Jianzhou Zhao   Vilhelm Sjöberg   Stephanie Weirich

Computer and Information Sciences Department, University of Pennsylvania
{liminjia,jianzhou,vilhelm,sweirich}@cis.upenn.edu

Abstract

The definition of type equivalence is one of the most important design issues for any typed language. In dependently-typed languages, because terms appear in types, this definition must rely on a definition of term equivalence. In that case, decidability of type checking requires decidability for the term equivalence relation.

Almost all dependently-typed languages require this relation to be decidable. Some, such as Coq, Epigram or Agda, do so by employing analyses to force all programs to terminate. Conversely, others, such as DML, ATS, Ωmega, or Haskell, allow nonterminating computation, but do not allow those terms to appear in types. Instead, they identify a terminating index language and use singleton types to connect indices to computation. In both cases, decidable type checking comes at a cost, in terms of complexity and expressiveness.

Conversely, the benefits to be gained by decidable type checking are modest. Termination analyses allow dependently typed programs to verify total correctness properties. However, decidable type checking is not a prerequisite for type safety. Furthermore, decidability does not imply tractability. A decidable approximation of program equivalence may not be useful in practice.

Therefore, we take a different approach: instead of a fixed notion for term equivalence, we parameterize our type system with an abstract relation that is not necessarily decidable. We then design a novel set of typing rules that require only weak properties of this abstract relation in the proof of the preservation and progress lemmas. This design provides flexibility: we compare valid instantiations of term equivalence which range from beta-equivalence, to contextual equivalence, to some exotic equivalences.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory

General Terms Design, Languages, Theory

Keywords Dependent types, Program equivalence

1. Introduction

Dependent type systems promise the smooth integration of lightweight invariant checking with full program verification. In languages with dependent types, the types of a program may express rich, statically-checkable properties about its behavior.

Central in the design of a dependently-typed language is the notion of type equivalence. Because types include programs, type checking requires a definition of term equivalence. Therefore, decidable type checking requires that the term equivalence relation be decidable.

Previous work has almost uniformly insisted on decidable type checking, and hence decidable term equivalence. Some languages, such as Coq [Coq Development Team 2009], Epigram [McBride and McKinna 2004] or Agda [Norell 2007], do so by employing analysis that force all programs to terminate. This strong requirement has the benefit that type checking implies total correctness. If a function has type $\tau \to \Sigma y{:}\tau'.\ P\ y$ then one can be assured that it will terminate and produce a value satisfying property P.

Other languages, such as Dependent ML [Xi and Pfenning 1999], ATS [Xi 2004], Ωmega [Sheard 2006] and Haskell (with GADTs [Peyton Jones et al. 2006]), allow diverging computation and sacrifice total correctness. They retain decidable type checking by not allowing terms to appear in types. Instead, they identify a terminating index language (such as the type language in the case of Haskell) and use singleton types to connect indices to computation.

In each of these cases, decidable type checking comes at a cost, in terms of both complexity and expressiveness. Requiring all programs to terminate severely limits the generality of a programming language. Furthermore, the complexity of the termination analysis can make it difficult for programmers to understand why their code does not type check. In phase-sensitive languages, singleton types lead to code duplication, as programs must often be written twice, once in the computation language and again in the index language. More troublesome, there is no restriction that the semantics of the index language match that of the computation language: only their first-order values are required to agree.

At the same time, the benefits to be gained by decidable type checking are modest. Although termination analyses provide stronger correctness guarantees, they do not need to be integrated into the type system. Partial correctness guarantees that are implied by type safety could be extended to total correctness where necessary by an external termination analysis. Furthermore, decidability does not imply practicality. Why rule out undecidable specifications a priori, when they could behave well in practice?

Therefore, we design a full-spectrum, dependently-typed language $\lambda^{\cong}$, pronounced "lambda-eek", that does not pre-

suppose decidable program equivalence. This language is both simple and expressive: not only does it include general recursive function definitions and dependent products, but it also supports indexed datatypes with elimination forms to both terms (case expressions) and types (large eliminations).

Dependent type systems with undecidable type checking are compatible with type safety [Cardelli 1986], and have been explored before [Augustsson 1998]. We extend this line of work by making key design decisions that permit a simple proof of type saftey. Our straightforward proof of type safety is based on standard preservation and progress lemmas and has been formalized in the Coq proof assistant.

An important aspect of $\lambda^\cong$ is that it is actually a *family of languages* because its type system is parameterized by an abstract relation that specifies program equivalence. This three-place relation, written $\textbf{isEq}\,(\,\Delta\,,\,e_1\,,\,e_2\,)$, asserts when terms e_1 and e_2 are equivalent in some context Δ of assumptions about the equivalence of terms. This specification of program equivalence is isolated from typing, and the type safety proof depends on properties of program equivalence that make no reference to the type system. This separation simplifies the type safety proof.

For generality, we would like weak requirements for **isEq**. In particular, we would like to admit call-by-value respecting equivalences, since the operational semantics of $\lambda^\cong$ is call-by-value. Surprisingly, we revised our design several times before we found one one that would admit such relations.

Although it is impossible to claim that we have the weakest possible set of requirements, our design permits many different equivalence relations: from standard β-equivalence, to contextual equivalence, to some exotic equivalences. The finest equivalence makes our system admit no more terms than the simply-typed lambda calculus. More surprisingly, equivalences based on call-by-name evaluation are also valid, as well as some exotic equivalences that identify certain terminating and nonterminating expressions. We also have Coq proofs that show that all of these equivalences satisfy the required properties.

We also found that the requirements of the preservation proof force all valid instantiations of **isEq** to be undecidable. However, preservation is not a necessary requirement for type safety. Any language that type checks strictly fewer programs than a type-safe language is itself type safe. Therefore, any *decidable approximation* of a particular notion of program equivalence also defines a type-safe language. Consequently, $\lambda^\cong$ can be used as a template for languages with both decidable and undecidable definitions of program equivalence.

The organization of this paper is as follows. In Section 2 we introduce the syntax and operational semantics of $\lambda^\cong$. We then describe its type system, parameterized by the predicate **isEq** in Section 3. Working through a standard proof of preservation and progress leads to requirements on **isEq**—we describe those properties in Section 4. In Section 5 we give several definitions of **isEq** that satisfy our requirements. Variations of our type system lead to stronger requirements on **isEq**, which we discuss in Section 6. We discuss extensions to this system and other issues in Section 7. Finally, in Sections 8 and 9 we discuss related work and conclude.

The Coq development[1] for this paper is available online at `http://www.seas.upenn.edu/~plclub/lambda-eek/lambda-eek.tgz`.

---

[1] Because we are dealing with potentially undecidable relations, our developments use Coq's classical logic library.

$$
\begin{array}{lll}
\text{Terms} & e,\, u & ::= & x \mid \textbf{unit} \mid \textbf{fun}\, f(x) = e \mid e_1\, e_2 \\
& & \mid & \langle\, e_1\,,\, e_2\,\rangle \mid e.\,1 \mid e.\,2 \\
& & \mid & C\, e \mid \textbf{case}\, e\, \textbf{of}\, \{\, \overline{C_i\, x_i \Rightarrow e_i}^{\,i}\, \} \\
\text{Values} & v & ::= & \textbf{unit} \mid \textbf{fun}\, f(x) = e \mid \langle\, v_1\,,\, v_2\,\rangle \mid C\, v
\end{array}
$$

Figure 1. Syntax

$$\overline{(\,\textbf{fun}\, f(x) = e_1\,)\, v_2 \longrightarrow e_1\{v_2/x\}\{\textbf{fun}\, f(x) = e_1/f\}}$$

$$\overline{\langle\, v_1\,,\, v_2\,\rangle.\,1 \longrightarrow v_1} \qquad \overline{\langle\, v_1\,,\, v_2\,\rangle.\,2 \longrightarrow v_2}$$

$$\frac{C_j \in \overline{C_i}^{\,i\in 1..n}}{\textbf{case}\, C_j\, v\, \textbf{of}\, \{\, \overline{C_i\, x_i \Rightarrow e_i}^{\,i\in 1..n}\, \} \longrightarrow e_j\{v/x_j\}}$$

Figure 2. Operational semantics (excerpt)

2. A call-by-value language

Figure 1 presents the syntax of terms and values of $\lambda^\cong$. Importantly, terms do not contain typing annotations in order to isolate the specification of **isEq** from the type system of $\lambda^\cong$. A worry is that **isEq** might distinguish between terms with syntactically different but semantically equivalent type annotations. To trivially rule this out, terms do not contain types, and $\lambda^\cong$ uses a Curry-style type system.

The term language includes only standard features of programming languages: variables, unit, (recursive) functions, applications, binary products, projections, data constructors and case analysis. We use the metavariables e and u to denote terms and v to denote values. In a recursive function $\textbf{fun}\, f(x) = e$, the variables f and x are bound in the body e. If f does not appear in the body of the function, then we write it as $\lambda x. e$. In a case expression $\textbf{case}\, e\, \textbf{of}\, \{\, \overline{C_i\, x_i \Rightarrow e_i}^{\,i}\, \}$, the variables x_i are bound within each of the branches e_i. We use the notation $e\{e'/x\}$ for the capture-avoiding substitution of e' for x in e.

For simplicity, every data constructor must be of arity one and must always be applied to its argument. This limitation does not affect expressiveness—nullary and multiargument data constructors can be encoded. Throughout the paper, we assume a standard Peano encoding of natural numbers, with, for example, 0 represented as $C_{zero}\, \textbf{unit}$ and 1 represented by $C_{succ}\, (C_{zero}\, \textbf{unit})$. The boolean values `true` and `false` can be similarly encoded.

The key rules for the small-step, call-by-value operational semantics appear in Figure 2. This semantics is completely standard. Importantly, applications of recursive functions only step when their arguments are values.

3. A parameterized type system

We now define the type system for $\lambda^\cong$. Figure 3 defines the necessary additions to the syntax. The judgment forms of the type system are summarized in Figure 4. The rules of the type system itself appear in Figures 5, 6 and 7.

The types of $\lambda^\cong$ are divided into *proper types* of kind $*$ that classify terms directly; and *indexed types* of kind $(x{:}\tau) \Rightarrow *$ that must first be applied to a single term (of type τ).

Proper types include **Unit**, the type of the **unit** term, function types $(x{:}\tau) \rightarrow \tau'$ and product types $\Sigma x{:}\tau.\,\tau'$. In the latter two types, the variable x is bound in τ'. The result type of a function may depend on the argument value, and the type of the second component of a product may depend on the first component.

Kinds	κ	$::= * \mid (x{:}\tau) \Rightarrow *$
Types	$\tau, \sigma ::=$	$\mathbf{Unit} \mid (x{:}\tau) \to \tau' \mid \Sigma x{:}\tau.\,\tau' \mid T$
		$\mid \tau\,e \mid \mathbf{case}\,e\,\langle\,T\,u\,\rangle\,\mathbf{of}\,\{\,\overline{C_i\,x_i \Rightarrow \tau_i}^{\,i}\,\}$
Signatures	Ψ	$::= \cdot \mid \Psi, C : (x{:}\tau) \to T\,e$
		$\mid \Psi, T : (x{:}\tau) \Rightarrow *$
Contexts	Γ	$::= \cdot \mid \Gamma, x : \tau \mid \Gamma,\,e \cong e'$
Eq ctxs	Δ	$::= \cdot \mid \Delta,\,e \cong e'$
Pure terms	w	$::= x \mid \mathbf{unit} \mid \mathbf{fun}\,f(x) = e$
		$\mid \langle\,w_1\,,\,w_2\,\rangle \mid w.1 \mid w.2 \mid C\,w$

Figure 3. Types and contexts

Data constructors are typed by datatype constants, T, which are indexed types. The kinds of datatype constants and the types of data constructors are recorded by a signature Ψ. We assume that there is one fixed, well-formed signature Ψ_0 for an entire program, so we leave it implicit. We also assume that all data constructors and datatype constants are in the domain of Ψ_0.

For simplicity, we require that all datatype constants be of kind $(x{:}\tau) \Rightarrow *$. Standard data types use the uninformative index $\mathbf{unit}$. For example, the notation $\mathbf{Nat}$ abbreviates the type $T_{Nat}\,\mathbf{unit}$, where the constant T_{Nat} has kind $(x{:}\mathbf{Unit}) \Rightarrow *$. We use a similar definition for the type $\mathbf{Bool}$.

Often, however, the index is informative. For example, suppose the constant T_{List} is indexed by its length, a natural number. The data constructor C_{nil} creates a list of type $T_{List}\,0$. When type checking a case analysis where the scrutinee has type $T_{List}\,x$, the type checker can assume that x is equal to 0 in the C_{nil} branch.

The type language also includes a strong elimination form: case analysis of terms to produce types. In a type pattern match, $\mathbf{case}\,e\,\langle\,T\,u\,\rangle\,\mathbf{of}\,\{\,\overline{C_i\,x_i \Rightarrow \tau_i}^{\,i}\,\}$, a finite number of types τ_i are indexed by a term e that is expected to be of type $T\,u$. (We discuss the need for this annotation in Section 3.3.) This mechanism provides the technique of "Universes" in dependently-typed languages. For example, in a context containing the assumption $x : \mathbf{Bool}$, the term

$\quad\mathbf{case}\,x\,\mathbf{of}\,\{\,\mathtt{true} \Rightarrow 1\,;\,\mathtt{false} \Rightarrow \mathtt{false}\,\}$

can be assigned the type

$\quad\mathbf{case}\,x\,\langle\,\mathbf{Bool}\,\rangle\,\mathbf{of}\,\{\,\mathtt{true} \Rightarrow \mathbf{Nat}\,;\,\mathtt{false} \Rightarrow \mathbf{Bool}\,\}$

The type system is defined in terms of a number of assumption lists. Besides signatures Ψ, there are *contexts* Γ and *equivalence contexts* Δ. Contexts are ordered lists of variable type assumptions and term equivalence assumptions. The domain of a context is the set of variables for which there are type assumptions. Equivalence contexts Δ contain term equivalence assumptions only. We denote context concatenation with Γ, Γ' when the domain of Γ and Γ' are disjoint (likewise Δ, Δ'). We use $\Gamma^\star$ to produce the equivalence context containing the equivalence assumptions in Γ.

Some places in the specification of the type system require the definition of *pure terms*. We use the metavariable w to range over a simple set of terms that are known to terminate.

3.1 Parameterized equivalence: isEq

As mentioned above, the type system of $\lambda^\cong$ is parameterized by the predicate $\mathbf{isEq}\,(\,\Delta\,,\,e\,,\,e'\,)$. This predicate holds when the terms e and e' are equivalent under the set of equivalence assumptions in Δ.

Formation Judgments		Equivalence Judgments	
$\vdash \Psi$	Signature	$\vdash \Delta \equiv \Delta'$	Equivalence Context
$\vdash \Gamma$	Context	$\vdash \Gamma \equiv \Gamma'$	Context
$\Gamma \vdash \kappa$	Kinds	$\Delta \vdash \tau \equiv \tau'$	Types
$\Gamma \vdash \tau : \kappa$	Types	$\Delta \vdash \kappa \equiv \kappa'$	Kinds
$\Gamma \vdash e : \tau$	Terms		

Figure 4. Type system judgment forms

We use $\mathbf{isEq}$ to define an auxiliary relation used for type checking. The predicate $\mathbf{incon}\,(\,\Delta\,)$ determines if there exists a contradiction in the equivalence assumptions of Δ. An equivalence context Δ is inconsistent when $\mathbf{isEq}$ equates two pure terms headed by different constructors.

DEFINITION 3.1 (Inconsistency).
Define $\mathbf{incon}\,(\,\Delta\,)$ *if there exist terms* $C_i\,w_i$ *and* $C_j\,w_j$ *such that* $\mathbf{isEq}\,(\,\Delta\,,\,C_i\,w_i\,,\,C_j\,w_j\,)$ *and* $C_i \neq C_j$.

Most dependently typed languages use β-equivalence or $\beta\eta$-equivalence to decide term equivalence. In our language, we leave $\mathbf{isEq}$ abstract. However, to ensure that our system enjoys standard properties (such as preservation and progress) $\mathbf{isEq}$ must itself satisfy a number of properties, as we describe in Section 4.

The equivalence assumptions in Δ are equations between arbitrary terms. These terms do not need to be well-typed or even have the same type (though our rules only add such assumptions to the equivalence context). Furthermore, these equations do not need to be consistent, though when they are not, all terms are typeable with all types.

3.2 Typing

The type system of $\lambda^\cong$ is defined by two main categories of judgments (see Figure 4). One set determines when syntactic elements are well-formed. The other set determines when they are equivalent. The formation rules refer to the equivalence rules, but the equivalence rules are independent. We start our discussion with the formation rules, and return to the equivalence rules in Section 3.3.

The formation rules appear in Figures 5 and 6. Most rules are straightforward; we focus on the term typing rules. One significant departure from standard rules is that we use equivalence assumptions instead of substitution. For example, a standard rule for application substitutes the operand e_2 for the variable in the result type:

$$\frac{\Gamma \vdash e_1 : (x{:}\tau_1) \to \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\,e_2 : \tau_2\{e_2/x\}}\;\text{E\_APP}'$$

However, in $\lambda^\cong$, instead of substituting the operand e_2 in the result type, rule E_APP checks if τ_2 is equal to some τ under an equivalence context that extends $\Gamma^\star$ with the equation $x \cong e_2$. Furthermore, to ensure that x is not free in τ, the rule checks that τ is well-formed under the context Γ. Similarly, the typing rules for dependent pairs, projections and constructors also extend the context with equivalence assumptions rather than use explicit substitution.

We use equivalence assumptions instead of substitution because substituting e into a type leads to stronger requirements on the substitution property of $\mathbf{isEq}$. Intuitively, requiring that $\mathbf{isEq}$ be closed under substituting an arbitrary e limits our term equivalence relations to those based on call-by-name evaluation. However, our system is call-by-value, leading to an undesirable mismatch. We discuss this issue further in Section 6.1.

$\boxed{\vdash \Gamma}$

$$\frac{}{\vdash \cdot} \text{ C\_E} \qquad \frac{\vdash \Gamma \quad x \notin dom(\Gamma) \quad \Gamma \vdash \tau : *}{\vdash \Gamma, x : \tau} \text{ C\_Term}$$

$$\frac{\vdash \Gamma \quad \Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\vdash \Gamma, \, e_1 \cong e_2} \text{ C\_Eq}$$

$\boxed{\Gamma \vdash \kappa}$

$$\frac{\vdash \Gamma}{\Gamma \vdash *} \text{ K\_Type} \qquad \frac{\Gamma, x : \tau \vdash \kappa}{\Gamma \vdash (x{:}\tau) \Rightarrow \kappa} \text{ K\_Pi}$$

$\boxed{\Gamma \vdash \tau : \kappa}$

$$\frac{\vdash \Gamma}{\Gamma \vdash \mathbf{Unit} : *} \text{ T\_Unit} \qquad \frac{\vdash \Gamma \quad T : \kappa \in \Psi_0}{\Gamma \vdash T : \kappa} \text{ T\_Cst}$$

$$\frac{\Gamma, x : \tau_1 \vdash \tau_2 : *}{\Gamma \vdash (x{:}\tau_1) \to \tau_2 : *} \text{ T\_Pi} \qquad \frac{\Gamma, x : \tau_1 \vdash \tau_2 : *}{\Gamma \vdash \Sigma x{:}\tau_1. \tau_2 : *} \text{ T\_Sigma}$$

$$\frac{\Gamma \vdash \tau : (x{:}\tau_1) \Rightarrow \kappa_1 \quad \Gamma \vdash e : \tau_1 \quad \Gamma^\star, x \cong e \vdash \kappa_1 \equiv \kappa \quad \Gamma \vdash \kappa}{\Gamma \vdash \tau\,e : \kappa} \text{ T\_App}$$

$$\frac{\Gamma \vdash \kappa \quad \Gamma \vdash e : T\,u \quad \mathsf{CtrOf}(T) = \overline{C_i}^{\,i \in 1..n} \quad \overline{C_i : (x_i{:}\sigma_i) \to T\,u_i \in \Psi_0}^{\,i \in 1..n} \quad \overline{\Gamma, x_i : \sigma_i, \, u \cong u_i, \, e \cong C_i\,x_i \vdash \tau_i : \kappa}^{\,i \in 1..n}}{\Gamma \vdash \mathbf{case}\, e \, \langle\, T\,u\,\rangle \, \mathbf{of}\, \{ \, \overline{C_i\,x_i \Rightarrow \tau_i}^{\,i \in 1..n} \, \} : \kappa} \text{ T\_Case}$$

$$\frac{\vdash \Gamma \quad \mathbf{incon}\,(\Gamma^\star)}{\Gamma \vdash \tau : \kappa} \text{ T\_InCon}$$

$$\frac{\Gamma \vdash \tau : \kappa \quad \Gamma^\star \vdash \kappa \equiv \kappa' \quad \Gamma \vdash \kappa'}{\Gamma \vdash \tau : \kappa'} \text{ T\_KConv}$$

Figure 5. Context, kind, and type formation rules

$\boxed{\Gamma \vdash e : \tau}$

$$\frac{\vdash \Gamma \quad x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \text{ E\_Var} \qquad \frac{\vdash \Gamma}{\Gamma \vdash \mathbf{unit} : \mathbf{Unit}} \text{ E\_Unit}$$

$$\frac{\Gamma, x : \tau_1, f : (x{:}\tau_1) \to \tau_2 \vdash e : \tau_2}{\Gamma \vdash \mathbf{fun}\,f(x) = e : (x{:}\tau_1) \to \tau_2} \text{ E\_Fix}$$

$$\frac{\Gamma \vdash e_1 : (x{:}\tau_1) \to \tau_2 \quad \Gamma \vdash e_2 : \tau_1 \quad \Gamma^\star, x \cong e_2 \vdash \tau_2 \equiv \tau \quad \Gamma \vdash \tau : *}{\Gamma \vdash e_1\,e_2 : \tau} \text{ E\_App}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2' \quad \Gamma^\star, x \cong e_1 \vdash \tau_2' \equiv \tau_2 \quad \Gamma, x : \tau_1 \vdash \tau_2 : *}{\Gamma \vdash \langle\, e_1\,,\, e_2\,\rangle : \Sigma x{:}\tau_1. \tau_2} \text{ E\_Sigma}$$

$$\frac{\Gamma \vdash e : \Sigma x{:}\tau_1. \tau_2}{\Gamma \vdash e.1 : \tau_1} \text{ E\_Proj1}$$

$$\frac{\Gamma \vdash e : \Sigma x{:}\tau_1. \tau_2 \quad \Gamma \vdash \tau : * \quad \Gamma^\star, x \cong e.1 \vdash \tau_2 \equiv \tau}{\Gamma \vdash e.2 : \tau} \text{ E\_Proj2}$$

$$\frac{C : (x{:}\sigma) \to T\,u \in \Psi_0 \quad \Gamma \vdash e : \sigma \quad \Gamma^\star, x \cong e \vdash T\,u \equiv \tau \quad \Gamma \vdash \tau : *}{\Gamma \vdash C\,e : \tau} \text{ E\_Ctr}$$

$$\frac{\Gamma \vdash e : T\,u \quad \mathsf{CtrOf}(T) = \overline{C_i}^{\,i \in 1..n} \quad \Gamma \vdash \tau : * \quad \overline{C_i : (x_i{:}\tau_i) \to T\,u_i \in \Psi_0}^{\,i \in 1..n} \quad \overline{\Gamma, x_i : \tau_i, \, u \cong u_i, \, e \cong C_i\,x_i \vdash e_i : \tau}^{\,i \in 1..n}}{\Gamma \vdash \mathbf{case}\, e \, \mathbf{of}\, \{ \, \overline{C_i\,x_i \Rightarrow e_i}^{\,i \in 1..n} \, \} : \tau} \text{ E\_Case}$$

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

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

Figure 6. Term formation rules (typing)

The typing rule for pattern-match E_CASE also uses equivalence assumptions, but for a different purpose. This rule first uses the premise $\mathsf{CtrOf}(T) = \overline{C_i}^{\,i \in 1..n}$ to check that the branch is exhaustive. During execution, if the ith branch is taken, the scrutinee must match the pattern $C_i\,x_i$, and the index u of the scrutinee's type must match with the index u_i in the signature. Therefore, this rule checks each branch under a context that extends Γ with equivalence assumptions that the indices are the same ($u \cong u_i$) and that the scrutinee is equal to the pattern ($e \cong C_i\,x_i$).

The fact that $\lambda^\cong$ uses equivalence assumptions instead of substitution to represent the information gained via case analysis is powerful. In particular, $\lambda^\cong$ can take advantage of information such as $f\,x \cong \mathtt{true}$ in a way that languages such as Coq and Agda cannot. For example, suppose we have a datatype T indexed by booleans with constructors $C_1 : T\,\mathtt{true}$ and $C_2 : T\,\mathtt{false}$. Then, in the following context

$f : \mathbf{Nat} \to \mathbf{Bool}, x : \mathbf{Nat}, h : T\,(f\,x) \to \mathbf{Bool}$

there are instantiations of **isEq** such that the following term typechecks

$\mathbf{case}\,f\,x\,\mathbf{of}\,\{\mathtt{true} \Rightarrow h\,C_1; \mathtt{false} \Rightarrow \mathit{false}\}$

To typecheck $h\,C_1$, the type checker must show the equivalence of $T\,(f\,x)$ and $T\,\mathtt{true}$ in the first branch, when the equation $f\,x \cong \mathtt{true}$ is available. Systems based on unification cannot make this information available via substitution, so they require the result of $f\,x$ to be named.[2]

Note that in rule E_CASE, the order in which the equivalence assumptions are added to the context is important for maintaining the well-formedness of the context. The type of e is $T\,u$, and the type of $C_i\,x_i$ is $T\,u_i$. For the extended context to be well-formed, we need to insert the assumption $u \cong u_i$ before $e \cong C_i\,x_i$, so that $u \cong u_i$ is available for checking that e and $C_i\,x_i$ have the same type.

The equivalence assumptions in Γ could become inconsistent, for example, while checking a $\mathtt{false}$ branch in a case expression where the scrutinee is $\mathtt{true}$. In that case, the assumption $\mathtt{true} \cong \mathtt{false}$ is added to the context. However, this branch is inaccessible at runtime, so there is no need to type check it. Therefore, rule E_INCON assigns an arbitrary type τ to e when the equivalence assumptions in Γ are contradictory.

---

[2] The Agda version of this example typechecks due to some ad hoc machinery, but small variations do not.

$$\boxed{\Delta \vdash \kappa \equiv \kappa'}$$

$$\frac{}{\Delta \vdash * \equiv *} \quad \text{KQ\_Refl}$$

$$\frac{\Delta \vdash \tau \equiv \tau' \quad \Delta \vdash \kappa \equiv \kappa'}{\Delta \vdash (x{:}\tau) \Rightarrow \kappa \equiv (x{:}\tau') \Rightarrow \kappa'} \quad \text{KQ\_Pi}$$

$$\boxed{\Delta \vdash \tau \equiv \tau'}$$

$$\frac{\mathbf{incon}\,(\Delta)}{\Delta \vdash \tau \equiv \tau'} \quad \text{TQ\_InCon}$$

$$\frac{}{\Delta \vdash \mathbf{Unit} \equiv \mathbf{Unit}} \quad \text{TQ\_URefl}$$

$$\frac{T : \kappa \in \Psi_0}{\Delta \vdash T \equiv T} \quad \text{TQ\_TRefl}$$

$$\frac{\Delta \vdash \tau_1 \equiv \tau_1' \quad \Delta \vdash \tau_2 \equiv \tau_2'}{\Delta \vdash (x{:}\tau_1) \to \tau_2 \equiv (x{:}\tau_1') \to \tau_2'} \quad \text{TQ\_Pi}$$

$$\frac{\Delta \vdash \tau_1 \equiv \tau_1' \quad \Delta \vdash \tau_2 \equiv \tau_2'}{\Delta \vdash \Sigma x{:}\tau_1.\,\tau_2 \equiv \Sigma x{:}\tau_1'.\,\tau_2'} \quad \text{TQ\_Sigma}$$

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

$$\frac{\begin{array}{c}\mathbf{isEq}\,(\Delta,\, e,\, e') \quad \mathbf{isEq}\,(\Delta,\, u,\, u') \\ \overline{C_i : (x_i{:}\sigma_i) \to T\,u_i \in \Psi_0}^{\,i \in 1..n} \\ \overline{\Delta,\, u \cong u_i,\, e \cong C_i\,x_i \vdash \tau_i \equiv \tau_i'}^{\,i \in 1..n}\end{array}}{\Delta \vdash \mathbf{case}\ e\,\langle\, T\,u\,\rangle\,\mathbf{of}\,\{\,\overline{C_i\,x_i \Rightarrow \tau_i}^{\,i \in 1..n}\,\} \equiv \ \mathbf{case}\ e'\,\langle\, T\,u'\,\rangle\,\mathbf{of}\,\{\,\overline{C_i\,x_i \Rightarrow \tau_i'}^{\,i \in 1..n}\,\}} \quad \text{TQ\_Case}$$

$$\frac{\begin{array}{c}\mathbf{isEq}\,(\Delta,\, e,\, C_j\,w) \quad C_j \in \overline{C_i}^{\,i \in 1..n} \\ C_j : (x_j{:}\sigma_j) \to T\,u_j \in \Psi_0 \\ \mathbf{isEq}\,((\,\Delta,\, w \cong x_j\,),\, u,\, u_j) \\ \Delta,\, w \cong x_j,\, e \cong C_j\,x_j \vdash \tau_j \equiv \tau\end{array}}{\Delta \vdash \mathbf{case}\ e\,\langle\, T\,u\,\rangle\,\mathbf{of}\,\{\,\overline{C_i\,x_i \Rightarrow \tau_i}^{\,i \in 1..n}\,\} \equiv \tau} \quad \text{TQ\_Red1}$$

$$\frac{\begin{array}{c}\mathbf{isEq}\,(\Delta,\, e,\, C_j\,w) \quad C_j \in \overline{C_i}^{\,i \in 1..n} \\ C_j : (x_j{:}\sigma_j) \to T\,u_j \in \Psi_0 \\ \mathbf{isEq}\,((\,\Delta,\, w \cong x_j\,),\, u,\, u_j) \\ \Delta,\, w \cong x_j,\, e \cong C_j\,x_j \vdash \tau \equiv \tau_j\end{array}}{\Delta \vdash \tau \equiv \mathbf{case}\ e\,\langle\, T\,u\,\rangle\,\mathbf{of}\,\{\,\overline{C_i\,x_i \Rightarrow \tau_i}^{\,i \in 1..n}\,\}} \quad \text{TQ\_Red2}$$

Figure 7. Kind and type equivalence

The last typing rule is a conversion rule. If e can be assigned type τ, then E_TConv allows e to be given any well-kinded type that is equivalent to τ.

3.3 Equivalence

Several typing rules require determining when two types are equivalent. A couple of type formation rules require kind equivalence. We present these two equivalence judgments for $\lambda^{\cong}$ in Figure 7. These judgments do not check well-formedness. Instead, the formation rules only use the equivalence judgments on well-formed constructs. For instance, in rule E_TConv, both τ and τ' must be well-kinded. This de-

sign allows the properties of equivalence to be proven independently of those for formation.

Most of the rules are straightforward. Below, we focus on the type equivalence rules. The type equivalence judgment has the form $\Delta \vdash \tau_1 \equiv \tau_2$, where Δ is the equivalence context under which τ_1 and τ_2 are considered.

The first rule, TQ_InCon, states that when Δ is inconsistent, any two types are equivalent. The next few rules are congruence rules stating that two types are equivalent if the corresponding sub-terms are equivalent. Rule TQ_App uses **isEq** to check the equivalence of the two embedded terms. The congruence rule for case types TQ_Case checks that the corresponding branches are equivalent with added assumptions that the actual index is equal to the stated index of the constructor and that the scrutinee is equal to pattern for that branch. This rule must check not only the equivalence of the scrutinees, but also that the indices in the scrutinees' types are equal. Because our equivalence rules do not depend on well-formedness rules, the only way to find out the type of the scrutinee is to annotate the case type with $\langle T\,u \rangle$.

The last two rules consider the situation when a case type could reduce to one of its branches. The rule TQ_Red2 is symmetric to TQ_Red1. The first premise of TQ_Red1 checks if the scrutinee e is equal to some pure term $C_j\,w$, where C_j heads one of the patterns. The rule also checks that the index u in e's type is equal to u_j, which is the index of $C_j\,x_j$'s type. If the jth branch τ_j is equivalent to a type τ (which does not contain x_j), then we can conclude that the case type is equivalent to τ.

Like E_Case, TQ_Red1 extends Δ with the equation $w \cong x_j$. Additionally, when checking if τ_j is equal to τ, both $w \cong x_j$ and $e \cong C_j\,x_j$ are in the context. Although the latter assumption is semantically redundant, not including this assumption leads to stronger requirements for **isEq**. Another design choice is why we require a pure term $C_j\,w$ in the first premise, instead of $C_j\,v$ or $C_j\,e$. We address this decision in Section 6.1.

Our type equivalence rules are defined to be easily invertible. For example, by examining the rules, we can conclude that there does not exist a derivation for $\Delta \vdash T\,e \equiv (x{:}\tau_1) \to \tau_2$ when Δ is consistent, an important property for the progress and preservation lemmas.

4. Properties of the type system

The type system of $\lambda^{\cong}$ depends on the relation $\mathbf{isEq}\,(\Delta,\, e_1,\, e_2)$. Consequently, the type safety property of $\lambda^{\cong}$ depends on properties of this relation. In this Section, we investigate the properties shown in Figure 8 that we use in the proof of the progress and preservation lemmas. Although these proofs are straightforward, we include details here to motivate each of the properties listed in Figure 8.

Note that these properties are independent of the type system. We make no requirements that the arguments to **isEq** have the same type, or even have a type, or that the assumptions in the equivalence context are well-formed in any way. Thus our parameterization is simple and well-defined.

4.1 Basic lemmas

We start with four basic properties (weakening, substitution, cut, and context conversion) that should hold for every judgment. Because our judgments include **isEq** as a hypothesis, these properties are also required for **isEq** (see the first four properties in Figure 8).

PROPERTY 4.1 (IsEq Weakening).
If $\textbf{isEq}\,((\,\Delta\,,\,\Delta''\,)\,,\,e_1\,,\,e_2\,)$,
then $\textbf{isEq}\,((\,\Delta\,,\,\Delta'\,,\,\Delta''\,)\,,\,e_1\,,\,e_2\,)$.

PROPERTY 4.2 (IsEq Substitution). *If* $\textbf{isEq}\,(\,\Delta\,,\,e_1\,,\,e_2\,)$, *then* $\textbf{isEq}\,(\,\Delta\{w/x\}\,,\,e_1\{w/x\}\,,\,e_2\{w/x\}\,)$.

PROPERTY 4.3 (IsEq Cut).
If $\textbf{isEq}\,((\,\Delta\,,\,u_1\,\cong\,u_2\,,\,\Delta'\,)\,,\,e_1\,,\,e_2\,)$, *and* $\textbf{isEq}\,(\,\Delta\,,\,u_1\,,\,u_2\,)$,
then $\textbf{isEq}\,((\,\Delta\,,\,\Delta'\,)\,,\,e_1\,,\,e_2\,)$.

PROPERTY 4.4 (IsEq Context Conversion).
If $\textbf{isEq}\,(\,\Delta\,,\,e_1\,,\,e_2\,)$, *and* $\vdash\,\Delta\,\equiv\,\Delta'$, *then* $\textbf{isEq}\,(\,\Delta'\,,\,e_1\,,\,e_2\,)$.

PROPERTY 4.5 (IsEq Reflexivity). $\textbf{isEq}\,(\,\Delta\,,\,e\,,\,e\,)$.

PROPERTY 4.6 (IsEq Symmetry). *If* $\textbf{isEq}\,(\,\Delta\,,\,e_1\,,\,e_2\,)$, *then* $\textbf{isEq}\,(\,\Delta\,,\,e_2\,,\,e_1\,)$.

PROPERTY 4.7 (IsEq Transitivity). *If* $\textbf{isEq}\,(\,\Delta\,,\,e_1\,,\,e_2\,)$, *and* $\textbf{isEq}\,(\,\Delta\,,\,e_2\,,\,e_3\,)$, *then* $\textbf{isEq}\,(\,\Delta\,,\,e_1\,,\,e_3\,)$.

PROPERTY 4.8 (IsEq Injectivity).
If $\textbf{isEq}\,(\,\Delta\,,\,C\,w_1\,,\,C\,w_2\,)$, *then* $\textbf{isEq}\,(\,\Delta\,,\,w_1\,,\,w_2\,)$.

PROPERTY 4.9 (IsEq Beta). *If* $e\,\longrightarrow\,e'$, *then* $\textbf{isEq}\,(\,\cdot\,,\,e\,,\,e'\,)$.

PROPERTY 4.10 (IsEq Empty).
If $C_i\,\neq\,C_j$, *then* $\neg\textbf{isEq}\,(\,\cdot\,,\,C_i\,w_i\,,\,C_j\,w_j\,)$.

Figure 8. The **isEq** Properties

Weakening states that if a judgment holds under context Γ (or Δ), then it also holds under a larger context.

LEMMA 4.1 (Weakening).
1. If $\Delta_1\,,\,\Delta_3\,\vdash\,J$, *then* $\Delta_1\,,\,\Delta_2\,,\,\Delta_3\,\vdash\,J$.
2. If $\Gamma_1\,,\,\Gamma_3\,\vdash\,J$, *and* $\vdash\,\Gamma_1\,,\,\Gamma_2\,,\,\Gamma_3$, *then* $\Gamma_1\,,\,\Gamma_2\,,\,\Gamma_3\,\vdash\,J$.

The Substitution Lemma states that equivalence judgments are closed under the substitution of *pure* terms and that the formation judgments are closed under the substitution of values.

LEMMA 4.2 (Substitution).
1. If $\Delta\,\vdash\,J$ *then* $\Delta\{w/x\}\,\vdash\,J\{w/x\}$.
2. If $\Gamma,x:\tau_1\,,\,\Gamma'\,\vdash\,J$ *and* $\Gamma\,\vdash\,v\,:\,\tau_1$
   then $\Gamma\,,\,\Gamma'\{v/x\}\,\vdash\,J\{v/x\}$.

Because our language has a call-by-value semantics, we do not need substitution to be true for arbitrary terms, only pure terms and values respectively. As a result, **isEq** need only be closed over the substitution of pure terms. Property 4.2 is a particularly weak requirement, as we discuss in Section 6.

The Cut Lemma removes redundant equivalence assumptions from the context.

LEMMA 4.3 (Cut).
1. If $\Delta\,,\,e\,\cong\,e'\,,\,\Delta'\,\vdash\,J$ *and* $\textbf{isEq}\,(\,\Delta\,,\,e\,,\,e'\,)$
   then $\Delta\,,\,\Delta'\,\vdash\,J$.
2. If $\Gamma\,,\,e\,\cong\,e'\,,\,\Gamma'\,\vdash\,J$ *and* $\textbf{isEq}\,(\,\Gamma^\star\,,\,e\,,\,e'\,)$
   then $\Gamma\,,\,\Gamma'\,\vdash\,J$.

Finally, both the equivalence judgments and the formation judgments are closed under equivalent contexts. To state this lemma, we first define when both sorts of contexts are equivalent. Although these definitions are asymmetric (they always use their left argument to compare each pair) the de-

fined relations are symmetric because of IsEq Context Conversion (Prop 4.4).

DEFINITION 4.1 (Δ-Equivalence).

$$\overline{\vdash\,\cdot\,\equiv\,\cdot}$$

$$\frac{\vdash\,\Delta\,\equiv\,\Delta'\quad\textbf{isEq}\,(\,\Delta\,,\,e_1\,,\,e_1'\,)\quad\textbf{isEq}\,(\,\Delta\,,\,e_2\,,\,e_2'\,)}{\vdash\,\Delta\,,\,e_1\,\cong\,e_2\,\equiv\,\Delta'\,,\,e_1'\,\cong\,e_2'}$$

DEFINITION 4.2 (Context equivalence).

$$\frac{}{\vdash\,\cdot\,\equiv\,\cdot}\;\text{CQ\_EMPTY}$$

$$\frac{\vdash\,\Gamma\,\equiv\,\Gamma'\quad\Gamma^\star\,\vdash\,\tau\,\equiv\,\tau'}{\vdash\,\Gamma,x:\tau\,\equiv\,\Gamma',x:\tau'}\;\text{CQ\_TERM}$$

$$\frac{\vdash\,\Gamma\,\equiv\,\Gamma'\quad\textbf{isEq}\,(\,\Gamma^\star\,,\,e_1\,,\,e_1'\,)\quad\textbf{isEq}\,(\,\Gamma^\star\,,\,e_2\,,\,e_2'\,)}{\vdash\,(\,\Gamma\,,\,e_1\,\cong\,e_2\,)\,\equiv\,(\,\Gamma'\,,\,e_1'\,\cong\,e_2'\,)}\;\text{CQ\_EQ}$$

We then show that all formation judgments are stable under context equivalence, and that all equivalence judgments are stable under Δ-equivalence.

LEMMA 4.4 (Context Conversion).
1. If $\Delta\,\vdash\,J$ *and* $\vdash\,\Delta\,\equiv\,\Delta'$ *then* $\Delta'\,\vdash\,J$.
2. If $\Gamma\,\vdash\,J$ *and* $\vdash\,\Gamma\,\equiv\,\Gamma'$ *and* $\vdash\,\Gamma'$ *then* $\Gamma'\,\vdash\,J$.

4.2 Properties of type equivalence

The type equivalence rules shown in Figure 7 do not contain rules for reflexivity, symmetry, or transitivity, permitting simple inversion. Instead, we prove the following lemmas about the equivalence judgments to show that these rules are admissible. Again, to show these properties, they also must be true of **isEq** (see Properties 4.5-4.7 in Figure 8).

LEMMA 4.5 (Refl). $\Delta\,\vdash\,\tau\,\equiv\,\tau$.
LEMMA 4.6 (Symm). *If* $\Delta\,\vdash\,\tau\,\equiv\,\tau'$ *then* $\Delta\,\vdash\,\tau'\,\equiv\,\tau$.
LEMMA 4.7 (Transitivity). *If* $\Delta\,\vdash\,\tau\,\equiv\,\tau'$ *and* $\Delta\,\vdash\,\tau'\,\equiv\,\tau''$ *then* $\Delta\,\vdash\,\tau\,\equiv\,\tau''$.

The proofs of reflexivity and symmetry are straightforward, but transitivity is less so, so we show one case of the proof below. This proof requires one more property of **isEq**—that data constructors are injective for pure terms (Prop. 4.8). To show transitivity, we must first generalize the statement of the lemma so that the contexts of the two type equivalence derivations are not the same, but are equivalent.

LEMMA 4.8 (Transitivity'). *If* $\Delta\,\vdash\,\tau\,\equiv\,\tau'$ *and* $\Delta'\,\vdash\,\tau'\,\equiv\,\tau''$ *and* $\vdash\,\Delta\,\equiv\,\Delta'$ *then* $\Delta\,\vdash\,\tau\,\equiv\,\tau''$.

The proof is by a double induction on the structure of the pair of assumed judgments; call the first one $\mathcal{D}$ and the second one $\mathcal{E}$. Consider the case where the last rule used in $\mathcal{D}$ is TQ_RED2 and the last rule of $\mathcal{E}$ is TQ_RED1. Then, these derivations are of the form:

$$\frac{\begin{array}{ll}C_j\,\in\,\overline{C_i}^{\,i\in 1..n} & C_j\,:\,(x_j{:}\sigma_j)\,\to\,T\,u_j\,\in\,\Psi_0\\[4pt]\textbf{isEq}\,(\,\Delta\,,\,e\,,\,C_j\,w\,) & \textbf{isEq}\,((\,\Delta\,,\,w\,\cong\,x_j\,)\,,\,u\,,\,u_j\,)\\[4pt]\Delta\,,\,w\,\cong\,x_j\,,\,e\,\cong\,C_j\,x_j\,\vdash\,\sigma\,\equiv\,\tau_j\end{array}}{\Delta\,\vdash\,\sigma\,\equiv\,\textbf{case}\,e\,\langle\,T\,u\,\rangle\,\textbf{of}\,\{\,\overline{C_i\,x_i\,\Rightarrow\,\tau_i}^{\,i\in 1..n}\,\}}$$

and

$$\frac{\begin{array}{ll}C_m\,\in\,\overline{C_i}^{\,i\in 1..n} & C_m\,:\,(x_m{:}\sigma_m)\,\to\,T\,u_m\,\in\,\Psi_0\\[4pt]\textbf{isEq}\,(\,\Delta'\,,\,e\,,\,C_m\,w'\,) & \textbf{isEq}\,((\,\Delta'\,,\,w'\,\cong\,x_m\,)\,,\,u\,,\,u_m\,)\\[4pt]\Delta'\,,\,w'\,\cong\,x_m\,,\,e\,\cong\,C_m\,x_m\,\vdash\,\tau_m\,\equiv\,\sigma'\end{array}}{\Delta'\,\vdash\,\textbf{case}\,e\,\langle\,T\,u\,\rangle\,\textbf{of}\,\{\,\overline{C_i\,x_i\,\Rightarrow\,\tau_i}^{\,i\in 1..n}\,\}\,\equiv\,\sigma'}$$

We need to show that $\Delta \vdash \sigma \equiv \sigma'$. To use the induction hypothesis, we need to know that both $\mathcal{E}$ and $\mathcal{D}$ reduce using the same branch. In other words, $j = m$. We know that $\mathbf{isEq}(\Delta, e, C_j w)$ and $\mathbf{isEq}(\Delta', e, C_m w')$. By the Symmetry, Transitivity, and Context Conversion Properties of $\mathbf{isEq}$, we conclude that $\mathbf{isEq}(\Delta, C_j w, C_m w')$. To continue the proof, we must conclude either that $C_j = C_m$ or that Δ is inconsistent, hence our definition of $\mathbf{incon}(\Delta)$.

Now suppose that $j = m$. To apply the induction hypothesis, we must show

$$\vdash (\Delta, w \cong x_j, e \cong C_j x_j) \equiv (\Delta', w' \cong x_j, e \cong C_j x_j)$$

We have $\vdash \Delta \equiv \Delta'$ by assumption, so for these two contexts to be equivalent, we need only show $\mathbf{isEq}(\Delta, w, w')$. We also have $\mathbf{isEq}(\Delta, C_j w, C_j w')$, so the Injectivity Property (4.8) of $\mathbf{isEq}$ suffices.

By applying the induction hypothesis, we have $\Delta, w \cong x_j, e \cong C_j x_j \vdash \sigma \equiv \sigma'$. By substituting w for x_j, we conclude that $\Delta, w \cong w, e \cong C_j w \vdash \sigma \equiv \sigma'$ (because x_j is not free in Δ, e, σ and σ'). To conclude $\Delta \vdash \sigma \equiv \sigma'$, we need only remove $w \cong w$ and $e \cong C_j w$ from the context. We already know these facts via reflexivity and assumption, so we use the Cut Lemma (4.3), finishing the case.

In this proof we must substitute a pure term w into the judgment, not a value v. For that reason, our Substitution Lemma (4.2) on equivalence must hold for pure terms.

4.3 Type safety

We prove type safety for our language via standard progress and preservation Lemmas [Wright and Felleisen 1994].

LEMMA 4.9 (Preservation). *If* $\Gamma \vdash e : \tau$ *and* $e \longrightarrow e'$, *then* $\Gamma \vdash e' : \tau$.

The proof is by induction on the reduction relation. In some of the cases, the typing of e' depends on a subterm in e' that takes a step. Those cases make yet another requirement on $\mathbf{isEq}$, the IsEq Beta Property (4.9). We use the case when $e = e_1 e_2$ and $e_2 \longrightarrow e_2'$ as an example. By assumption we know that

$$\frac{\Gamma \vdash e_1 : (x{:}\tau_1) \rightarrow \tau_2 \quad \Gamma \vdash e_2 : \tau_1 \quad \Gamma^\star, x \cong e_2 \vdash \tau_2 \equiv \tau \quad \Gamma \vdash \tau : *}{\Gamma \vdash e_1 e_2 : \tau} \ \text{E\_App}$$

By the induction hypothesis, we know $\Gamma \vdash e_2' : \tau_1$. We need to show that $\Gamma^\star, x \cong e_2' \vdash \tau_2 \equiv \tau$. Because Property 4.9 requires $\mathbf{isEq}$ to identify e_2 and e_2', we know that the context $\Gamma^\star, x \cong e_2$ is equivalent to the context $\Gamma^\star, x \cong e_2'$. Therefore, by using context conversion (Lemma 4.4), we can conclude $\Gamma^\star, x \cong e_2' \vdash \tau_2 \equiv \tau$.

To show progress, we must first prove a canonical forms lemma.

LEMMA 4.10 (Canonical Forms). *Suppose* $\neg\mathbf{incon}(\Gamma^\star)$.

1. *If* $\Gamma \vdash v : \mathbf{Unit}$ *then* v *is* $\mathbf{unit}$.
2. *If* $\Gamma \vdash v : (x{:}\tau_1) \rightarrow \tau_2$ *then* v *is* $\mathbf{fun}\, f(x) = e$.
3. *If* $\Gamma \vdash v : \Sigma x{:}\tau_1.\,\tau_2$ *then* v *is* $\langle v_1, v_2 \rangle$.
4. *If* $\Gamma \vdash v : T\, e$ *then* v *is* $C\, v'$ *and* $C : (x{:}\sigma) \rightarrow T\, u \in \Psi_0$.

To prove the above, we show that the type system does not equate types with different top level forms when the assumptions in the equivalence context are consistent.

DEFINITION 4.3 (Value types). *A type* τ *is a* value type *if it is of the top level form* $\mathbf{Unit}$, $\Sigma x{:}\sigma_1.\,\sigma_2$, $(x{:}\sigma_1) \rightarrow \sigma_2$, *or* $T\, e$.

LEMMA 4.11 (Value Type Consistency). *If* $\neg\mathbf{incon}(\Delta)$ *and* $\Delta \vdash \tau_1 \equiv \tau_2$, *where* τ_1 *and* τ_2 *are value types, then* τ_1 *and* τ_2 *have the same top-level structure.*

LEMMA 4.12 (Progress). *If* $\cdot \vdash e : \tau$, *then* $\exists e'.\ e \longrightarrow e'$ *or* e *is a value.*

In the proof of this lemma we need $\neg\mathbf{incon}(\cdot)$, so the Canonical Forms Lemma is available. Because $\mathbf{incon}(\Delta)$ is defined in terms of $\mathbf{isEq}$, we have one last requirement on $\mathbf{isEq}$ (Prop 4.10). The empty context must be consistent, *i.e.* if $C_i \neq C_j$ then $\neg\mathbf{isEq}(\cdot, C_i w_i, C_j w_j)$.

A straightforward application of preservation and progress gives us the final result: Well-typed $\lambda^{\cong}$ programs do not get stuck.

THEOREM 4.1 (Type Safety). *If* $\cdot \vdash e : \tau$, *then either there exists a* v *such that* $e \longrightarrow^* v$ *or* e *diverges.*

5. Instantiations

Having identified a set of properties of $\mathbf{isEq}$ that are strong enough to prove type safety, we now examine definitions of term equivalence that satisfy those properties.

It is not hard to see that any instantiation is undecidable: let $\mathbf{isEqX}$ be some instantiation and consider the predicate $\phi(e) = \mathbf{isEqX}(\cdot, e, C_1\, \mathbf{unit})$. The properties require ϕ to be nontrivial (since $\phi(C_1\, \mathbf{unit})$ but $\neg\phi(C_2\, \mathbf{unit})$) and respect β-convertibility, so by a lambda calculus variant of Rice's theorem ([Barendregt 1981] p.144) ϕ is undecidable.

However, we could have a decidable predicate that does not satisfy the $\mathbf{isEq}$ properties but still allows type safety to hold for $\lambda^{\cong}$. Suppose we have an instantiation $\mathbf{isEqX}$, and consider a predicate $\mathbf{isEqX}'$ which is dominated by $\mathbf{isEqX}$, that is if $\mathbf{isEqX}'$ returns true then so does $\mathbf{isEqX}$. Then any program that typechecks using $\mathbf{isEqX}'$ will also typecheck using $\mathbf{isEqX}$, and type safety for $\mathbf{isEqX}$ tells us that the program will never reach a stuck state.

What we are seeing here is the distinction between type safety and preservation/progress. Any predicate that is dominated by one that satisfies the properties is sufficiently weak to ensure type safety, so it is safe to use it in a programming language implementation. Such a predicate will not necessarily be strong enough to typecheck all the intermediate states of a computation.

5.1 Beta-equivalence

Many dependently-typed languages use β-equivalence as the underlying equivalence of the type system. In this section, we show that β-equivalence is indeed a valid instantiation that satisfies the properties in Figure 8.

Call-by-value evaluation Some dependently typed languages test term equivalence by reducing both inputs to a normal form and then comparing, so one expects this algorithm to be a valid instantiation. Indeed it is, although we must adjust the definition slightly: because of nontermination we cannot reduce to normal form, so instead we say that two terms are $\mathbf{isEq}$ if they reduce to *some* common term (not necessarily normal). As a result, the predicate is only semidecidable because we do not know how long to evaluate. Thus we define our first instantiation, called $\mathbf{isEq}_{\longrightarrow}$.

DEFINITION 5.1. *Define* $\mathbf{isEq}_{\longrightarrow}(\Delta, e, e')$ *when there exists* u *such that* $e \longrightarrow^* u$ *and* $e' \longrightarrow^* u$.

LEMMA 5.1. $\mathbf{isEq}_{\longrightarrow}$ *satisfies the* $\mathbf{isEq}$ *properties.*

Note that **isEq**$_{\longrightarrow}$ is the finest equivalence satisfying the properties. Because we require that **isEq** be an equivalence relation which includes $\longrightarrow$, any valid instantiation must identify at least as many terms as **isEq**$_{\longrightarrow}$.

LEMMA 5.2. *Let* **isEqX** *be a predicate which satisfies the* **isEq** *properties. Then* **isEq**$_{\longrightarrow}(\Delta, e, e')$ *implies* **isEqX**(Δ, e, e').

Generalized reduction relations The verification that **isEq**$_{\longrightarrow}$ satisfies the properties does not use many specific facts about $\longrightarrow$. Therefore, we can state a more general result about an arbitrary reduction relation $\rightsquigarrow$.

DEFINITION 5.2. *If $\rightsquigarrow$ is a binary relation between expressions, then define* **isEq**$_{\rightsquigarrow}(\Delta, e_1, e_2)$ *when there exists a u such that $e_1 \rightsquigarrow^* u$ and $e_2 \rightsquigarrow^* u$.*

LEMMA 5.3. *For a given relation on expressions $\rightsquigarrow$, if*
- $\longrightarrow \subseteq \rightsquigarrow$,
- $e \rightsquigarrow e'$ *implies* $e\{w/x\} \rightsquigarrow e'\{w/x\}$,
- $C\, e_0 \rightsquigarrow e'$ *implies that* $e' = C\, e_0'$ *and* $e_0 \rightsquigarrow e_0'$, *and*
- $\rightsquigarrow^*$ *is confluent,*

then **isEq**$_{\rightsquigarrow}$ *satisfies the* **isEq** *properties.*

The added generality of the above lemma shows that type safety is insensitive to the evaluation order used by the type checker. In particular, we can use a parallel reduction relation for $\rightsquigarrow$, where terms are nondeterministically reduced throughout, including underneath function definitions and inside case branches. In fact, there are many valid variants of parallel reduction, based on differences in the β rules.

We identify three variants of parallel reduction below.

$$
\begin{array}{lll}
e \Longrightarrow e' & \textit{Require values in active positions} \\
e \Longrightarrow_w e' & \textit{Require pure terms in active positions} \\
e \Longrightarrow_n e' & \textit{Allow arbitrary reductions}
\end{array}
$$

Surprisingly, all three of these relations are sound, including the last variant which permits β-reductions for arbitrary expressions. This relation allows the type checker to identify $(\lambda x.y)\,\Omega$ and y where Ω is a diverging term—a rather strange fact since these terms are not contextually equivalent under call-by-value evaluation.

However, note that deterministic call-by-name evaluation $\longrightarrow_n$, which never evaluates the argument of an application, is not a valid instantiation. This relation does not contain call-by-value evaluation, so **isEq**$_{\longrightarrow_n}$ does not satisfy the Beta property (4.9). Nevertheless, **isEq**$_{\longrightarrow_n}$ is strictly dominated by **isEq**$_{\Longrightarrow_n}$, which *is* a valid instantiation. This means that even though our language is CBV, it is safe to use CBN evaluation in the type checker.

Expressivity The **isEq**$_{\rightsquigarrow}$ instantiations formally satisfy the properties and highlight the similarities between our system and other dependently-typed languages, but they are of minimal use: our type system relies on introducing equations into the context, but **isEq**$_{\rightsquigarrow}$ does not even look at them! This is only possible because the properties do not force **isEq** to make use of the context; in particular we do not require the following property:

PROPERTY 5.1 (Assumption).
If $e_1 \cong e_2 \in \Delta$ then **isEq**(Δ, e_1, e_2).

This property is not necessary for type safety, so we do not require it. However, it *is* interesting when we consider the expressivity of our type system. In fact, the equivalence assumptions provide all the "dependent" features of our type

system: if the **isEq** instantiation ignores them, we can type no more terms than in the simply typed lambda calculus.

DEFINITION 5.3. *Define a type erasure function $(\cdot)^o$, mapping types τ to simple types, as follows:*

$$
\begin{array}{llll}
(\mathbf{Unit})^o & = \mathbf{Unit} & ((x{:}\tau_1) \to \tau_2)^o & = (\tau_1)^o \to (\tau_2)^o \\
(T)^o & = T & (\Sigma x{:}\tau_1.\,\tau_2)^o & = (\tau_1)^o \times (\tau_2)^o \\
(\tau\, e)^o & = (\tau)^o
\end{array}
$$

$$
(\mathbf{case}\; e \,\langle\, T\, u \,\rangle\, \mathbf{of}\, \{\, \overline{C_i\, x_i \Rightarrow \tau_i}^{\,i} \,\})^o
$$
$$
= \begin{cases} (\tau_i)^o & \textit{if } \mathbf{isEq}(\,\cdot\,,\, e,\, C_i\, w) \\ \mathbf{Unit} & \textit{otherwise} \end{cases}
$$

We write Γ^o to denote the pointwise lifting of the erase operation applied to Γ with all of its equivalence assumptions removed.

LEMMA 5.4 (Erasure). *Suppose that* **isEq**(Δ, e_1, e_2) *implies* **isEq**$(\,\cdot\,, e_1, e_2)$. *Then* $\Gamma \vdash e : \tau$ *implies* $\Gamma^o \vdash_{STLC} e : \tau^o$, *where* $\vdash_{STLC}$ *is the type system for the simply-typed lambda calculus with unit, products and datatypes.*

5.2 Beta-equivalence with assumptions

To extend **isEq**$_{\longrightarrow}$ to a relation satisfying the Assumption Property we can give a direct inductive definition and include enough rules to satisfy the properties:

DEFINITION 5.4 (**isEqFiat**).
Define the relation **isEqFiat**(Δ, e_1, e_2) *as the least relation satisfying the following rules:*

$$
\frac{e_1 \cong e_2 \in \Delta}{\mathbf{isEqFiat}(\Delta, e_1, e_2)} \qquad \frac{e_1 \longrightarrow e_2}{\mathbf{isEqFiat}(\Delta, e_1, e_2)}
$$

$$
\frac{\mathbf{isEqFiat}(\Delta, C\, w_1, C\, w_2)}{\mathbf{isEqFiat}(\Delta, w_1, w_2)}
$$

$$
\frac{}{\mathbf{isEqFiat}(\Delta, e, e)} \qquad \frac{\mathbf{isEqFiat}(\Delta, e_1, e_2)}{\mathbf{isEqFiat}(\Delta, e_2, e_1)}
$$

$$
\frac{\mathbf{isEqFiat}(\Delta, e_1, e_2) \quad \mathbf{isEqFiat}(\Delta, e_2, e_3)}{\mathbf{isEqFiat}(\Delta, e_1, e_3)}
$$

LEMMA 5.5. **isEqFiat** *satisfies the* **isEq** *properties.*

Properties 4.5–4.9 hold for **isEqFiat** by its definition. The properties about substitution and context operations are proved by easy inductions on **isEqFiat**(Δ, e, e'). Finally we get the Empty property for free since when Δ is empty **isEqFiat** coincides with **isEq**$_{\longrightarrow}$.

Just like **isEq**$_{\longrightarrow}$, we can vary the evaluation relation used in the second rule—any relation that works for **isEq**$_{\rightsquigarrow}$ also works for **isEqFiat**. We use the notation **isEqFiat**$_{\rightsquigarrow}$ for alternate versions of this relation.

Like **isEq**$_{\longrightarrow}$, **isEqFiat**$_{\longrightarrow}$ is semidecidable. However, its definition does not suggest a particular algorithm to search for derivations.

5.3 Contextual equivalence with assumptions

In the previous subsections we showed that various β-equivalences are valid instantiations. Our ultimate goal, however, is to find the strongest equivalence we can; then an implementation can use anything weaker than it and be assured of type safety. The natural instantiation to aim for then is contextual equivalence. If we can show that contextual equivalence satisfies the properties, then an implementation will be free to use any known technique in its equivalence-checking algorithm.

Therefore we must state what it means for two terms to be contextually equivalent in the presence of equivalence assumptions. We take as our starting point the notion of CIU-equivalence, which is one of many equivalent definitions of contextual equivalence [Mason and Talcott 1991]. It says that two terms are equivalent if all Closed Instantiations (substitutions of values for free variables) of them have the same termination behavior when Used (placed in a closed evaluation context).

The one subtlety here is what evaluation relation we should consider the termination behavior for. Recall that the type-equivalence rule for **case** will reduce with an open scrutinee $C\,w$, while the operational semantics will only reduce when the scrutinee is a closed value $C\,v$. The **isEq** predicate is part of typechecking, so it is the former behavior that is relevant; for instance we must not identify the stuck terms $C_1\,((\lambda x.x).1)$ and $C_2\,((\lambda x.x).1)$ even though they are contextually equivalent under CBV reduction.

Therefore, we define a "CBW" variant of the evaluation relation, which we write $\longrightarrow_w$. This relation is exactly the same as $\longrightarrow$ except that it replaces all vs with terminal ws. For example, the β rule reads:

$$\frac{w_2 \;\not\longrightarrow_w}{(\,\mathbf{fun}\,f(x)\,=\,e_1\,)\,w_2 \;\longrightarrow_w\; e_1\{w_2/x\}\{\mathbf{fun}\,f(x)\,=\,e_1/f\}}$$

In the definition of contextual equivalence, we use the $\longrightarrow_w$ relation and let the substitutions range over ws.

Note that this subtlety is only for stuck terms. For well-typed terms, it does not matter whether we use $\longrightarrow$ or $\longrightarrow_w$, the same terms will be equated. Therefore, we are justified in considering this a "CBV" contextual equivalence.

DEFINITION 5.5. *Define $e \Downarrow$ if there exists u such that $e \longrightarrow_w^* u$ and not $u \longrightarrow_w u'$ for any u'.*

Now define evaluation contexts in the standard manner.

DEFINITION 5.6 (Evaluation contexts).

$$\begin{aligned}E \;\;::=&\;\;\; \square \;\mid\; E\,e \;\mid\; v\,E \;\mid\; \langle E,\,e \rangle \;\mid\; \langle v,\,E \rangle \;\mid\; E.1 \\ &\mid\;\; E.2 \;\mid\; C\,E \;\mid\; \mathbf{case}\,E\,\mathbf{of}\,\{\,\overline{C_i\,x_i \;\Rightarrow\; e_i}^{\,i}\,\}\end{aligned}$$

DEFINITION 5.7 (CBV Contextual Equivalence). *Define* **isEqC**$(\,e_1,\,e_2\,)$ *iff $\forall E$, $\forall\delta$ such that δ maps variables to ws, if $E\,[\,\delta e_1\,]$ and $E\,[\,\delta e_2\,]$ are closed then $E\,[\,\delta e_1\,] \Downarrow$ iff $E\,[\,\delta e_2\,] \Downarrow$.*

As one might expect, **isEqC** satisfies the **isEq** properties. However, it does not make any use of the equivalence context. The key idea to generalize the definition is to restrict what substitutions should be considered, i.e. if the context contains the equivalence $e_1 \cong e_2$, we should only consider substitutions that make e_1 and e_2 equal. We thus introduce a new judgment $\Delta \vdash \delta$ (pronounced "δ respects Δ") as follows.

DEFINITION 5.8 (Equivalence respecting substitution).

$$\frac{}{\cdot \vdash \delta} \qquad \frac{\Delta \vdash \delta \quad \textbf{isEqC}\,(\,\delta e_1,\,\delta e_2\,)}{\Delta, e_1 \cong e_2 \vdash \delta}$$

We define two expressions to be equivalent under an equivalence context Δ if they have the same behavior for all substitutions that respect the context.

DEFINITION 5.9 (CBV Contextual Equiv. with Assumptions). *Define* **isEqCA**$(\,\Delta,\,e_1,\,e_2\,)$ *iff $\forall E$, $\forall\delta$ such that δ maps variables to ws, if $E\,[\,\delta e_1\,]$ and $E\,[\,\delta e_2\,]$ are closed and $\Delta \vdash \delta$ then $E\,[\,\delta e_1\,] \Downarrow$ iff $E\,[\,\delta e_2\,] \Downarrow$.*
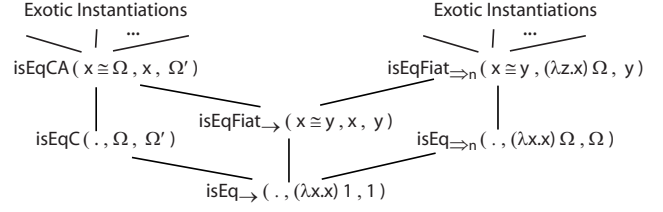

Figure 9. Inclusions between the instantiations

Note that under this definition, if Ω is some nonterminating expression then the equivalence context $x \cong \Omega$ is inconsistent. No δ can reconcile x and Ω, so **isEqCA** will equate all terms compared under this context. Yet this behavior is appropriate for a call-by-value language. We can use it to give arbitrary types to expressions such as $(\lambda x.3)\,\Omega$ or $C\,\Omega$ or $\langle\Omega,\,\texttt{false}\rangle.2$, but all of these expressions will diverge.

LEMMA 5.6. *The relation* **isEqCA** *satisfies the* **isEq** *properties.*

It is straightforward to show that **isEqCA** is an equivalence relation. It also satisfies the Beta Property because $\longrightarrow_w$ includes $\longrightarrow$. The proofs for the rest of the properties rely on two techniques: the proofs of the Weakening, Cut, Substitution and Context Conversion properties follow from proofs of the analogous properties for $\Delta \vdash \delta$; the proofs of the Injectivity and Empty properties follow from carefully constructed evaluation contexts.

Finally, we prove the following lemma to show that **isEqCA** does make use of the assumptions in the Δ context.

LEMMA 5.7. *If $e_1 \cong e_2 \in \Delta$, then* **isEqCA**$(\,\Delta,\,e_1,\,e_2\,)$.

5.4 Exotic Instantiations

The relation **isEqCA** is a strong instantiation, strictly coarser than **isEqFiat**. But it is not the limit—we have already seen that $\textbf{isEq}_{\Longrightarrow_n}$ can safely identify terms that are not contextually equivalent. In fact, the **isEq** properties place very weak restrictions on what terms may be identified, the only negative statements are Empty and Injectivity, and they only apply when both terms are of the form $C\,w$.

Therefore, given a valid **isEq** instantiation, we can create another coarser one by merging two of its equivalence classes, as long as the two classes do not contain pure terms headed by different constructors. For instance, contextual equivalence considers all diverging terms to be equal and certainly no diverging expression is a constructor value, so we can create a coarser instantiation by also saying that any nonterminating term is equal to the integer constant 3 (and all additional equivalences forced by transitivity). Of course, we could also make it equal to 4—but we had better not do both, since then transitivity would make 3 and 4 equal.

This example shows that while there is a weakest valid instantiation, $\textbf{isEq}_{\longrightarrow}$, there is no strongest one. Figure 9 summarizes the ordering of the various instantiations we have discussed as a Hasse diagram.

6. Variations

Different versions of our typing rules lead to different requirements for **isEq**, which in turn affects what instantiations of **isEq** are valid. In this section, we present variations to $\lambda^{\cong}$'s type system, show how they lead to stronger properties for **isEq**, and discuss how that affects instantiations.

6.1 Values, pure terms or terms

A few rules have flexibility about whether some component must be a value, a pure term, or an unrestricted term. Although the last is the most permissive, we have chosen in some cases to restrict to pure terms to weaken the substitution requirement for **isEq**.

For example, consider the type equivalence rule TQ_RED1 in Figure 7. The first precondition requires the scrutinee to be equal to a constructor applied to a pure term. Possible alternatives allow the argument to the constructor to be an arbitrary expression, or require it to be a value.

If we had used an arbitrary expression, then the proof of transitivity in Section 4.2 would require the stronger properties shown below:

PROPERTY 6.1 (Impure Substitution). *If* **isEq** (Δ, e_1, e_2), *then* **isEq** $(\Delta\{e/x\}, e_1\{e/x\}, e_2\{e/x\})$.

PROPERTY 6.2 (Impure Empty). *If* $C_i \neq C_j$, *then* $\neg$**isEq** $(\cdot, C_i e_i, C_j e_j)$.

Unfortunately, instantiations of **isEq** that are based on CBW-evaluation, such as **isEqBeta**$_{\Longrightarrow_w}$ or **isEqCA** do not satisfy these properties because they are not closed under substitution of arbitrary terms. For example, if Ω is a diverging term, then $(\lambda x.z) y$ is equivalent to z under **isEqBeta**$_{\Longrightarrow_w}$ and **isEqCA**, but $(\lambda x.z) \Omega$ is not. Further, although **isEqBeta**$_{\Longrightarrow_w}$ trivially satisfies Impure Empty, **isEqCA** does not; all contexts identify $C_i \Omega$ and $C_j \Omega'$.

Alternatively, if we require the scrutinee to be equivalent to some constructor *value*, i. e. $C_j v$, then we would limit the expressiveness of the type system. For example, the case type

$$\textbf{case } C_1 y \langle T u \rangle \textbf{ of} \{ C_1 x_1 \Rightarrow \textbf{Nat} \mid C_2 x_2 \Rightarrow \textbf{Bool} \}$$

cannot be shown equivalent to **Nat**.

Finally, the syntactic categorization of pure terms in $\lambda^{\cong}$ can be viewed as a very weak and conservative termination analysis. However, unlike Coq or Agda, a complex termination analysis only slightly increases the expressiveness of $\lambda^{\cong}$'s term language, and only in terms of type convertibility. For instance, if $\lambda^{\cong}$ were to use Coq's termination checker in the above example where y is replaced by (*factorial* n), the type would still be equivalent to **Nat**.

6.2 Substitution versus equivalence assumptions

As we discussed in Section 3.2, some of our typing rules diverge from standard practice in that, instead of substitution, they add equivalence assumptions to the context. We have designed our rules in this manner for two reasons. One reason is that we can make the E_CASE rule more expressive by using equations. A second reason is that stating rules with substitution requires a stronger substitution property for **isEq**. With the the alternate E_APP' rule in Section 3.2, **isEq** would need to be closed under the substitution of related expressions inside related expressions.

PROPERTY 6.3 (Equivalent substitution).
If **isEq** (Δ, e_1, e_2), *and* **isEq** (Δ, e, e'),
then **isEq** $(\Delta\{e/x\}, e_1\{e/x\}, e_2\{e'/x\})$.

The reason for this property is the need to show a stronger substitution property for type equivalence $\Delta \vdash \tau\{e_2/x\} \equiv \tau\{e_2'/x\}$ in the case of the preservation lemma when e is an application $e_1 e_2$ and $e_2 \longrightarrow e_2'$. Our previous proof required a weaker lemma that substituted the same pure term throughout the judgment.

We could modify the definitions of **isEqFiat** to satisfy the Equivalent Substitution Property. However, Property 6.3 implies Impure Substitution Property (Property 6.1); therefore, neither **isEq**$_{\longrightarrow}$ nor **isEqCA** satisfies it.

These two examples show two different axes: whether CBW-respecting relations are allowed and whether the equivalence must be stronger than reflexivity for binders, e.g. is $\lambda x.e$ equivalent to $\lambda x.e'$ when e reduces to e'. It is possible to design the type system that interpolates between these two requirements, requiring a "pure equivalent substitution" property, by maintaining the invariant that only ws are ever substituted in terms. Then **isEq**$_{\Longrightarrow_w}$ satisfies the pure equivalent substitution, but **isEq**$_{\longrightarrow}$ does not since it does not reduce under the binder. However, we prefer the simplicity of the current system.

7. Extensions

We have simplified the design of $\lambda^{\cong}$ in a few ways so that we can focus on its novel features. Here, we discuss extensions that would make it more practical as a programming language.

Polymorphism For simplicity, $\lambda^{\cong}$ is not polymorphic. Adding Haskell-style higher-order polymorphism [Jones 1995] would require straightforward changes to the language. Another simple extension is first class-polymorphism, as in Curry-style System F [Girard 1972]. (Note that type checking for Curry-style System F is also undecidable [Wells 1999].) In both cases, type abstraction and application would be implicit as we do not wish to include types in the syntax of terms.

Adding abstractions to the type language, such as in F_ω, would require more significant changes. In particular, our definition of type equivalence would have to be extended to include β-equivalence for these abstractions. A kind-directed specification, which retains the easy inversions of our current definition of type equivalence seems possible, but we leave this extension to future work.

Church-style type system For reasons discussed in Section 2, $\lambda^{\cong}$ does not include typing annotations in expressions. As a result, the type system can assign multiple non-equivalent types to the same expressions. Given the difficulty of complete type inference for dependently-typed languages, a practical source language would include annotations to guide type inference and eliminate ambiguity.

An extension to $\lambda^{\cong}$ with type annotations would take the form of an *external language* that elaborates to and is defined by $\lambda^{\cong}$ typing derivations. This external language would be free to use any type inference technology available for elaboration. As long as elaboration produces valid $\lambda^{\cong}$ typing derivations, this external language is type safe. In particular, ideas from the design of ICC* [Barras and Bernardo 2008] seem relevant.

Type-directed term equivalence Our design decision that the properties of **isEq** should not refer to the type system means that **isEq** cannot receive any typing information from the type checker, such as type annotations embedded in the terms, or the types of the two terms, or a typing context. Therefore, certain type-directed equivalence algorithms [Coquand 1991, Stone and Harper 2000], which use type information to provide stronger extensionality properties, cannot be used for **isEq**. However, in a call-by-value language with nontermination, η-equivalences are restricted: $\lambda x.e\, x$ is not

equivalent to e because e could diverge. Instead, this equivalence only holds for pure terms. Therefore, it is not clear how to extend type-directed equivalences to this setting.

Termination analysis   Because we do not enforce termination, every type in $\lambda^{\cong}$ is inhabited. Therefore interpreting types as logical formulas gives an inconsistent logic—diverging terms are bogus proofs.

Nevertheless, in a CBV language, type safety alone implies useful properties. For example, if a program has type $\Sigma x{:}\mathbf{Nat}.\,\mathbf{lessThan}\,x\,5$ then type safety tells us that if the program terminates the result will be less than 5, because a bogus proof in the second component of the pair would cause the program to diverge. Thus the type serves as a "partial correctness" assertion, although not in the standard sense.

However, this style of reasoning only works for properties that can be witnessed by a first-order data type. If the type contains an implication, e.g. $\Sigma x{:}\mathbf{Nat}.\,(\,\mathbf{lessThan}\,2\,3\,\rightarrow\,\mathbf{lessThan}\,x\,5\,)$, we cannot conclude much since the "proof" may be a function that diverges when applied. Adding a termination analysis (as a separate analysis) would remove these limitations.

Furthermore, a termination analysis would provide a significant source of program optimizations. In a dependently-typed program, many terms are the encodings of proofs that are needed for the program to type check, but otherwise do not affect the actual result of computation. Some languages [Coq Development Team 2009, Barras and Bernardo 2008, Mishra-Linger and Sheard 2008] distinguish between computational and proof terms, allowing the latter to be erased prior to execution. This erasure leads to significant gains in performance.

However, such optimization must not change the termination behavior of the program. In a call-by-value language, computationally irrelevant code can be erased only if it terminates. For example, even if x is not free in e_2, $\mathbf{let}\,x\,=\,e_1\,\mathbf{in}\,e_2$ is only equivalent to e_2 if e_1 is known to terminate. An infinite loop that prevents the program state from reaching a stuck computation should not be removed.

8.   Related work

The past decade has seen much research in the design of dependently-typed programming languages, including Cayenne [Augustsson 1998], Epigram [McBride and McKinna 2004], Ωmega [Sheard 2006], PIE [Vytiniotis and Weirich 2007], DML [Xi and Pfenning 1999], ATS [Xi 2004], DML reformulated [Licata and Harper 2005], GURU [Stump et al. 2009], ConCoqtion [Fogarty et al. 2007], Delphin [Poswolsky and Schürmann 2008], and Ynot [Nanevski et al. 2008]. A number of proof assistants, such as Agda [Norell 2007] and Coq [Coq Development Team 2009], have also successfully been used as dependently-typed languages [Leroy 2006, Oury and Swierstra 2008]. We do not attempt to survey this vast field here. Instead, we only describe aspects of the most related systems.

Parameterized equivalence   Like, $\lambda^{\cong}$, Dependent ML (DML) [Xi and Pfenning 1999] is a family of dependently-typed languages. Types in DML depend not on terms, but on elements of some index language $\mathcal{L}$, a parameter to the system. This constraint language must include booleans and a binary function $\doteq_s$ which must return a boolean for every sort of the language. The constraint relation $\phi; \vec{P} \models P$, which states when proposition P about $\mathcal{L}$ is derivable from assumptions, is likewise a parameter to the system. This relation must satisfy a number of *regularity* rules, somewhat analogous to the **isEq** properties in Figure 8. Xi points out that this constraint relation may be undecidable, but discourages undecidable instances of it.

However, because DML is phase-sensitive, the index language $\mathcal{L}$ is *not* the computation language, and is not computationally relevant. Therefore, there is no analogue of Property IsEqBeta for the constraint relation as the index language is never evaluated. To program in DML, singleton types must be used to make a connection between the index language and computations, leading to redundancy.

Ou et al. [2004] also axiomatize an equivalence judgment on terms in the context of a dependent type system that include general recursion and mutable reference. To ensure decidability of type checking, they restrict the terms in types to only *pure* terms—predefined constants and applications involving pure terms. Therefore, their types may not depend on user-defined functions and their axioms do not include Property IsEq Beta.

Pattern matching with dependent types   Languages that support dependently-typed pattern matching, such as Epigram, Coq and Agda, typically specify the rules for pattern matching using some variant of unification to represent the static information gained during case analysis.

Of these languages, Agda's specification of pattern matching is the most sophisticated [Norell 2007]. Agda uses unification to match the index of the scrutinee's type and the index of the pattern's type. The unification algorithm will simply give up when the unification is hard; for instance, unifying a function application with a term. As a result, Agda's type checking algorithm is not substitutive; unification between a variable y and an arbitrary term always succeeds, however after substituting $f\,x$ for y, the unification algorithm might fail. In our system, instead of solving a unification problem, we add the assumption that the indices are equivalent in the context. Consequently, $\lambda^{\cong}$ is substitutive.

There are some languages that use equivalence assumptions to specify dependently-typed case analysis. A notable example is Altenkirch and Oury's core dependently-typed language $\Pi\Sigma$ [Altenkirch and Oury 2008]. Like $\lambda^{\cong}$, type checking in $\Pi\Sigma$ is undecidable. However, $\Pi\Sigma$ differs from $\lambda^{\cong}$ in that its type system uses β-equivalence as the term equivalence relation.

Likewise, some specifications of *generalized algebraic datatypes* (GADTs, aka guarded recursive datatypes) use equivalence assumptions [Xi et al. 2003, Pottier and Régis-Gianas 2006]. GADTs add index equivalences (but not scrutinee/pattern equivalences) to the context when type checking pattern matching. In these settings, the index language is restricted so that there is an effective algorithm for using these assumptions during type checking. As a result of this restriction, this specification is no more expressive than one that uses unification.

Nontermination in types   Few dependently typed languages allow general recursive functions to appear in types. Cayenne and $\Pi\Sigma$ do, but as far as we know no results, such as type safety, have been proven about them.

Cardelli's Type:Type language [Cardelli 1986] allows nontermination and has a type safety proof based on denotational semantics. However, the proof does not handle case-expressions and dependent elimination. The type system bakes in $\beta\eta$-equivalence as the term equivalence relation.

9. Conclusion

In this paper, we have explored the trade-off between decidable type checking and the complexity of the design of $\lambda^{\cong}$, an expressive, dependently-typed language. Because we have not insisted in the former, we are able to give a simple specification to $\lambda^{\cong}$, despite its advanced features, that permits straightforward, modular proof of type safety. We view this *simplicity* as a contribution of our approach.

The second contribution of our work is the *uniformity* of semantics. Although many different instantiations of **isEq** are valid, we have worked hard to ensure that **isEqCA** is one of them. Therefore, the same semantics can be used to reason about the program both statically and dynamically.

The final contribution of our design is its *generality*. We can view $\lambda^{\cong}$ with **isEqCA** as an ideal goal for the design of a dependently-typed language, much as System F is an ideal model of a polymorphic functional language. Of course, we can never implement a *complete* type checker for $\lambda^{\cong}$ with **isEqCA**; the problem is undecidable. We can however, specify and implement complete type checkers for decidable sublanguages, as any equivalence dominated by **isEqCA** defines a type safe language.

Acknowledgments

References

Thorsten Altenkirch and Nicolas Oury. PiSigma: A core language for dependently typed programming. Draft, 2008.

Lennart Augustsson. Cayenne–a language with dependent types. In *Proc. 3rd ACM Symp. on Principles of Programming Languages (ICFP)*, pages 239–250, 1998.

Brian Aydemir and Stephanie Weirich. LNgen: Tool support for locally nameless representations. Submitted for publication, October 2009.

H. P. Barendregt. *The lambda calculus : Its syntax and semantics*. North-Holland Pub. Co., 1981.

Bruno Barras and Bruno Bernardo. The Implicit Calculus of Constructions as a programming language with dependent types. In *Proc. of the 11th International Conf. on Foundations of Software Science and Computational Structures (FoSSaCS)*, pages 365–379, 2008.

Luca Cardelli. A polymorphic lambda-calculus with type:type. Technical Report SRC Research Report 10, Digital Equipment Corporation Systems Research Center, May 1986.

The Coq Development Team. *The Coq Proof Assistant Reference Manual (Version 8.2)*, 2009. URL http://coq.inria.fr.

Thierry Coquand. An algorithm for testing conversion in Type Theory. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*, pages 255–277, 1991.

Seth Fogarty, Emir Pasalic, Jeremy Siek, and Walid Taha. Concoqtion: indexed types now! In *ACM SIGPLAN 2007 Workshop on Partial Evaluation and Program Manipulation (PEPM)*, pages 112–121, 2007.

Jean-Yves Girard. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Université Paris VII, 1972.

Mark P. Jones. A system of Constructor Classes: Overloading and implicit higher-order polymorphism. *J. Funct. Program.*, 5(1):1–35, 1995.

Xavier Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *Proc. 33rd ACM Symp. on Principles of Programming Languages (POPL)*, pages 42–54, 2006.

Daniel R. Licata and Robert Harper. A formulation of Dependent ML with explicit equality proofs. Technical Report CMU-CS-05-178, Carnegie Mellon University Dept. of Computer Science, 2005.

Ian A. Mason and Carolyn L. Talcott. Equivalence in Functional Languages with Effects. *J. Funct. Program.*, 1(3):287–327, 1991.

C McBride and J McKinna. The view from the left. *J. Funct. Program.*, 14(1):69–111, 2004.

Nathan Mishra-Linger and Tim Sheard. Erasure and polymorphism in Pure Type Systems. In *Proc. of the 11th International Conf. on Foundations of Software Science and Computational Structures (FoSSaCS)*, pages 350–364, 2008.

Aleksandar Nanevski, Greg Morrisett, Avraham Shinnar, Paul Govereau, and Lars Birkedal. Ynot: dependent types for imperative programs. In *Proc. 13th ACM Symp. on Principles of Programming Languages (ICFP)*, pages 229–240, 2008.

Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University, 2007.

Xinming Ou, Gang Tan, Yitzhak Mandelbaum, and David Walker. Dynamic typing with dependent types. In *IFIP International Conference on Theoretical Computer Science*, pages 437–450, 2004.

Nicolas Oury and Wouter Swierstra. The power of Pi. In *Proc. 13th ACM Symp. on Principles of Programming Languages (ICFP)*, pages 39–50, 2008.

Simon L. Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. Simple unification-based type inference for GADTs. In *Proc. 11th ACM Symp. on Principles of Programming Languages (ICFP)*, pages 50–61, 2006.

Adam Poswolsky and Carsten Schürmann. Practical programming with higher-order encodings and dependent types. In *Proc. of the 17th European Symp. on Programming (ESOP)*, pages 93–107, 2008.

François Pottier and Yann Régis-Gianas. Stratified type inference for generalized algebraic data types. In *Proc. 33rd ACM Symp. on Principles of Programming Languages (POPL)*, pages 232–244, 2006.

Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar, and Rok Strniša. Ott: Effective Tool Support for the Working Semanticist. In *Proc. 12th ACM Symp. on Principles of Programming Languages (ICFP)*, pages 1–12, 2007.

T. Sheard. Type-level computation using narrowing in Ωomega. In *The 1st workshop on Programming Languages meets Program Verification (PLPV)*, pages 105–128, 2006.

Chris Stone and Robert Harper. Deciding type equivalence in a language with singleton kinds. In *Proc. 27th ACM Symp. on Principles of Programming Languages (POPL)*, pages 214–227, 2000.

Aaron Stump, Morgan Deters, Adam Petcher, Todd Schiller, and Timothy Simpson. Verified programming in Guru. In *Proc. of the 3rd workshop on Programming Languages meets Program Verification (PLPV)*, pages 49–58, 2009.

Dimitrios Vytiniotis and Stephanie Weirich. Dependent types: Easy as PIE. In *8th Symp. on Trends in Functional Programming*, 2007.

Joe B. Wells. Typability and type checking in System F are equivalent and undecidable. *Annals of Pure and Applied Logic*, 98(1–3):111–156, 1999.

Andrew K. Wright and Matthias Felleisen. A Syntactic Approach to Type Soundness. *Information and Computation*, 115:38–94, 1994.

Hongwei Xi. Applied type system. In *Proceedings of TYPES 2003*, Lecture Notes in Computer Science, pages 394–408. 2004.

Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *Proc. 26th ACM Symp. on Principles of Programming Languages (POPL)*, pages 214–227, 1999.

Hongwei Xi, Chiyan Chen, and Gang Chen. Guarded recursive datatype constructors. In *Proc. 30th ACM Symp. on Principles of Programming Languages (POPL)*, pages 224–235, 2003.