# Dependent Interoperability

Peter-Michael Osera    Vilhelm Sjöberg    Steve Zdancewic
University of Pennsylvania
{posera, vilhelm, stevez}@cis.upenn.edu

## Abstract

In this paper we study the problem of *interoperability*—combining constructs from two separate programming languages within one program—in the case where one of the two languages is dependently typed and the other is simply typed. We present a core calculus called SD, which combines dependently- and simply-typed sublanguages and supports user-defined (dependent) datatypes, among other standard features. SD has "boundary terms" that mediate the interaction between the two sub-languages. The operational semantics of SD demonstrates how the necessary dynamic checks, which must be done when passing a value from the simply-typed world to the dependently typed world, can be extracted from the dependent type constructors themselves, modulo user-defined functions for marshaling values across the boundary. We establish type-safety and other meta-theoretic properties of SD, and contrast this approach to others in the literature.

***Categories and Subject Descriptors*** D.3.1 [*Programming Languages*]: Formal Definitions and Theory — Semantics; D.2.12 [*Interoperability*]: Data mapping

***General Terms*** Languages, Theory

***Keywords*** Dependent types, language interoperability, contracts

## 1. Introduction

Dependently-typed languages allow programmers to specify a rich set of properties about their programs that are verifiable during type-checking. This comes at the price of complexity — it is at best extremely time-consuming and at worse infeasible to use dependently-typed languages in large software developments. A natural way to mitigate this weakness is to use a dependently-typed language to provide specifications for critical components while the rest of the system is written in a mainstream programming language. However, care must be taken to ensure that the specifications of the dependently-typed language are respected by "weaker" programming language. In this paper, we study the problem of interoperability between a language with dependent types and a language with simple types, focusing on the key meta-theoretic issues that arise in this setting.

Prior work on interoperability initially focused on the implementation of such interoperability systems. Many languages provide an escape hatch into C, such as Java's JNI [15], or OCaml's [13] and Haskell's [17] FFI. Other work considers how to achieve interoperability by developing a *lingua franca* for languages to talk to each other. Proposals include C [2], the Java virtual machine [16], COM [26], or the .NET framework [30]. More recently, the focus has shifted to understanding the relationship between dynamic and typed languages with contracts [7], blame [33], and the integration of scripting and typed languages [34].

In these systems, dynamic checks ensure that the static guarantees of the typed language are respected by the untyped language. The dynamic check amounts to a simple type tag check, e.g., verifying that $\mathsf{typeof}\ (\lambda x{:}\,S.s)$ is indeed a function. However, the same concerns arise if we consider languages with richer type systems, namely those with dependent types. A simply-typed language will be able to enforce only some of a dependently-typed language's static guarantees during type-checking; the difference must again be made up with dynamic checks. However these dynamic checks must now perform non-trivial computation rather than simply checking type tags.

For example, suppose that your dependently-typed language provides a certified library that you would like to use in your application. For simplicity's sake, let's consider a List datatype that contains Ints.

$$\begin{aligned}&\mathsf{List} : \mathsf{Int} \Rightarrow *\\ &\mathsf{Nil} : (y{:}\mathsf{Unit}) \to \mathsf{List}\ y\\ &\mathsf{Cons} : (y_1{:}\mathsf{Int}) \to (y_2{:}\mathsf{Int}) \to \mathsf{List}\ y_1 \to \mathsf{List}\ y_1 + 1\end{aligned}$$

List is indexed by an integer than represents its length, and that invariant is maintained by its two constructors Nil and Cons. Suppose that our library also has a dependently-typed function $\mathsf{PrettyPrintList5} : \mathsf{List}\ 5 \to \mathsf{Unit}$ that prints out lists of length five in a special way, but instead of giving it a dependently-typed List, we'd like to provide it our standard simply-typed List instead. Our interoperability layer must not only marshal the List value between languages, but also ensure that the simply-typed List has length five.

### 1.1 Contributions and Outline

How do we craft an interoperability layer that can generate such dynamic checks? How does such an interoperability layer affect the meta-theoretic properties of the languages involved? In order to answer these questions, we propose a calculus in the style of Matthews and Findler [19] that combines two languages together — in our case, a simply-typed and dependently-typed language — via boundary terms.

Our work on dependent interoperability contributes the following:

1. A core calculus called SD that combines a simply-typed and dependently-typed lambda calculus extended with user-defined datatypes. While we are aware of previous efforts to combine simply-typed and dependently-typed programming, to our knowledge, this is the first work that looks at the problem from the perspective of language interoperability with the cor-

*PLPV'12,* January 24, 2012, Philadelphia, PA, USA.
Copyright © 2012 ACM 978-1-4503-1125-0-12-01. . . $10.00

responding aim of modifying the languages as little as possible when integrating them.

2. Analysis of the meta-theoretic properties of SD, in particular, a proof of type safety for the language.
3. Exploration of the design space of dependent interoperability, including changes to the design to guarantee termination in the presence of recursive functions and alternatives to directly translating data.
4. A comparison of our system to real world systems such as Coq and Agda that provide limited forms of language interoperability. Such comparisons strengthen our claim that our model faithfully captures dependent interoperability, but also suggests how these real world systems can improve in this area.

We open in Section 2 by expanding on the benefits of dependent interoperability. In Section 3, we describe the syntax and semantics of SD. We discuss the metatheory of SD in Section 4. Next we describe additional interesting properties of SD in Section 5. In Section 6 we compare SD to real world dependently-typed systems that offer interoperability facilities. Finally we discuss related and future work in Section 7 and close in Section 8.

## 2. Motivation

Before we discuss SD proper, we first motivate further why dependent interoperability is a useful idea by discussing three use cases in more detail. Along the way we will foreshadow the potential difficulties in creating an interoperability layer that we will solve in Section 3.

1. **Using a simply-typed library in a dependently-typed context.** While our dependently-typed language may be safer to use, it will typically not have all the functionality we would like. For example, we may wish to use a simply-typed library that provides network access, e.g., a function $\mathsf{sendData} : \mathsf{Packet} \to \mathsf{Unit}$, from our dependently-typed program. It is a good bet (although not always true) that our dependent type system is strictly more powerful than the simple type system, so intuition tells us that we shouldn't need any dynamic checks here. Therefore, our interop boundary needs only to marshal the data from the dependently-typed language into the Packet that the simply-typed function expects to use.
2. **Using a dependently-typed library in a simply-typed context.** The dual of the previous use case is the desire to use dependently-typed code in a simply-typed context. In the introduction, we used the toy example of a $\mathsf{List}\ n$. However, you can imagine wanting to use a verified library for a particular data structure or protocol from a simply-typed context and be assured that the simply-typed data you feed it does not break the properties the verified library enforces. Discovering and enforcing these properties is the primary challenge our interoperability boundary faces.
3. **Verifying properties of simply-typed code**. Finally, because we are working with a dependently-typed language, an interesting question arises. In addition to verifying properties of dependently-typed terms, can we do the same with simply-typed terms? That is, rather than implement a verified library in the dependently-typed language and translating simply-typed data into that library, we would like to verify properties of a simply-typed library directly. Ideally the dependently-typed language would be able to do this all during typechecking, but realistically, complete checking of a term across an interop boundary is impossible. We expect that the result is similar to a hybrid type system [8] where some properties are verified during compilation and the rest are "made up" with dynamic checks.

| | $\lambda^{\to}$ | $\lambda^{\cong}$ |
|---|---|---|
| Kinds | | $K$ |
| Types | $S$ | $T$ |
| Terms | $s$ | $t$ |
| Variables | $x$ | $y$ |
| Datatypes | $A$ | $B$ |

**Figure 1.** Metavariable Conventions for $\lambda^{\to}$ and $\lambda^{\cong}$

| Judgment | Description |
|---|---|
| $\Gamma \vdash s : S$ | $\lambda^{\to}$ Typing |
| $\Gamma \vdash K$ | $\lambda^{\cong}$ Well-formed Kinds |
| $\Gamma \vdash T : K$ | $\lambda^{\cong}$ Kinding |
| $\Gamma \vdash t : T$ | $\lambda^{\cong}$ Typing |
| $\vdash \Psi$ | Well-formed Signature |
| $\vdash \Gamma$ | Well-formed Context |
| $\mathsf{FO}\,(T)$ | First-order Type |
| $S \Leftrightarrow T$ | Type Translation |
| $\Gamma \vdash K \equiv K'$ | $\lambda^{\cong}$ Kind Equivalence |
| $\Gamma \vdash T \equiv T'$ | $\lambda^{\cong}$ Type Equivalence |
| $\Gamma \vdash t \cong t'$ | $\lambda^{\cong}$ Term Equivalence |
| $s \longrightarrow s'$ | $\lambda^{\to}$ Evaluation |
| $t \longrightarrow t'$ | $\lambda^{\cong}$ Evaluation |

**Figure 2.** SD Judgments

## 3. Language

Our language SD consists of a *simply-typed* and a *dependently-typed* lambda calculus joined together by boundary terms in the style of Matthews and Findler [19]. Throughout this paper, we use a meta-variable convention to distinguish terms of the simply-typed fragment ($\lambda^{\to}$) and the dependently-typed fragment ($\lambda^{\cong}$) outlined in Figure 1. In addition, there are several judgments that make up SD. In the interest of the brevity, we only present the salient features of each of these judgments. The extended version of our paper [23] contains the complete definitions of our system along with proofs.

### 3.1 Syntax

$\lambda^{\to}$ is a standard lambda calculus with simple types as defined in Figure 3. We augment the calculus with pairs $<s_1, s_2>$, unit, an error term that will be raised if a boundary check fails, and user-defined data constructors $C$ with corresponding datatypes $A$. Constructors are modeled as taking only a single argument but this is not a limitation since multiple arguments can be combined using pairs. For example, the constructor $\mathsf{Cons}^{\to}$ has type

$$\mathsf{Cons}^{\to} : (\mathsf{List} * \mathsf{Int}) \to \mathsf{List}.$$

In SD we presuppose a signature $\Psi_0$ containing the definitions of these constructors.

The notable addition to $\lambda^{\to}$ is the addition of the typed boundary term $\mathsf{SD}^{S}_{T}\,t$ which can be read as an interoperability boundary that translates the inner $\lambda^{\cong}$ term $t$ of type $T$ to a $\lambda^{\to}$ term of type $S$. Such boundaries are responsible for *marshaling* data from one side of the boundary to the other and *checking* that this marshaled data is appropriate for the context it will be used in. Our formulation focuses on understanding the latter responsibility: what checks are necessary to ensure type-safety when moving across boundaries?

$\lambda^{\cong}$ is a standard dependently-typed lambda calculus inspired Jia et al's system "Lambda-eek" [12]. The syntax of $\lambda^{\cong}$ as given in Figure 3 mirrors the syntactic forms found in $\lambda^{\rightarrow}$: it has dependent functions and pairs along with $\mathsf{unit}$ and $\mathsf{error}$. The types of dependent functions and pairs are written $(y\colon T_1) \rightarrow T_2$ and $(y\colon T_1) * T_2$ reflecting the fact that $T_2$ in both cases may contain the bound term variable $y$. A datatype $B$ is now a type-level function that, given a term $t$, produces a type $B\ t$. Consequently, we introduce kinds to classify such type-level functions $T \Rightarrow *$, versus proper kinds $*$.

Constructors in $\lambda^{\cong}$ also take single arguments. Combining multiple arguments using pairs is trickier because of dependent types, but still manageable. For example, the type of dependent $\mathsf{Cons}^{\cong}$ is

$$\mathsf{Cons}^{\cong} : (y_1 : (y_2 : \mathsf{Int}) * (\mathsf{List}\, y_2 * \mathsf{Int})) \rightarrow \mathsf{List}\,(y.1) + 1$$

In effect, we use dependent pairs to introduce additional arguments and then project out the arguments when needed to compute the index of the datatype.

In the interest of simplifying the syntax, the introduction forms for the different constructs are shared between $\lambda^{\rightarrow}$ and $\lambda^{\cong}$. This is not problematic as we can look at a term's sub-terms to determine which syntactic category it belongs to. In particular, the names of constructors $C$ are shared between the two calculi, with the implicit assumption that each constructor has $\lambda^{\rightarrow}$ and $\lambda^{\cong}$ counterparts. This simplifies our reasoning when dealing with translating constructors, as we only need to worry about translating the arguments of the constructor.

We introduce a guard term $t_1 \cong t_2 \rhd t_3$ that is the result of reducing a boundary term $\mathsf{DS}^{T}_{S}\, s$. This guard term makes explicit the equivalence check that must occur before we create the marshaled term $t$ from $s$. In our presentation of SD, the only check we need is an equivalence check $t_1 \cong t_2$ that determines whether two $\lambda^{\cong}$ terms are indeed equivalent at runtime.

The attentive reader may notice that guards appear only on the $\lambda^{\cong}$ side of the boundary. Intuitively this is because the types of $\lambda^{\cong}$ make strictly stronger guarantees than $\lambda^{\rightarrow}$. When going from $\lambda^{\cong}$ to $\lambda^{\rightarrow}$, no checks are necessary because the $\lambda^{\cong}$ type system can verify all the properties that the $\lambda^{\rightarrow}$ type system tries to enforce. Conversely, $\lambda^{\rightarrow}$ cannot make such guarantees, so we make up the difference on the $\lambda^{\cong}$ side with dynamic checks in the form of our guards.

In both $\lambda^{\rightarrow}$ and $\lambda^{\cong}$ we introduce $\mathsf{let}$ forms as the standard syntactic sugar over abstraction binding.

$$\mathsf{let}\, x = s_1 \,\mathsf{in}\, s_2 \triangleq (\lambda x{:}\, S_1.s_2)\, s_1$$
$$\mathsf{let}\, y = t_1 \,\mathsf{in}\, t_2 \triangleq (\lambda y{:}\, T_1.t_2)\, t_1$$

However, in $\lambda^{\rightarrow}$ we also add the special let binding $\mathsf{letd}\ y = t \,\mathsf{in}\, s$ that crosses from $\lambda^{\rightarrow}$ to $\lambda^{\cong}$ to bind a $\lambda^{\cong}$ term and then returns to evaluate $s$. This form is used in order to avoid duplication of side-effects during evaluation. We discuss $\mathsf{letd}$ in more detail when we talk about the evaluation rules of SD.

## 3.2 Typing and well-formedness

The typing rules for the $\lambda^{\rightarrow}$ fragment are entirely standard, so we do not reproduce them in their entirety here. The only interesting addition is WF_STM_SD, which gives a type to our boundaries $\mathsf{SD}^{S}_{T}\, t$. A boundary is well-typed if the contained $\lambda^{\cong}$ term meets the type annotation on the boundary, and if the types on the boundary are compatible, written $S \Leftrightarrow T$. Figure 4 gives these rules.

Our type compatibility relation ensures that we can translate between data of the given types. For compound types such as arrows and pairs, we can translate between them if we can translate between their component types. Translating between $\mathsf{Unit}$ types is trivial. And since datatypes $A$ and $B$ are user-defined, we appeal to user-defined translations between them represented by the meta-function $\mathsf{corr}\,(A, B)$. As a concrete example, it is reasonable to expect that the $\mathsf{List}$ datatypes between the $\lambda^{\rightarrow}$ and $\lambda^{\cong}$ fragments are convertible so that we have $\mathsf{corr}\,(\mathsf{List}^{\rightarrow}, \mathsf{List}^{\cong})$. Note that $S \Leftrightarrow T$ strips away the term-components of a dependent type—it compares types only up to the simply-typed "skeleton". However, compatibility does require that the types of the indices of dependent data are first order, written $\mathsf{FO}\,(T)$. Intuitively, $\mathsf{FO}\,(T)$ means that the type $T$ does not contain any arrows. If we did allow arrows here, then when translating such datatypes we would be forced to compare equality of function values, which is a hard problem. This will become clear in Section 3.3 where we discuss the evaluation rules of SD. Note that the data that we are translating is allowed to contain functions, but the index of that datatype is not.

For $\lambda^{\cong}$ we present several of the kinding and typing rules in Figure 5 to remind the reader of the intricacies of dependent type systems and foreshadow the technical challenges of translating terms into these types during evaluation.

All programs are typed with respect to some fixed signature $\Psi_0$, which assigns types to constructors $C$ and kinds to datatypes $A$ and $B$. We assume that all the types and kinds in $\Psi_0$ are well-formed in the empty context. Because datatypes are type-level functions, we assign them kinds of the form $T_1 \Rightarrow *$, as shown in WF_DTY_DATA, while the remaining types have kind $*$, e.g., WF_DTY_ARR.

Rules WF_DTM_APP and WF_DTM_PAIR illustrate the dependent nature of abstraction and pairs in $\lambda^{\cong}$. The second component $T_2$ of the types may contain free occurrences of $y$ of type $T_1$, so we must close $T_2$ by substituting for $y$. WF_DTM_CONV is the standard conversion rule that allows us to take advantage of indexed types by establishing equivalences between them (via the type-equivalence judgment $\Gamma \vdash T \equiv T'$ as discussed in the next section). With WF_DTM_CTOR, we type a constructor $C$ at some datatype $B\,[t/y]t'$ where we substitute into the term the argument given to $C$. Note that the type of the argument to $C$ does not need to coincide with the type of the index of $B$. Finally when we type cases with WF_DTM_CASE in each branch we remember the refined type $B\ t_i'$ of the branch's associated constructor.

Checking $\mathsf{DS}$ via WF_DTM_DS is analogous to $\mathsf{SD}$ boundaries: the inner term must typecheck and the type annotations must coincide. WF_DTM_GUARD typechecks guards by checking to see if the types involved in the equivalence check are well-typed. In addition, $t$ must be well-typed under the assumption that the check holds. Finally, we require that the types of the guard are first-order with the judgment $\mathsf{FO}\,(T)$. The first-order judgment ensures that the types of guards are never arrows so that we do not have to determine the equivalence of functions.

The judgment $\mathsf{FO}\,(T)$ ensures that the inhabitants of $T$ do not contain function values. In the case of FO_DATA we check that all constructors of $B$ take first-order arguments. We do not need to check that the type of $B$'s index term $t_i$ is first-order, since the index is not part of the values inhabiting $B$.

## 3.3 Evaluation

The evaluation rules of SD are of most interest to us because this is where we do the actual work of checking values and marshaling them across boundaries. Figure 6 gives the syntax of our one-step evaluation contexts which define the standard call-by-value order for our language. In addition, Figure 6 also lists the interesting evaluation rules for both languages.

The evaluation of the usual syntactic forms — abstractions, pairs, and constructors — are standard. The interesting rules arise from evaluation of boundary terms. In both languages, the evaluation of boundaries is directed by their type annotations, so there is one rule for each value that might be sent across a boundary.

When we translate lambdas, e.g., a $\lambda^{\rightarrow}$ lambda to a $\lambda^{\cong}$ lambda as in EVAL_STM_DS_ABS, the output must be a $\lambda^{\cong}$ lambda. Our translation is similar to Matthews' and Findler's. This new $\lambda^{\cong}$

$$
\begin{array}{llcl}
\lambda^{\rightarrow}\ \text{Types} & S & ::= & S_1 \rightarrow S_2 \mid S_1 * S_2 \mid \mathsf{Unit} \mid A \\
\lambda^{\rightarrow}\ \text{Terms} & s & ::= & x \mid \lambda x{:}S.s \mid s_1\, s_2 \\
 & & \mid & <s_1, s_2> \mid s.1 \mid s.2 \\
 & & \mid & C\, s \mid \mathsf{case}\, s\, \mathsf{of}\, \overline{C_i\, x_i \rightarrow s_i}^{\,i} \\
 & & \mid & \mathsf{unit} \mid \mathsf{error} \mid \mathsf{letd}\, y = t\, \mathsf{in}\, s \mid \mathsf{SD}^{S}_{T} t
\end{array}
\qquad
\begin{array}{llcl}
\lambda^{\cong}\ \text{Kinds} & K & ::= & * \mid T \Rightarrow * \\
\lambda^{\cong}\ \text{Types} & T & ::= & (y{:}T_1) \rightarrow T_2 \mid T\, t \\
 & & \mid & (y{:}T_1) * T_2 \mid \mathsf{Unit} \mid B \\
\lambda^{\cong}\ \text{Terms} & t & ::= & y \mid \lambda y{:}T.t \mid t_1\, t_2 \\
 & & \mid & <t_1, t_2> \mid t.1 \mid t.2 \\
 & & \mid & C\, t \mid \mathsf{case}\, t\, \mathsf{of}\, \overline{C_i\, y_i \rightarrow t_i}^{\,i} \\
 & & \mid & \mathsf{unit} \mid \mathsf{error} \\
 & & \mid & \mathsf{DS}^{T}_{S} s \mid t_1 \cong t_2 \rhd t_3
\end{array}
$$

**Figure 3.** SD Syntax

$\boxed{\Gamma \vdash s : S}$

$$
\frac{\begin{array}{c}\Gamma \vdash t : T \\ S \Leftrightarrow T\end{array}}{\Gamma \vdash \mathsf{SD}^{S}_{T} t : S}\ \text{WF\_STM\_SD}
$$

$\boxed{S \Leftrightarrow T}$

$$
\frac{\begin{array}{c}S_1 \Leftrightarrow T_1 \\ S_2 \Leftrightarrow T_2\end{array}}{S_1 \rightarrow S_2 \Leftrightarrow (y{:}T_1) \rightarrow T_2}\ \text{COMPAT\_ARR}
\qquad
\frac{\begin{array}{c}S_1 \Leftrightarrow T_1 \\ S_2 \Leftrightarrow T_2\end{array}}{S_1 * S_2 \Leftrightarrow (y{:}T_1) * T_2}\ \text{COMPAT\_PAIR}
$$

$$
\frac{}{\mathsf{Unit} \Leftrightarrow \mathsf{Unit}}\ \text{COMPAT\_UNIT}
\qquad
\frac{\begin{array}{l}B{:}T_0 \Rightarrow * \in \Psi_0 \\ \mathsf{FO}\,(T_0) \\ \mathsf{corr}\,(A, B)\end{array}}{A \Leftrightarrow B\, t}\ \text{COMPAT\_DATA}
$$

$\boxed{\mathsf{FO}\,(T)}$

$$
\frac{\mathsf{FO}\,(T)}{\mathsf{FO}\,(T\, t)}\ \text{FO\_APP}
\qquad
\frac{}{\mathsf{FO}\,(\mathsf{Unit})}\ \text{FO\_UNIT}
\qquad
\frac{\begin{array}{c}\mathsf{FO}\,(T_1) \\ \mathsf{FO}\,(T_2)\end{array}}{\mathsf{FO}\,((y{:}T_1) * T_2)}\ \text{FO\_PAIR}
\qquad
\frac{\begin{array}{l}\mathsf{constrs}\, B = \overline{C_i}^{\,i} \\ \overline{C_i{:}(y_i : T_i) \rightarrow B\, t_i' \in \Psi_0}^{\,i} \\ \overline{\mathsf{FO}\,(T_i)}^{\,i}\end{array}}{\mathsf{FO}\,(B\, t)}\ \text{FO\_DATA}
$$

**Figure 4.** Abridged $\lambda^{\rightarrow}$ Typing Rules, Type Compatibility, and First-order Types

lambda translates its argument $y$ to $\lambda^{\rightarrow}$, supplies that translated argument to the $\lambda^{\rightarrow}$ lambda, and translates the $\lambda^{\rightarrow}$ result of the application back to $\lambda^{\cong}$.

In the DS case this is straightforward. However, if we look at the SD case as presented in EVAL_DTM_SD_ABS, we note that $T_2$ may contain free occurrences of $y$ in the boundary. To fix this problem, we close $T_2$ with the $\lambda^{\rightarrow}$ lambda's translated argument. Thus, boundary type annotations are not simple annotations that can be erased at compile time. They are entities that affect evaluation, so they must have a concrete representation at runtime. Note that the DS case does not need a substitution due to our choice of creating a $\lambda^{\cong}$ lambda that implicitly captures the free variable found in $T_2$.

This observation that the second type component $T_2$ needs to be closed via a substitution is also applicable when translating pairs. In the EVAL_STM_SD_PAIR case the sub-components are already $\lambda^{\cong}$ terms, so we simply close $T_2$ with $v_1$. In the EVAL_DTM_DS_PAIR case, $u_1$ is a $\lambda^{\rightarrow}$ term, so we need to translate it before substituting into $T_2$. So as a first attempt, we might make the term step to $< \mathsf{DS}^{T_1}_{S_1} u_1, \mathsf{DS}^{[\mathsf{DS}^{T_1}_{S_1} u_1 / y]\, T_2}_{S_2} u_2 >$. However, that proposal has a different problem: $\mathsf{DS}^{T_1}_{S_1} u_1$ is not a value! In particular, while $u_1$ itself is a value, $T_1$ may contain non-value terms. By duplicating this expression, we potentially duplicate any of its side-effects.

To avoid this, in EVAL_DTM_DS_PAIR we let-bind the first component of the translated pair. This sequences the evaluation at runtime and avoids duplicating side-effects. Similarly, in EVAL_STM_SD_ABS we let-bind the translated argument $x$. However, an interesting technicality arises. The point at which we need to let-bind the argument — which is a $\lambda^{\cong}$ term — lies in $\lambda^{\rightarrow}$! To fix this issue, we use the letd construct that allows us to bind a value in $\lambda^{\cong}$ and then evaluate a $\lambda^{\rightarrow}$ term. In this context, letd has a natural interpretation: letd goes into $\lambda^{\cong}$ to bind a term in the environment, returns back to $\lambda^{\rightarrow}$, and evaluates as normal.

The translation of datatypes is more involved because, in addition to variable capture, we must also check that the translation "respects" the property represented by the datatype's index. For example, in the case of List, a reasonable translation from a $\mathsf{List}^{\rightarrow}$ to $\lambda^{\cong}$ should produce a $\mathsf{List}^{\cong}\ t$ where $t$ is the length of the list. In general, what the translation should do is dependent on the datatypes we are translating.

Thus, in addition to presupposing user-defined constructors $C$ of datatypes $A$ and $B\ t$, we also presuppose user-defined *conver-*

$\boxed{\Gamma \vdash K}$

$$\frac{}{\Gamma \vdash *}\,\text{WF\_DKN\_PROPER} \qquad \frac{\Gamma \vdash T : *}{\Gamma \vdash T \Rightarrow *}\,\text{WF\_DKN\_ARR}$$

$\boxed{\Gamma \vdash T : K}$

$$\frac{\begin{array}{l}\Gamma \vdash T_1 : * \\ \Gamma, y{:}T_1 \vdash T_2 : *\end{array}}{\Gamma \vdash (y : T_1) \to T_2 : *}\,\text{WF\_DTY\_ARR} \qquad \frac{B{:}T \Rightarrow * \in \Psi_0}{\Gamma \vdash B : T \Rightarrow *}\,\text{WF\_DTY\_DATA}$$

$\boxed{\Gamma \vdash t : T}$

$$\frac{\begin{array}{l}\Gamma \vdash t_1 : (y : T_1) \to T_2 \\ \Gamma \vdash t_2 : T_1 \\ \Gamma \vdash [t_2/y]\, T_2 : *\end{array}}{\Gamma \vdash t_1\, t_2 : [t_2/y]\, T_2}\,\text{WF\_DTM\_APP} \qquad \frac{\begin{array}{l}\Gamma \vdash t_1 : T_1 \\ \Gamma \vdash t_2 : [t_1/y]\, T_2 \\ \Gamma \vdash (y : T_1) * T_2 : *\end{array}}{\Gamma \vdash <t_1, t_2>: (y : T_1) * T_2}\,\text{WF\_DTM\_PAIR}$$

$$\frac{\Gamma \vdash t : (y : T_1) * T_2}{\Gamma \vdash t.1 : T_1}\,\text{WF\_DTM\_PROJ1} \qquad \frac{\begin{array}{l}\Gamma \vdash t : (y : T_1) * T_2 \\ \Gamma \vdash [t.1/y]\, T_2 : *\end{array}}{\Gamma \vdash t.2 : [t.1/y]\, T_2}\,\text{WF\_DTM\_PROJ2}$$

$$\frac{\begin{array}{l}C{:}(y : T_1) \to B\, t' \in \Psi_0 \\ B{:}T_2 \Rightarrow * \in \Psi_0 \\ \Gamma \vdash t : T_1 \\ \Gamma \vdash B\, [t/y] t' : *\end{array}}{\Gamma \vdash C\, t : B\, [t/y] t'}\,\text{WF\_DTM\_CTOR} \qquad \frac{\begin{array}{l}\Gamma \vdash t : B\, t' \\ \Gamma \vdash T : * \\ \mathsf{constrs}\, B = \overline{C_i}^{\,i} \\ \overline{C_i{:}(y_i : T_i) \to B\, t'_i \in \Psi_0}^{\,i} \\ \overline{\Gamma, y_i{:}T_i, t' \cong t'_i, t \cong C_i\, y_i \vdash t_i : T}^{\,i}\end{array}}{\Gamma \vdash \mathsf{case}\, t\, \mathsf{of}\, \overline{C_i\, y_i \to t_i}^{\,i} : T}\,\text{WF\_DTM\_CASE}$$

$$\frac{\begin{array}{l}\Gamma \vdash s : S \\ \Gamma \vdash T : * \\ S \Leftrightarrow T\end{array}}{\Gamma \vdash \mathsf{DS}_S^T\, s : T}\,\text{WF\_DTM\_DS} \qquad \frac{\begin{array}{l}\Gamma \vdash t_0 : T_0 \\ \Gamma \vdash t_1 : T_0 \\ \mathsf{FO}\,(T_0) \\ \Gamma, t_1 \cong t_0 \vdash t : T\end{array}}{\Gamma \vdash t_1 \cong t_0 \rhd t : T}\,\text{WF\_DTM\_GUARD} \qquad \frac{\begin{array}{l}\Gamma \vdash t : T \\ \Gamma \vdash T \equiv T' \\ \Gamma \vdash T' : *\end{array}}{\Gamma \vdash t : T'}\,\text{WF\_DTM\_CONV}$$

**Figure 5.** Abridged $\lambda^{\cong}$ Typing Rules

*sions between arguments* of constructors, with the intent that these conversions preserve the dependent datatype's properties. These conversions come as a pair of functions

$$\begin{array}{r}\mathsf{argToS}_C\, v = u \\ \mathsf{argToD}_C\, u = v\end{array}$$

responsible for converting constructor arguments from one language to the other. At type-checking time, the arguments $v$ and $u$ could contain free variables making it unclear how to translate them, so we allow argToS and argToD to be partial functions. When they are undefined the corresponding boundary term is stuck. To ensure Progress, we require that they are always defined for closed well-typed values. We also require some additional conditions expressing that they are defined "naturally" in the argument that we discuss further in Section 4.3.

argToS and argToD can be viewed constructor-indexed user-level functions which, if $C{:}S \to A \in \Psi_0$, $C{:}(y : T_1) \to B\, t \in \Psi_0$, and $B{:}T_2 \Rightarrow * \in \Psi_0$, have the types

$$\begin{array}{r}\mathsf{argToS} : T_1 \to S \\ \mathsf{argToD} : S \to T_1.\end{array}$$

We distinguish them from user-level functions because as we have defined the calculus there is no way to form such mixed types. Also, in addition to their types, we intend that the functions are inverses. That is, the following equations should hold

1. $(\mathsf{argToS} \circ \mathsf{argToD})(u) = u$ with $u : S$
2. $(\mathsf{argToD} \circ \mathsf{argToS})(v) = v$ with $v : T_1$.

This makes argToS and argToD an isomorphism over the constructor $C$.

In EVAL_STM_SD_CONSTR, we use argToS to convert the $\lambda^{\cong}$ argument $v$. Intuitively, since we are going from $\lambda^{\cong}$ to $\lambda^{\to}$, no checks are necessary because the type system of $\lambda^{\cong}$ enforces all the properties that $\lambda^{\to}$ does and more.

Conversely, in EVAL_DTM_DS_CONSTR, we must verify that the argument converted from $\lambda^{\to}$ meets the specification demanded by

$$
\begin{array}{llcl}
\lambda^{\rightarrow}\text{ Values} & u & ::= & x \mid \lambda x{:}\,S.x \mid {<}u_1, u_2{>} \mid C\ u \\
\lambda^{\rightarrow}\text{ Contexts} & \mathcal{E}_s & ::= & \Box \mid \Box\, s \mid u\,\Box \mid <\Box, s> \mid <u, \Box> \\
 & & \mid & \Box.1 \mid \Box.2 \mid C\,\Box \mid \mathsf{letd}\ y = \Box\,\mathsf{in}\ s \\
 & & \mid & \mathsf{case}\,\Box\,\mathsf{of}\ \overline{C_i\ x_i \rightarrow s_i}^{\,i} \mid \mathsf{SD}_T^S\Box \\
\lambda^{\cong}\text{ Values} & v & ::= & y \mid \lambda y{:}\,T.t \mid {<}v_1, v_2{>} \mid C\ v \\
\lambda^{\cong}\text{ Contexts} & \mathcal{E}_t & ::= & \Box \mid \Box\, t \mid v\,\Box \mid <\Box, t> \mid <v, \Box> \\
 & & \mid & \Box.1 \mid \Box.2 \\
 & & \mid & C\,\Box \mid \mathsf{case}\,\Box\,\mathsf{of}\ \overline{C_i\ y_i \rightarrow t_i}^{\,i} \\
 & & \mid & \mathsf{DS}_S^T\Box \mid \Box \cong t_2 \rhd t \mid v \cong \Box \rhd t
\end{array}
$$

$\boxed{s \longrightarrow s'}$

$$\frac{\begin{array}{l} C{:}S \rightarrow A \in \Psi_0 \\ C{:}(y : T_1) \rightarrow B\ t_1 \in \Psi_0 \\ \mathsf{argToS}_C\, v = u \end{array}}{\mathsf{SD}^{A}_{(B\ t)}\, C\ v \longrightarrow C\ u}\ \text{EVAL\_STM\_SD\_CONSTR}$$

$$\frac{}{\mathsf{SD}^{(S_1 \rightarrow S_2)}_{((y{:}T_1) \rightarrow T_2)} \lambda y{:}\, T'_1.t \longrightarrow \lambda x{:}\, S_1.\mathsf{letd}\ y' = \mathsf{DS}^{T_1}_{S_1} x\ \mathsf{in}\ \mathsf{SD}^{S_2}_{([y'/y]\,T_2)}((\lambda y{:}\, T'_1.t)\ y')}\ \text{EVAL\_STM\_SD\_ABS}$$

$$\frac{}{\mathsf{SD}^{(S_1 * S_2)}_{((y{:}T_1) * T_2)}\ {<}v_1, v_2{>} \longrightarrow {<}\mathsf{SD}^{S_1}_{T_1} v_1, \mathsf{SD}^{S_2}_{([v_1/y]\,T_2)} v_2{>}}\ \text{EVAL\_STM\_SD\_PAIR}$$

$\boxed{t \longrightarrow t'}$

$$\frac{\begin{array}{l} C{:}S \rightarrow A \in \Psi_0 \\ C{:}(y : T_1) \rightarrow B\ t_1 \in \Psi_0 \\ \mathsf{argToD}_C\, u = v \end{array}}{\mathsf{DS}^{(B\ t)}_{A} (C\ u) \longrightarrow t \cong [v/y]t_1 \rhd (C\ v)}\ \text{EVAL\_DTM\_DS\_CONSTR}$$

$$\frac{}{\mathsf{DS}^{((y{:}T_1) \rightarrow T_2)}_{(S_1 \rightarrow S_2)} \lambda x{:}\, S'_1.s \longrightarrow \lambda y{:}\, T_1.\mathsf{DS}^{T_2}_{S_2}((\lambda x{:}\, S'_1.s)\, (\mathsf{SD}^{S_1}_{T_1}\, y))}\ \text{EVAL\_DTM\_DS\_ABS}$$

$$\frac{}{\mathsf{DS}^{((y{:}T_1) * T_2)}_{(S_1 * S_2)}\ {<}u_1, u_2{>} \longrightarrow \mathsf{let}\ y' = \mathsf{DS}^{T_1}_{S_1} u_1\ \mathsf{in}\ {<}y', \mathsf{DS}^{[y'/y]\,T_2}_{S_2} u_2{>}}\ \text{EVAL\_DTM\_DS\_PAIR}$$

$$\frac{}{v \cong v \rhd t \longrightarrow t}\ \text{EVAL\_DTM\_GUARD\_REFL} \qquad \frac{v \neq v'}{v \cong v' \rhd t \longrightarrow \mathsf{error}}\ \text{EVAL\_DTM\_GUARD\_ERROR}$$

**Figure 6.** SD Evaluation: Contexts and Rules

the $\lambda^{\cong}$ datatype. To generate this check, we note that the type of the new constructor $C\, v$ by WF_DTM_CTOR is $B\,[v/y]t_1$ where $B : T_1 \Rightarrow * \in \Psi_0$. The type demanded by the boundary is $B\, t$ and so we must check $t \cong [v/y]t_1$. Note that because of our restriction that $\mathsf{FO}\,(T_1)$, the equality check will never need to compare lambdas, only data of first-order type.

### 3.4 Equivalence

Equivalence checks are the core of a dependently-typed system. Figure 7 outlines the most important of these, equivalence over $\lambda^{\cong}$ terms. We elide $\lambda^{\cong}$ kind equivalence ($\Gamma \vdash K \equiv K'$) and $\lambda^{\cong}$ type equivalence ($\Gamma \vdash T \equiv T'$) as they are standard.

Our term-level equivalence is reflexive, transitive, and symmetric by the EQ_DTM_REFL, EQ_DTM_SYM, and EQ_DTM_TRANS rules. The most interesting of these rules is EQ_DTM_STEP which allows us to use reduction of $t$ in our equivalence relation. This rule is good because we do not need an explicit notion of $\lambda^{\rightarrow}$ equivalence, which would be unnatural. That is, in a real system, the $\lambda^{\cong}$ will only have available to it the ability to evaluate $\lambda^{\rightarrow}$ terms rather than have access to the internals of the entire $\lambda^{\rightarrow}$ program.

One subtlety that sets us apart from dependent languages like Coq and Agda is that our EQ_STM_STEP rule is restricted to *call-by-value* reduction. Pure, strongly normalizing languages have the luxury of allowing arbitrary $\beta$-reductions when comparing types because any order of evaluation gives the same answer. In our language that is not the case because of run-time errors, e.g. $(\lambda y\colon \mathsf{Unit}.\mathsf{unit})\ \mathsf{error}$ evaluates to error under CBV but to unit under CBN. This problem would get even worse if the language included more interesting side-effects.

For this reason, the type equivalence judgment is defined in terms of the evaluation relation $\longrightarrow$ which is explicitly CBV. Even so, we *do* want to allow reduction of open terms. For example to typecheck the usual *append* function we want $\mathsf{List}\,(0+y) \equiv \mathsf{List}\, y$. Therefore, our definition of values includes variables. To make that choice work, we are careful to only substitute values for variables. In particular, we need an extra premise in WF_DTM_APP to check that the type $[t_2/y]T_2$ is well-kinded. It might not be, since the well-kindedness of $(y\colon T_1) \rightarrow T_2$ may depend on $y$ being a value.

### 3.5 Examples

To get a better understanding of how our system works, let's expand on the List example we've used so far.

The complete set of definitions for our List datatype are

$$\begin{aligned}
&\mathsf{List} : \mathsf{Int} \Rightarrow *\\
&\mathsf{Nil} : \mathsf{Unit} \rightarrow \mathsf{List}\\
&\mathsf{Nil} : (y\colon\mathsf{Unit}) \rightarrow \mathsf{List}\,0\\
&\mathsf{Cons} : (\mathsf{List} * \mathsf{Int}) \rightarrow \mathsf{List}\\
&\mathsf{Cons} : (y_1\colon(y_2\colon\mathsf{Int}) * (\mathsf{List}\,y_2 * \mathsf{Int})) \rightarrow \mathsf{List}\,(y_1.1) + 1.
\end{aligned}$$

So the types of our argument conversion functions are

$$\begin{aligned}
&\mathsf{argToS}_{\mathsf{Nil}} : \mathsf{Unit} \rightarrow \mathsf{Unit}\\
&\mathsf{argToD}_{\mathsf{Nil}} : \mathsf{Unit} \rightarrow \mathsf{Unit}\\
&\mathsf{argToS}_{\mathsf{Cons}} : (y_1 : (y_2 : \mathsf{Int}) * (\mathsf{List}\ y_2 * \mathsf{Int})) \rightarrow (\mathsf{List} * \mathsf{Int})\\
&\mathsf{argToD}_{\mathsf{Cons}} : (\mathsf{List} * \mathsf{Int}) \rightarrow (y_1 : (y_2 : \mathsf{Int}) * (\mathsf{List}\ y_2 * \mathsf{Int})).
\end{aligned}$$

Note that the type of the arguments to $\mathsf{Cons}^{\rightarrow}$ is a pair whereas $\mathsf{Cons}^{\cong}$ is a triple. This is because the extra Int carried by $\mathsf{Cons}^{\cong}$ is required to represent the size of the argument List.

Morally, a $\mathsf{List}\, y$ has length $y$ so our conversions needs to respect that property. The conversions of the arguments to Nil are trivial.

$$\begin{aligned}
&\mathsf{argToS}_{\mathsf{Nil}}\mathsf{unit} = \mathsf{unit}\\
&\mathsf{argToD}_{\mathsf{Nil}}\mathsf{unit} = \mathsf{unit}
\end{aligned}$$

To convert from a $\mathsf{Cons}^{\cong}$ to a $\mathsf{Cons}^{\rightarrow}$, we can simply drop the index argument. To convert in the other direction, we must regenerate it by requesting the List's length.

$$\begin{aligned}
&\mathsf{argToS}_{\mathsf{Cons}}(k, l, v) = (l, v)\\
&\mathsf{argToD}_{\mathsf{Cons}}(l, v) = (\mathsf{length}(l), (l, v))
\end{aligned}$$

This is reminiscent of McBride's work on ornamental types [20] where he also makes the observation that the difference between a simply-typed list and a standard dependently-typed list is the "ornamental" length data.

Matthews and Amhed demonstrate how nested boundaries can enforce specifications over the behavior of the weakly-typed language while being written in a strongly-typed language [18]. In their system, they are only able to express simple type specifications, e.g., that a Scheme function performs at type $\mathsf{Int} \rightarrow \mathsf{Int}$. As expected with our dependently-typed language, we are able to express more powerful constraints via this method. For example consider a function pop over simply-typed Lists.

$$\mathsf{pop} : \mathsf{List} \rightarrow \mathsf{List}$$

Given this function, we can write a safe variant of pop in $\lambda^{\cong}$ that simply calls pop to do the heavy lifting:

$$\begin{aligned}
&\mathsf{safePop} : (n : \mathsf{Int}) \rightarrow \mathsf{List}\ n \rightarrow \mathsf{List}\ (n-1)\\
&\mathsf{safePop} = \lambda n : \mathsf{Int}.\lambda y : \mathsf{List}\ n.\mathsf{DS}_{\mathsf{List}}^{\mathsf{List}\ n-1}\mathsf{pop}(\mathsf{SD}_{\mathsf{List}\ n}^{\mathsf{List}}y))
\end{aligned}$$

Now, this function will verify via dynamic checks that — provided the length of the subject list $n$ — pop does the right thing for that list.

Providing this length argument explicitly is annoying, so we can write one more wrapper around this method that is callable directly from $\lambda^{\rightarrow}$ and has the signature we want. The difference between this and the original pop is that now the function will check to see if pop produces the correct value:

$$\begin{aligned}
\mathsf{verifiedPop}\quad &: \mathsf{List} \rightarrow \mathsf{List}\\
\mathsf{verifiedPop}\quad &= \lambda y : \mathsf{List}.\\
&\quad \mathsf{let}\ l = \mathsf{length}\ y\ \mathsf{in}\\
&\quad \mathsf{SD}^{\mathsf{List}}_{\mathsf{List}\ \mathsf{DS}^{\mathsf{Int}}_{\mathsf{Int}} l-1}(\\
&\qquad\qquad \mathsf{safePop}\ (\mathsf{DS}^{\mathsf{Int}}_{\mathsf{Int}} l)\ (\mathsf{DS}^{\mathsf{DS}^{\mathsf{Int}}_{\mathsf{Int}}\mathsf{List}\ l}_{\mathsf{List}}\ y))
\end{aligned}$$

verifiedPop is a good example of the power of dependent interoperability. We are able to take a simply-typed piece of code and then inject dynamic checks to verify its behavior against a dependently-typed specification.

## 4. Metatheory

Our technical contribution is a proof of type safety for SD: every well-typed term either goes to a value, diverges, or goes to error. We state this result in the usual way, via Preservation and Progress theorems.

The type-safety proof puts some requirements on the user-defined translation-functions argToD, argToS, and $\mathsf{corr}\,(A, B)$. These are stated in figure 8, and we will point out where they are needed. Note that the round-tripping law is not one of the properties needed for type-safety. The term equivalence judgment does not axiomatize this property, so violating it does not lead to type errors. However, we still feel that requiring it rules out bad behavior.

### 4.1 Structural Lemmas

We begin by showing basic structural properties of the type system: Weakening, Substitution, and ignoring redundant assumptions.

Since the different syntactic categories of our language (simple and dependent terms, types and kinds) form a mutually recursive system, the proofs of these lemmas also need to be by mutual induction. The typing judgments call out to the type equivalence judgments, but the equivalence is defined without any reference to

$\boxed{\Gamma \vdash t \cong t'}$

$$\frac{t \cong t' \in \Gamma}{\Gamma \vdash t \cong t'}\text{EQ\_DTM\_ASSUMPTION} \qquad \frac{t \longrightarrow t'}{\Gamma \vdash t \cong t'}\text{EQ\_DTM\_STEP}$$

$$\frac{}{\Gamma \vdash t \cong t}\text{EQ\_DTM\_REFL} \qquad \frac{\Gamma \vdash t' \cong t}{\Gamma \vdash t \cong t'}\text{EQ\_DTM\_SYM} \qquad \frac{\Gamma \vdash t \cong t' \quad \Gamma \vdash t' \cong t''}{\Gamma \vdash t \cong t''}\text{EQ\_DTM\_TRANS}$$

$$\frac{\Gamma \vdash t_1 \cong t_1' \quad y \notin \mathbf{dom}(\Gamma)}{\Gamma \vdash [t_1/y]t \cong [t_1'/y]t}\text{EQ\_DTM\_SUBST} \qquad \frac{\Gamma \vdash t \cong t' \quad y \notin \mathbf{dom}(\Gamma)}{\Gamma \vdash [v/y]t \cong [v/y]t'}\text{EQ\_DTM\_SUBST\_VAL} \qquad \frac{\Gamma \vdash t \cong t' \quad x \notin \mathbf{dom}(\Gamma)}{\Gamma \vdash [u/x]t \cong [u/x]t'}\text{EQ\_DTM\_SSUBST\_VAL}$$

**Figure 7.** $\lambda^{\cong}$ Term Equivalence

**Property 1** (Types of argToD/argToS)**.** *Suppose* $C{:}S \to A \in \Psi_0$ *and* $C{:}(y{:}T_1) \to B\ t_1 \in \Psi_0$.
*If* $\Gamma \vdash u : S$*, then* $\Gamma \vdash \mathsf{argToD}_C\, u : T_1$ *(if it is defined).*
*If* $\Gamma \vdash v : T_1$*, then* $\Gamma \vdash \mathsf{argToS}_C\, v : S$ *(if it is defined).*

**Property 2** (Correctness of $\mathsf{corr}\,(A, B)$)**.** *If* $\mathsf{corr}\,(A, B)$*, then* $A$ *and* $B$ *have the same constructors* $C_i$*.*

**Property 3** (argToD/argToS respect substitution)**.** *If* $\mathsf{argToD}_C\, u$ *and* $\mathsf{argToS}_C\, v$ *are defined, then*
$\mathsf{argToD}_C([u_1/x_1]u) = [u_1/x_1](\mathsf{argToD}_C\, u)$
$\mathsf{argToD}_C([v_1/y_1]u) = [v_1/y_1](\mathsf{argToD}_C\, u)$
$\mathsf{argToS}_C([u_1/x_1]v) = [u_1/x_1](\mathsf{argToS}_C\, v)$
$\mathsf{argToS}_C([v_1/y_1]v) = [v_1/y_1](\mathsf{argToS}_C\, v)$

**Property 4** (argToD/argToS respect $\longrightarrow_{\mathrm{p}}$)**.**
*If* $u \longrightarrow_{\mathrm{p}} u'$*, then* $\mathsf{argToD}_C\, u \longrightarrow_{\mathrm{p}} \mathsf{argToD}_C\, u'$*.*
*If* $v \longrightarrow_{\mathrm{p}} v'$*, then* $\mathsf{argToS}_C\, v \longrightarrow_{\mathrm{p}} \mathsf{argToS}_C\, v'$*.*

**Property 5.** argToD *and* argToS *are defined for closed values.*

**Figure 8.** Requirements on the conversion functions

types, so the proofs about the equivalence judgments can be done first. For example, Weakening can be proved in two lemmas, each of which is proved using mutual induction.

**Lemma 1** (Weakening for Equivalence)**.**

1. *If* $\Gamma_1, \Gamma_3 \vdash t \cong t'$*, then* $\Gamma_1, \Gamma_2, \Gamma_3 \vdash t \cong t'$*.*
2. *If* $\Gamma_1, \Gamma_3 \vdash T \equiv T'$*, then* $\Gamma_1, \Gamma_2, \Gamma_3 \vdash T \equiv T'$*.*
3. *If* $\Gamma_1, \Gamma_3 \vdash K \equiv K'$*, then* $\Gamma_1, \Gamma_2, \Gamma_3 \vdash K \equiv K'$

**Lemma 2** (Weakening)**.**

1. *If* $\Gamma_1, \Gamma_3 \vdash t : T$ *then* $\Gamma_1, \Gamma_2, \Gamma_3 \vdash t : T$*.*
2. *If* $\Gamma_1, \Gamma_3 \vdash s : S$ *then* $\Gamma_1, \Gamma_2, \Gamma_3 \vdash s : S$*.*
3. *If* $\Gamma_1, \Gamma_3 \vdash T : *$ *then* $\Gamma_1, \Gamma_3, \Gamma_3 \vdash T : *$*.*
4. *If* $\vdash \Gamma_1, \Gamma_2$ *then* $\vdash \Gamma_1$

The other lemmas are proved by similar mutual inductions. To save space we abbreviate sets of statements like this to $\Gamma \vdash J$, where the $J$ stands for all the judgment forms in the type system (equivalence, typing, and kinding).

For the Preservation proof we need a substitution lemma. Somewhat unusually, it is restricted to substituting values into the judgments, not arbitrary terms. This is because our term equivalence is CBV, so substituting a non-value could block reductions and cause types to no longer be equivalent.

**Lemma 3** (Substitution)**.**

1. *If* $\Gamma, x : S_2, \Gamma' \vdash J$ *and* $\Gamma \vdash u_2 : S_2$ *then* $\Gamma, [u_2/x]\Gamma' \vdash [u_2/x]J$*.*
2. *If* $\Gamma, y : T_2, \Gamma' \vdash J$ *and* $\Gamma \vdash v_2 : T_2$ *then* $\Gamma, [v_2/y]\Gamma' \vdash [v_2/y]J$*.*

Because we present dependent pattern matching using explicit equality assumptions in the context, we also need a set of structural lemmas stating that we can omit redundant equations and swap equivalent ones. These lemmas are used when proving type preservation of case-expressions and guard expressions: when the scrutinee steps, the corresponding equation changes to a syntactically different but $\beta$-equivalent one.

**Lemma 4** (Cut)**.** *If* $\Gamma \vdash t_1 \cong t_2$ *and* $\Gamma, t_1 \cong t_2, \Gamma' \vdash J$*, then* $\Gamma, \Gamma' \vdash J$*.*

**Lemma 5** (Context Equivalence)**.** *If* $\Gamma \vdash t_1 \cong t_1'$ *and* $\Gamma \vdash t_2 \cong t_2'$ *and* $\Gamma, t_1 \cong t_2, \Gamma' \vdash J$*, then* $\Gamma, t_1' \cong t_2', \Gamma' \vdash J$*.*

Cut is proved like a substitution lemma: each use of the equality assumption is replaced by the explicit derivation of the equation. The Context Equivalence lemma follows as a corollary of Weakening and Cut.

## 4.2 Preservation

We prove preservation by mutual recursion on the simple typing, dependent typing, and kinding judgment.

**Theorem 1** (Preservation)**.**

1. *If* $\Gamma \vdash s : S$ *and* $s \longrightarrow s'$ *then* $\Gamma \vdash s' : S$*.*
2. *If* $\Gamma \vdash [t/y]t_0 : T$ *and* $t \longrightarrow t'$ *then* $\Gamma \vdash [t'/y]t_0 : T$*.*
3. *If* $\Gamma \vdash [t/y]T_0 : K$ *and* $t \longrightarrow t'$ *then* $\Gamma \vdash [t'/y]T_0 : K$*.*

The statement for simple typing is standard but we have generalized the ones for dependent typing and kinding. The reason for this twist is again the CBV-style dependent typesystem: we need to know that the premise $\Gamma \vdash [t_2/y]T_2 : *$ to the WF_DTM_APP rule is preserved when $t_2$ steps. The generalization creates some extra congruence-like cases to deal with, but essentially this is still a standard Preservation proof.

The proof of this theorem informs the typing rules for the interoperability features. We highlight a few interesting cases.

First, the case when a SD-boundary for pairs steps is interesting because we substitute into the type on the SD boundary:

$$\mathsf{SD}^{S_1 * S_2}_{(y{:}T_1) * T_2}\ {<}v_1, v_2{>} \longrightarrow {<}\mathsf{SD}^{S_1}_{T_1}\, v_1, \mathsf{SD}^{S_2}_{[v_1/y]\,T_2}\, v_2{>}$$

This is different from prior work on non-dependent interoperability. We might worry that this would interfere with the compatibility check of the type. However, that is not the case, as we have the following lemma, which states that compatibility never looks at the terms embedded inside a type.

**Lemma 6.** *$S \Leftrightarrow T$ iff $S \Leftrightarrow [t/y]\,T$.*

Now, from the derivation of $\mathsf{SD}^{S_1 * S_2}_{(y{:}T_1) * T_2}\ < v_1, v_2 >$ we get $S_1 * S_2 \Leftrightarrow (y : T_1) * T_2$, so by inversion $S_2 \Leftrightarrow T_2$ and hence $S_2 \Leftrightarrow [v_1/y]\,T_2$, which is the compatibility condition that we need for the term $\mathsf{SD}^{S_2}_{[v_1/y]\,T_2}\, v_2$ to be well-typed.

Next, consider the case when a DS-boundary for a data constructor steps. This is the case that motivates our handling of dynamic checks:

$$\mathsf{DS}^{(B\ t)}_{A} (C\ u) \longrightarrow t \cong [v/y]t_1 \rhd (C\ v) \text{ where } \mathsf{argToD}_C u = v$$

when the signature contains declarations $C : S \to A$ and $C : (y : T_1) \to B\ t_1$. By our requirements on argToD we know that $\Gamma \vdash v : T_1$, so $\Gamma \vdash C\ v : B\ [v/y]t_1$. By the type conversion rules, that means $\Gamma, t \cong [v/y]t_1 \vdash C\ v : B\ t$. So we wrap the expression in a guard that enforces that equality assumption.

A final interesting case is when a guarded term steps. This motivates the structural lemmas Cut and Context Equivalence. The typing rule looks like

$$\frac{\begin{array}{l}\Gamma \vdash t_0 : T_0\\ \Gamma \vdash t_1 : T_0\\ \mathsf{FO}\,(T_0)\\ \Gamma, t_1 \cong t_0 \vdash t : T\end{array}}{\Gamma \vdash t_1 \cong t_0 \rhd t : T}\ \text{WF\_DTM\_GUARD}$$

Consider how the term can step. If $t_1 \longrightarrow t_1'$, then it suffices to show $\Gamma, t_1' \cong t_0 \vdash t : T$. But by the rule EQ_DTM_STEP, $\Gamma, t_1' \cong t_0$ and $\Gamma, t_1' \cong t_0$ are equivalent contexts. Otherwise, if it steps by $v \cong v \rhd t \longrightarrow t$, then by EQ_DTM_REFL the equation $v \cong v$ was redundant, so by Cut we can show $\Gamma \vdash t : T$ as required. Finally, it may step by $v \cong v' \rhd$ error. Since error is always well-typed, preservation holds. Although the proof doesn't illustrate it, the $\mathsf{FO}\,(T_0)$ restriction means that we will never go to error unless it is absolutely necessary, when $v$ and $v'$ are unequal first-order values.

### 4.3 Progress

As it turns out, the interoperability features do not add much complication to the Progress part of the proof. However, as is common in languages with dependent pattern matching, we need to do a bit of work to rule out contradictory equalities.

To prove progress we first need to prove a canonical forms lemma.

**Lemma 7** (Canonical Forms)**.**

1. *If $\cdot \vdash v : (y{:}\,T_1) \to T_2$ then $v$ is $\lambda y{:}\,T.t$.*
2. *If $\cdot \vdash v : (y{:}\,T_1) * T_2$ then $v$ is ${<}v_1, v_2{>}$.*
3. *If $\cdot \vdash v :$ Unit then $v$ is* unit.
4. *If $\cdot \vdash v : B\ t$ then $v$ is $C\ v'$ and $C{:}(y : T) \to B\ t' \in \Psi_0$.*

This does not follow immediately from inspecting the typing judgment, because of the rule EQ_DTY_INCON: if we could somehow in the empty context prove $\cdot \vdash C_1\, v_1 \cong C_2\, v_2$ where $C_1 \neq C_2$, then we could assign any term any type. So we need to rule out such an inconsistent equation. However, the way we define the term equivalence judgment $\Gamma \vdash t \cong t'$ makes that difficult. The definition is succinct, but because it has an explicit transitivity rule it doesn't give any leverage for doing induction on it.

Our solution is to define an auxiliary notion of *parallel reduction*, denoted $\longrightarrow_{\mathrm{p}}$, in the style of Takahashi [31]. This relation contains the evaluation relation $\longrightarrow$, but it also allows reducing more than one redex, and reducing inside the body of a lambda expression or other binder. For example, the two parallel reduction rules for applications are:

$$\frac{\begin{array}{c}t_1 \longrightarrow_{\mathrm{p}} t_1'\\ t_2 \longrightarrow_{\mathrm{p}} t_2'\end{array}}{t_1\, t_2 \longrightarrow_{\mathrm{p}} t_1'\, t_2'} \qquad \frac{\begin{array}{c}t_1 \longrightarrow_{\mathrm{p}} t_1'\\ v_2 \longrightarrow_{\mathrm{p}} v_2'\end{array}}{(\lambda y{:}\,T.t_1)\, v_2 \longrightarrow_{\mathrm{p}} [v_2'/y]t_1'}$$

As a result, unlike evaluation, parallel reduction is closed under substitution: if $v_1 \longrightarrow_{\mathrm{p}} v_2$ and $t_1 \longrightarrow_{\mathrm{p}} t_2$ then $[v_1/y]t_1 \longrightarrow_{\mathrm{p}} [v_2/y]t_2$ and $[t_1/y]t \longrightarrow_{\mathrm{p}} [t_2/y]t$. We also show that it is confluent. Together, these properties lets us prove a useful characterization of term equivalence.

**Lemma 8** (Parallel reduction contains term equivalence)**.** *If $\cdot \vdash t_1 \cong t_2$, then there exists some $t'$ such that $t_1 \longrightarrow_{\mathrm{p}^*} t'$ and $t_2 \longrightarrow_{\mathrm{p}^*} t'$.*

This lemma rules out the inconsistent equation we were worried about, since reducing a term can never change its constructor. We can then straightforwardly show Canonical Forms and Progress.

**Theorem 2** (Progress)**.**

1. *If $\cdot \vdash t : T$ then either $t \longrightarrow t'$, $t$ is a value, or $t$ is* error.
2. *If $\cdot \vdash s : S$ then either $s \longrightarrow s'$, $s$ is a value, or $s$ is* error.

However, there is a difficulty. In order to prove substitution and confluence of parallel reduction, we need to assume these properties for the argToD and argToS functions, because the reduction relation is defined in terms of them. This yields properties 3 and 4 in figure 8.

We expect these requirements to be satisfied by any "natural" definition of argToD and argToS. For example, one definition that would not respect parallel reduction would be to define

$$\begin{array}{ll}\mathsf{argToS}_C(\lambda y{:}\,\mathsf{Unit}.1 + 1) & = \mathsf{true}\\ \mathsf{argToS}_C(\lambda y{:}\,\mathsf{Unit}.2) & = \mathsf{false}\end{array}$$

But such a function, which examines the body of a $\lambda$-abstraction, could never be written by user code. In practice, we expect the translation functions to do pattern matching and to construct constructor applications and function calls, e.g. $\mathsf{argToD}_{\mathsf{Cons}}$ in section 3.5. Such translation functions automatically satisfy these requirements, because they are just built up from $\lambda^{\rightarrow}$ and $\lambda^{\cong}$ terms.

## 5. Additional Properties

Two important properties of SD that deserve special mention are the soundness of the dependently-typed fragment of the language and decidable typechecking.

### 5.1 Soundness

Soundness of a dependently-typed language is important because a sound language can function as a proof system. Unfortunately, by introducing boundaries that produce errors and defer complete typechecking until runtime, we've removed soundness from $\lambda^{\cong}$.

In the case of error we can simply consider the empty datatype false that should have no inhabitants. But due to SD_WF_DTM_-ERROR we can ascribe error that type.

With respect to complete typechecking, consider the term

$$\mathsf{case}\ \mathsf{DS}^{(\mathsf{Foo}\ 1)}_{\mathsf{Foo}} \mathsf{mkFoo}\ \mathsf{unit}\ \mathsf{of}\ \mathsf{mkFoo}\ y \to t$$

Where $\mathsf{Foo} : \mathsf{Int} \Rightarrow *$ and $\mathsf{mkFoo} : (y : \mathsf{Unit}) \to \mathsf{Foo}\, 0$. The boundary typechecks giving $\mathsf{DS}^{(\mathsf{Foo}\ 1)}_{\mathsf{Foo}} s$ the type $\mathsf{Foo}\, 1$, an

$\boxed{s \longrightarrow s'}$ $\boxed{t \longrightarrow t'}$

$$\frac{}{\mathsf{SD}^{S}_{\mathbf{L}}(\mathsf{DS}^{\mathbf{L}}_{S} u) \longrightarrow u}\text{EVAL\_STM\_SD\_LUMP}$$

$$\frac{}{\mathsf{DS}^{T}_{\mathbf{L}}(\mathsf{SD}^{\mathbf{L}}_{T} v) \longrightarrow v}\text{EVAL\_DTM\_DS\_LUMP}$$

**Figure 9.** Evaluation Rules for Lumps

uninhabited type. By SD_WF_DTM_CASE, in the only case for Foo we arrive at the inequality $0 \cong 1 \in \Gamma$ and can thus typecheck the case to false.

Note that this is an unavoidable consequence of boundaries. We need to signal errors at runtime and our boundaries necessarily make claims (e.g., above that the boundary expects a $\mathsf{Foo}\,1$ even though it is impossible) that can only be verified at runtime.

However, like Lambda-eek [12], we believe that while an interoperating calculus such as SD is not necessarily suitable as a proof system, it is interesting as a programming language in its own right.

### 5.2 Decidable Typechecking

A related question to the soundness of $\lambda^{\cong}$ is whether the typechecking of SD is decidable in the presence of term evaluation in types. With our current formulation of $\lambda^{\rightarrow}$ and $\lambda^{\cong}$, we believe (but do not prove) that SD is strongly normalizing and thus typechecking of SD is decidable. We believe that this is reasonable given that both $\lambda^{\rightarrow}$ and $\lambda^{\cong}$ appear to be strongly normalizing and the type-directed boundaries that we consider in SD themselves do not contribute any additional computational power to the language.

Irrespective of this, it is clear that we can make SD typechecking undecidable by giving $\lambda^{\rightarrow}$ recursive functions. This is because we determine the equivalence of $t_1 \cong t_2$ by $\beta$-reduction as per the EQ_DTM_STEP rule (Figure 7). With recursive functions in $\lambda^{\rightarrow}$, evaluation of a DS boundary could end up in an infinite loop.

Because our actual $\lambda^{\rightarrow}$ language will likely be a general-purpose functional language with recursion, how might we recover decidable typechecking in this scenario? One such approach is to introduce a purity check in $\lambda^{\cong}$ that restricts boundaries from being embedded in types. This is a clean way to regain decidable typechecking but at the cost of losing the ability to embed terminating boundary terms in types.

Finally, we may give up the ambition that the typechecker automatically decides term equivalence by evaluating terms, and instead require the programmer to add explicit annotations stating what should be evaluated for how many steps. An example of a language taking this approach is Guru [29].

### 5.3 Lumping and Non-termination

One tempting suggestion to alleviate the problem of decidable typechecking is to limit how we can compute with values across the boundary. Rather than marshaling values, perhaps we can treat data on the other side of the boundary as a opaque *lump* that we can carry around and give back, but otherwise not inspect its contents. We give the evaluation rules in Figure 9. While appealing at first glance, it turns out that this system admits non-termination.

In the lump variant of our rules, we introduce a type $\mathbf{L}$ that represents an opaque lump value contained in a boundary. With lumps, boundaries no longer marshal values between languages or otherwise look at their structure. Instead, boundaries are "canceled out" when they meet each other as per EVAL_STM_SD_LUMP and EVAL_DTM_DS_LUMP. The problem is that it turns out that you can write an infinite loop with these boundaries in a similar manner to type dynamic [1] where you use a pair of functions of type $\mathbf{L} \rightarrow (\mathbf{L} \rightarrow \mathbf{L})$ and $(\mathbf{L} \rightarrow \mathbf{L}) \rightarrow \mathbf{L}$ to encode a term $\Omega$ that loops. The actual terms for these functions and $\Omega$ are the same as Matthews' and Findler's versions for their ML-in-ML calculus [19] but adapted to our boundaries.

Because of this, any interoperability boundary between simply- and dependently-typed languages using a lump style induces undecidable typechecking if boundaries can appear in dependent types and reduce.

## 6. Comparisons

Many real-world dependently-typed languages provide some facilities for interoperability with simply-typed languages. However we know of no language that provides the flexibility suggested by SD. Now that we've established SD and its properties, it is instructive to compare the techniques used by these dependently-typed languages with how SD establishes its interoperability boundaries for two reasons. First, if SD can accurately describe the interoperability features of these languages, then it builds confidence that SD is a good model for dependent interoperability in general. And second, the differences between the two suggests ways that the dependently-typed language can improve its interoperability support, or conversely, why it may be hard to do so.

### 6.1 ATS Data Translation

ATS [5] is built with interoperability with C in mind. Since the two languages share the same data representation, marshaling is relatively trivial. ATS values are typically exposed to C as wrapped structs, e.g., a C int has type `ats_int_type` in ATS. ATS functions can be exposed to C via `extern` declarations and C code can either be inlined into ATS files or referenced as external values or types. In this sense, ATS closely mimics the two-way interoperability boundary of SD.

However, beyond basic type-checking, ATS interoperability makes no attempt at checking to see if dependent type properties are preserved when traveling in and out of C. This is because with arbitrary casts, C code can arbitrarily munge ATS values or otherwise break the type guarantees made by ATS.

### 6.2 Extraction in Coq

The theorem prover Coq [32] provides a mechanism, `Extraction`, that extracts functional programs written in OCaml (or other functional languages such as Haskell) from proofs of specifications [14]. Coq distinguishes between computationally relevant types (Sets) and computationally irrelevant types (Props) and uses that information to guide `Extraction`. Datatypes extracted from Coq are translated into comparable datatypes in ML. Alternatively, Coq provides a mechanism for the user to map a Coq datatype and its associated constructors into a ML datatype and its constructors.

For our purposes, `Extraction` is a form of *one-way* interoperability where ML code can use verified Coq code . If we imagine the extracted program as living in $\lambda^{\cong}$ and the ML code living in $\lambda^{\rightarrow}$, then this amounts to only allowing the user to call $\lambda^{\cong}$ code via a SD boundary.

However, there are several limitations to the one-way interoperability model offered by `Extraction`:

1. **Extracted code does not enforce the properties of datatypes.** By design the extracted code is correct up to the verification done in Coq. However, because of erasure, the extracted code cannot verify that ML data passed to it meets the pre-conditions (if any) to use that code. For example, our $\mathsf{List}\,y$ example datatype would be erased to a simple List in ML. If the extracted code depends on receiving a non-empty List then it must trust

the user to give it a non-empty List rather than enforcing that pre-condition itself.

2. **User-defined translation of datatypes is simple macro replacement.** In SD, the user-defined translation function argToS is any function from the arguments of the $\lambda^{\cong}$ constructor to the $\lambda^{\rightarrow}$ constructor that respects the properties we outlined in the previous sections. In Coq, the analogous `Extract Inductive` command performs a macro-replacement of the occurrences of the datatype and its constructors with the strings specified with the commands. The resulting ML code is not even checked for well-formedness.

### 6.3 Agda Data Translation

Agda [22] provides a foreign-function interface that allows Agda to call into Haskell code. As part of the FFI, the user specifies Haskell functions to call from Agda with the `{-# COMPILED #-}` pragma. The user can also specify translations from Agda datatypes to Haskell datatypes via the `{-# COMPILED_DATA ... #-}` pragma.

Like Coq `Extraction`, the Agda FFI is a *one-way* interoperability layer. The difference is that the FFI allows Agda, the dependently-typed language, to invoke Haskell code, the simply-typed language . Translation occurs when Agda invokes a Haskell function. The arguments are converted to Haskell and the return value converted back to Agda according to the FFI's built in rules to translate Agda types coupled with the declared `COMPILED_DATA` pragmas.

Agda's FFI suffers from problems similar to Coq `Extraction` due to the restrictive nature of Agda's translation function. Agda erases terms in types down to unit so the translation has no way of preserving or even checking to see if the properties of dependent types are preserved. Unlike Coq `Extraction`'s macro-based datatype compatibility declarations, Agda's compatibility declarations are type-directed. However, they are still less flexible than SD as you can only map constructors of the same number of arguments and types.

### 6.4 Coq's Program Tactic

Coq's `Program` tactic [28] offers a different flavor of interoperability than `Extraction`. `Program` allows the user to write dependently-typed code in the form of predicate subtyping [27] over terms, but using a simply-typed language instead. This simply-typed language is a relaxed version of Coq's term language, but could very well be OCaml or Haskell instead.

The work flow of `Program` occurs in two steps:

1. The user writes a program in the simply-typed fragment. This includes predicates over types written in the refinement style `{x | P}`. The user does not need to write any proofs during this step.
2. Coq elaborates the program into Coq terms and then generates a series of proof obligations that the user must discharge. The result is a complete Coq term that is the program that meets the specifications outlined via the predicates of the program.

`Program` is an example of a dependently-typed system utilizing the power of a simply typed system to do interesting work. We can view the elaboration step from the simply-typed fragment to Coq as a translation from $\lambda^{\rightarrow}$ to $\lambda^{\cong}$ where we are interested in using $\lambda^{\cong}$ to prove properties of the $\lambda^{\rightarrow}$ program.

## 7. Prior Work

We believe our work is the first to directly address the technical challenges involved with interoperating between a dependently-typed and simply-typed programming language. However, there has been considerable effort in related areas that we highlight here.

***Interoperability Implementation*** Since different programming languages typically operate under different runtime environments, much of the early work in interoperability research focuses how to reconcile those environments. Frequently the analysis takes specific pairs of languages, usually C, with other languages such as Java [6], ML [3], and Haskell [4], but sometimes also with other language pairs such as Python to Scheme [25] or SML to Java [21]. Other approaches attempt to develop a *lingua franca* by which two languages can communicate such as C [2], the Java virtual machine, COM [26], or the .NET framework [30].

***Interoperability Semantics*** There has been comparatively less work in understanding the semantics of interoperating languages. We extend Matthews's and Findler's original work [19] that showed that even with simple language pairs — untyped and simply-typed lambda calculi — interoperability leads to some surprising results. Their latest work in this area focuses on adding polymorphism to a interoperability setting while preserving parametricity [18].

***Mixing Dependency with Dynamic*** A different thread of related research comes from analyses of dependently-typed languages intermixed with type dynamic [1]. Ou et al [24] introduce `simple` and `dependent` constructs in which dynamically-typed and dependently-typed, respectively, exists. They allow for nesting of such constructs (e.g., `simple{dynamic{...}}`) and provide rules for how simple blocks dynamically enforce constraints imposed by dependent blocks. Gronski et al [11] extend this approach to a pure-type system without explicit, separate constructs for dynamic and dependent types. Instead, they include dynamic as a base type and assume the rest of the world is dependent.

***Refinement Types and Contracts*** The underlying framework for many of these systems is the theory of refinement types [9] and higher-order contracts [7]. Recently, the study of contracts has gone in many directions, for example assigning blame [33]. Directly relevant to our work is the study of dependent contracts, e.g., the systems studied by Greenberg et al [10].

## 8. Future Work and Conclusion

We tackle the problem of making dependently-typed programming more accessible from the viewpoint of interoperability. Can we author an interoperability boundary between a dependently-typed language and a simply-typed language that preserves the properties enforced by the dependently-typed language? Our solution, the language SD, is able to meet design goals we set forth for such an interoperability layer: using code from one language from within the other language and verifying properties of simply-typed code with the dependently-typed language.

In the future, we would like to apply the ideas in this paper to improve the interop support of real-world languages like Coq and Agda, e.g., adding true "two-way" interoperability. Theoretically, there is also room for more careful analysis: proofs of strong normalization and a theorem characterizing when boundaries can be inserted without changing program behavior in harmful ways.

There are also more design variations for SD worth exploring. In particular, we restrict datatype indices at boundaries to be first-order. While this is not a serious limitation, it would be interesting to adapt ideas from the contracts literature and decompose equality checks of functions into checks at their use sites during type conversion. Finally, we can move beyond the pairing of dependent and simple types are explore other combinations such as dependent and dynamic types and pairings involving linear types.

## Acknowledgments

We thank the anonymous reviewers of PLPV for their invaluable feedback in refining this paper. This work was also partially supported by the National Science Foundation (NSF grant 0910500).

## References

[1] M. Abadi, L. Cardelli, B. C. Pierce, and G. Plotkin. Dynamic typing in a statically typed language. *ACM Transactions on Programming Languages and Systems*, 13(2):237–268, 1991.

[2] D. M. Beazley. Swig: an easy to use tool for integrating scripting languages with c and c++. In *Proceedings of the 4th conference on USENIX Tcl/Tk Workshop*, 1996.

[3] M. Blume. No-longer-foreign: Teaching an ml compiler to speak c "natively". In *Workshop on Multi-Language Infrastructure and Interoperability (BABEL)*, 2001.

[4] M. M. T. Chakravarty. C→haskell, or yet another interfacing tool. In *International Workshop on Implementation of Functional Languages (IFL)*, 1999.

[5] C. Chen and H. Xi. Combining programming with theorem proving. In *Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*, ICFP '05, pages 66–77, New York, NY, USA, 2005. ACM. ISBN 1-59593-064-7.

[6] G. T. et al. Safe java native interface. In *IEEE International Symposium on Secure Software Engineering*, 2006.

[7] R. B. Findler and M. Felleisen. Contracts for higher-order functions. *SIGPLAN Not.*, 37(9):48–59, 2002. ISSN 0362-1340.

[8] C. Flanagan. Hybrid type checking. *SIGPLAN Not.*, 41:245–256, January 2006. ISSN 0362-1340.

[9] T. Freeman and F. Pfenning. Refinement types for ml. In *Conference on Programming Language Design and Implementation*, pages 268–277, 1991.

[10] M. Greenberg, B. C. Pierce, and S. Weirich. Contracts made manifest. In *Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '10, pages 353–364, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-479-9. doi: http://doi.acm.org/10.1145/1706299.1706341. URL `http://doi.acm.org/10.1145/1706299.1706341`.

[11] J. Gronski, K. Knowles, A. Tomb, S. N. Freund, and C. Flanagan. Sage: Hybrid checking for flexible specifications. In *Scheme and Functional Programming Workshop*, pages 93–104, 2006.

[12] L. Jia, J. Zhao, V. Sjöberg, and S. Weirich. Dependent types and program equivalence. In *POPL '10: Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 275–286, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-479-9.

[13] X. Leory, D. Doligez, A. Frisch, J. Garrigue, D. Rémy, and J. Vouillon. *The Objective Caml system release 3.12*. 2010.

[14] P. Letouzey. Extraction in coq: An overview. In *Proceedings of the 4th conference on Computability in Europe: Logic and Theory of Algorithms*, pages 359–369, 2008.

[15] S. Liang. *The Java Native Interface: Programmer's Guide and Specification*. Addison Wesley, Reading, MA, 1999.

[16] T. Lindholm and Y. Frank. *The Java virtual machine specification second edition*. Prentice Hall, 1999.

[17] S. Marlow. *Haskell 2010 Language Report*.

[18] J. Matthews and A. Ahmed. Parametric polymorphism through run-time sealing or, theorems for low, low prices! In *Proceedings of the Theory and practice of software, 17th European conference on Programming languages and systems*, ESOP'08/ETAPS'08, pages 16–31, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 3-540-78738-0, 978-3-540-78738-9. URL `http://dl.acm.org/citation.cfm?id=1792878.1792881`.

[19] J. Matthews and R. B. Findler. Operational semantics for multi-language programs. *ACM Trans. Program. Lang. Syst.*, 31(3):1–44, 2009. ISSN 0164-0925.

[20] C. McBride. Ornamental algebras, algebraic ornaments. 2010.

[21] A. K. N. Benton. Interlanguage working without tears: Blending sml with java. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 126–137, 1999.

[22] U. Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.

[23] P.-M. Osera, V. Sjöberg, and S. Zdancewic. Dependent interoperability (extended version). Technical report, University of Pennsylvania, 2011.

[24] X. Ou, G. Tan, Y. Mandelbaum, and D. Walker. Dynamic typing with dependent types. In *IFIP TCS*, pages 437–450, 2004.

[25] D. S. P. Meunier. From python to plt scheme. In *Proceedings of the Fourth Workshop on Scheme and Functional Programming*, pages 24–29, 2003.

[26] R. Pucella. Towards a formalization for com, part i: The primitive calculus. In *Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, 2002.

[27] J. Rushby, S. Owre, and N. Shankar. Subtypes for specifications: Predicate subtyping in PVS. *IEEE Transactions on Software Engineering*, 24(9):709–720, sep 1998.

[28] M. Sozeau. Subset coercions in coq. In *Proceedings of the 2006 international conference on Types for proofs and programs*, TYPES'06, pages 237–252, Berlin, Heidelberg, 2007. Springer-Verlag.

[29] A. Stump, M. Deters, A. Petcher, T. Schiller, and T. Simpson. Verified programming in guru. In T. Altenkirch and T. Millstein, editors, *Programming Languges meets Program Verification (PLPV)*, pages 49–58, 2009.

[30] D. Syme. Ilx: Extending the .net common il for functional language interoperability. In *Workshop on Multi-Language Infrastructure and Interoperability (BABEL)*, 2001.

[31] M. Takahashi. Parallel reductions in $\lambda$-calculus. *Inf. Comput.*, 118(1):120–127, 1995. ISSN 0890-5401. doi: http://dx.doi.org/10.1006/inco.1995.1057.

[32] T. C. D. Team. The coq proof assistant: Reference manaul, 2010. URL `http://coq.inria.fr/refman/`.

[33] P. Wadler and R. B. Findler. Well-typed programs can't be blamed. In *ESOP '09: Proceedings of the 18th European Symposium on Programming Languages and Systems*, pages 1–16, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 978-3-642-00589-3.

[34] T. Wrigstad, F. Z. Nardelli, S. Lebresne, J. Östlund, and J. Vitek. Integrating typed and untyped code in a scripting language. In *POPL '10: Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 377–388, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-479-9.