A Calculus with Polymorphic and Polyvariant Flow Types

J. B. WELLS∗
Heriot-Watt University

ALLYN DIMOCK†
Harvard University

ROBERT MULLER‡
Boston College

FRANKLYN TURBAK§
Wellesley College

Abstract

We present λ^{CIL}, a typed λ-calculus which serves as the foundation for a typed intermediate language for optimizing compilers for higher-order polymorphic programming languages. The key innovation of λ^{CIL} is a novel formulation of intersection and union types and flow labels on both terms and types. These *flow types* can encode polyvariant control and data flow information within a polymorphically typed program representation. Flow types can guide a compiler in generating customized data representations in a strongly typed setting. Since λ^{CIL} enjoys confluence, standardization, and subject reduction properties, it is a valuable tool for reasoning about programs and program transformations.

1 Introduction

Explicitly typed intermediate languages (TILs) facilitate the safe and efficient compilation of programming languages (Morrisett, 1995; Tarditi *et al.*, 1996; Peyton Jones, 1996; Peyton Jones & Meijer, 1997; Shao, 1997; Benton *et al.*, 1998; Tolmach & Oliva, 1998; Morrisett *et al.*, 1999; Fitzgerald *et al.*, 1999; Cejtin *et al.*, 2000). The type information in the intermediate representation can be used to guide program analyses and transformations and to support run-time operations such as

∗ (*e-mail:* `jbw@cee.hw.ac.uk`), Edinburgh EH14 4AS, Scotland. This author's work was done while at Boston University, the University of Glasgow, and Heriot-Watt University. It was partially supported by NSF grants CCR–9113196, CCR–9417382, EIA–9806745 and EPSRC grant GR/L 36963.
† (*e-mail:* `dimock@deas.harvard.edu`), Cambridge, MA 02138, USA. Supported by NSF grants CCR–9417382, EIA–9806745/9806746/9806747.
‡ (*e-mail:* `muller@cs.bc.edu`), Chestnut Hill, MA 02467, USA. Supported by NSF grant EIA–9806746.
§ (*e-mail:* `fturbak@wellesley.edu`), Wellesley MA 02481, USA. Supported by NSF grant EIA–9806747.

garbage collection. TILs are also an important debugging aid in the compiler development process. Finally, there is growing recognition of the importance of TILs in certifying compilers (Necula & Lee, 1998; Appel & Felty, 2000).

TILs are especially important for the safe and efficient compilation of modern programming languages that support higher-order functions, polymorphic types, abstract data types or objects. These features make it difficult to compute efficient data and control representations. In the absence of type information, compilers for these language have usually adhered to a strong *uniform representation assumption* (URA), which dictates (1) that each type constructor in the language is associated with a single representation in the target machine language and (2) that all values are accessed through a fixed-size interface (usually achieved by "boxing" (Leroy, 1992)). While the URA simplifies compilation and facilitates the support of polymorphism and separate compilation, it stands in the way of customized data representation and classical optimizations. TIL-based compilers can use type information to choose different representations for different instances of the same type constructor. For example, a function mapping integers to integers can be given a different calling convention than a function mapping reals to reals — something not possible under the URA.

Typed intermediate languages based on System F (Girard, 1972; Reynolds, 1974) represent type polymorphism with universally and existentially quantified type variables. The explicit type information in these languages can be used to relax the URA for monomorphically typed data, however, these languages still impose the URA in a weaker form. First, the binding occurrences of the type variables effectively hide the types at which the polymorphic value is used. This makes it difficult to customize data representations. Second, even in the best case, data representations are computed on a per-type basis. But it is often desirable to choose different representations for values of the same type based on their usage. Multiple representations are often suggested by *polyvariant* analyses, i.e, those in which a given value or a given usage type is analyzed in multiple contexts. System F is not expressive enough to encode many polyvariant analyses.

In this paper we introduce λ^{CIL}, an explicitly typed λ-calculus that serves as a basis for a TIL in compilers for higher-order polymorphic programming languages.[1] The key innovation of the language is the integration of polyvariant flow information and polymorphic type information into a single *flow type* system that supports transformations into customized data representations. The type system represents flow information using a novel formulation of intersection and union types as well as flow label annotations on both terms and types. The types and annotations approximate the flow of values from the points where they are defined (their *sources*) to the points where they are used (their *sinks*).

The flow annotations can be used to construct data representations tailored

[1] "CIL" is an acronym for "Church Intermediate Language". The authors are members of the Church Project (`http://www.cs.bu.edu/groups/church/`) which is investigating the application of type systems in the safe and efficient compilation of higher-order typed programming languages.

specifically to source/sink pairs and to implement efficient control structures. For example, a compiler may choose to represent a function flowing to several application sites with several customized pieces of run-time code. If the function is given an intersection type, the run-time code may be specialized for the various concrete types in the intersection type. Intersection and union types together with flow labels support a more expressive notion of type polymorphism which can support customizing the run-time code for many context dependent properties such as callee-save register conventions, activation record layout, stack availability of free variable values, etc.

In λ^{CIL}, customization opportunities are represented by *virtual* forms. *Virtual tuples* introduce values of intersection type. Their components, whose code is identical except for type annotations, are duplicated (or split) representations of a value that is used in different contexts. *Virtual case expressions* eliminate values of union type. Their clauses, whose code is identical except for type annotations, can be thought of as duplicated (or split) contexts. These forms are virtual in the sense that they are compile-time entities that are not represented in the run-time code.

To customize code, the compiler may reify the virtual forms into corresponding real forms that are represented at run-time. When the compiler elects to customize a source, it reifies the virtual tuple by converting it to a real tuple and by converting the corresponding virtual projections to real ones. Additionally, reification of virtual tuples typically introduces customized code for the components that is no longer the same modulo type annotations. Similarly, the compiler may elect to represent a given application site with several customized run-time calling sequences, possibly inlining some number of the calls. In this case the compiler reifies a virtual case expression by converting it to a real one and by converting the corresponding virtual injections to real ones (i.e., ones with run-time tags). Once all representation decisions have been made, many virtual entities that have not been reified can be merged into a single real representation. Any virtual tuple (resp. virtual case expression) that persists to the back end is compiled into the code for *one* of its components (resp. clauses), since all of these are guaranteed to compile to the exactly the same code. A concrete example of the sort of customization supported by λ^{CIL} is presented in section 2.2. See (Dimock *et al.*, 2001a) for an extensive discussion of how λ^{CIL} supports customized function representations.

Flow types naturally integrate types with the sorts of control and data flow analyses that are computed in most modern compilers. The main benefit of this integration is that it helps to maintain the consistency of analyses across program transformations. Although it is in general difficult to maintain this consistency, it is easier than when the information is recorded separately, and, most importantly, the type checker can mechanically certify the consistency. In a well-typed λ^{CIL} program, any flows encoded in the types are conservative approximations of run-time flows. In contrast, in systems that maintain type and flow information separately, it can be difficult to update the flow information to be consistent with the results of a program transformation. In such systems, it may be necessary to reanalyze the program after every transformation. Not only is reanalysis potentially expensive, but analysis of a transformed program may yield less precise information than for the original program.[2]

The benefits of flow types are not achieved without cost. First, the encoding of flow information in the types and the duplication implied by virtual tuples and case expressions can lead to extremely large terms and types. Much of our effort in implementing a compiler based on λ^{CIL} has been focused on reducing the space required by our intermediate language. We have adapted the space-saving techniques described in (Shao *et al.*, 1998) as well as introduced some new techniques. These engineering issues are outside the scope of this paper and are reported elsewhere (Dimock *et al.*, 2001b).

Second, in their simplest form, flow types appear to require analysis of a whole program, and so seem to be at odds with separate compilation. Indeed, our prototype ML compiler based on λ^{CIL} assumes whole-program compilation. In this respect, it follows many other modern compilers for higher-order polymorphic languages that have used whole-program compilation to achieve high degrees of efficiency (Chambers *et al.*, 1996; Tolmach & Oliva, 1998; Benton *et al.*, 1998; Siskind, 1999; Cejtin *et al.*, 2000). But it is worth noting that flow types may not be inherently incompatible with separate compilation. Type systems closely related to that of λ^{CIL} (such as (Banerjee, 1997; Kfoury & Wells, 1999)) have a *principal typing* property – a key property for modular systems (Jim, 1996). (In contrast, neither System F nor the Hindley-Milner system have principal typings. The weak "principal types" property of the Hindley-Milner system does not support modular analysis.) This suggests the tantalizing possibility that λ^{CIL} could be the basis for a modular compilation system in which some compilation is performed at link-time (e.g., see (Fernandez, 1995)). Even if flow types do prove useful in a modular setting, they may be too "low-level" for user-level specification of module interfaces; universal and existential types seem more appropriate for such specification. Since intersection and union types appear to have different strengths and weaknesses than universal and existential types, it may be fruitful to integrate them into a single type system.

Flow and type information have been combined in other type systems, but most of these can only express monovariant flow analyses and none has been the basis for a calculus at the core of a typed intermediate language. The first correspondence between a type system and a monovariant flow analysis was shown in (Palsberg & O'Keefe, 1995). Heintze coined the term "control flow types" in the context of a system in which function types are annotated with sets of source labels (Heintze, 1995); the system is limited to the expression of monovariant flow analyses. The control flow effect system in (Tang & Jouvelot, 1994) also involves source label annotations of arrow types but cannot express polyvariant flow analyses. The first formal correspondence between a class of polyvariant flow analyses and intersec-

[2] This is particularly true in the case of modular analyses such as that of Banerjee (Banerjee, 1997), where the *rank* restriction can force flow information about the various inputs of a function to be merged when the function is passed to another function. Note that for some combinations of transformation and analysis, the reanalysis of the transformed program can yield *more* precise information. Indeed, that is one of the goals of some transformations.

tion and union types was shown in (Palsberg & Pavlopoulou, 2001). An improved correspondence presented in (Amtoft & Turbak, 2000) involves a type system with tagged intersection and union types that is closely related to that of λ^{CIL}. Flow and type information are also merged in constrained types (Curtis, 1990; Eifrig *et al.*, 1995; Aiken & Wimmers, 1993; Aiken *et al.*, 1994). Palsberg and Smith (Palsberg & Smith, 1996) show that the system of constrained types in (Eifrig *et al.*, 1995) accepts the same programs as the type system of Amadio and Cardelli (Amadio & Cardelli, 1993). Two recent compilers (Tolmach & Oliva, 1998; Cejtin *et al.*, 2000) have used flow information to reduce the size of closure datatypes introduced by defunctionalization. The resulting sum-of-product closure representations can be viewed as a simple flow type system in which all virtual tuples and variants have been reified.

Unlike many other typed intermediate languages, λ^{CIL} is a calculus (an equational theory of program fragments) rather than just a language with a deterministic operational semantics. In this paper, we prove that λ^{CIL} is confluent and has the subject reduction property. The former property ensures that the equational theory of λ^{CIL} is non-trivial. Elsewhere, we prove that λ^{CIL} has a standardization property, which effectively determines an operational semantics for λ^{CIL} by specifying an evaluation relation that is a subrelation of the calculus relation (e.g., see (Ariola & Felleisen, 1997)). Taken together, confluence and standardization imply computational soundess: terms equated by the theory can be shown to be observationally equivalent. This is an essential property of an intermediate language, since compiler writers can use the theory to justify an important class of local program transformations.

The remainder of this paper is organized as follows. Section 2 presents an informal overview of λ^{CIL} motivating the properties of the calculus with examples. Section 3 discusses design trade-offs in the formulation of the calculus as well as related work. Section 4 gives a formal presentation of three related calculi: an untyped calculus ($\lambda^{\mathrm{CIL}}_{\mathrm{ut}}$), an explicitly typed calculus (λ^{CIL}), and an implicitly typed calculus ($\lambda^{\mathrm{CIL}}_{\mathrm{i}}$). Section 5 concludes with a brief discussion of our practical experience using a typed intermediate language based on λ^{CIL}. The appendix presents the details of combinatory reduction systems which are used in our proof of confluence.

2 Informal Overview of λ^{CIL}

In this section, we give an informal overview of λ^{CIL} by discussing its syntax and semantics in the context of a series of simple examples. We present two versions of the calculus: the untyped calculus $\lambda^{\mathrm{CIL}}_{\mathrm{ut}}$ and the typed calculus λ^{CIL}.[3] The connection between these two calculi will be developed in section 4.

[3] In section 4 we present yet another language: the implicitly typed language $\lambda^{\mathrm{CIL}}_{\mathrm{i}}$. This calculus is different from the two summarized here. Because $\lambda^{\mathrm{CIL}}_{\mathrm{i}}$ is not critical to the informal exposition, we will not discuss it here.

2.1 The Untyped Calculus $\lambda^{\text{CIL}}_{\text{ut}}$

The untyped language $\lambda^{\text{CIL}}_{\text{ut}}$ is a call-by-value lambda calculus extended with constants, recursion, tuples, and variants. For presentational purpose, we use standard infix primitives in our examples even though they do not appear in the formal calculus. As a first example, consider the following $\lambda^{\text{CIL}}_{\text{ut}}$ term:

$$\hat{M}_a \equiv \textbf{let } f = \lambda x.x \textbf{ in } \times(f \mathbin{@} 17, f \mathbin{@} 23, f \mathbin{@} \textbf{true})$$

In the untyped language, **let** is the familiar syntactic sugar for an application of an abstraction to a term. The body of the **let** expression is a *tuple* containing three applications of the identity function. Tuples are written as $\times(\ldots)$, where the symbol $\times$ serves to distinguish the subterm from a *virtual tuple*, which will be introduced later. Function application is indicated by an explicit @ symbol, which serves as a placeholder for flow labels in the typed versions of the language. The call-by-value reduction rules for $\lambda^{\text{CIL}}_{\text{ut}}$ (see section 4.2.1) are straightforward. Under these rules,$\hat{M}_a$ reduces to the normal form $\times(17, 23, \textbf{true})$.

As a second example, consider:

$$\begin{array}{rl} \hat{M}_b \equiv & \textbf{let } g = \lambda s.\ \textbf{case}^{+}\ s \textbf{ bind } w \textbf{ in} \\ & \qquad\qquad\qquad \times(\lambda x.x + 1, w), \\ & \qquad\qquad\qquad \times(\lambda y.y * 2, w + 1), \\ & \qquad\qquad\qquad \times(\lambda z.\textbf{if } z \textbf{ then } 1 \textbf{ else } 0, w) \\ & \textbf{in}\ \ \textbf{let } h = \lambda a.\ \textbf{let } p = g \mathbin{@} a \\ & \qquad\qquad\qquad \textbf{in}(\pi^{\times}_{1}\ p) \mathbin{@} (\pi^{\times}_{2}\ p) \\ & \quad \textbf{in } \times\big(h \mathbin{@} \big(\textbf{in}^{+}_{1}\ 3\big), h \mathbin{@} \big(\textbf{in}^{+}_{2}\ 5\big), h \mathbin{@} \big(\textbf{in}^{+}_{3}\ \textbf{true}\big)\big) \end{array}$$

In this example, g and h are **let**-bound to functions that take variants as arguments. The term $\pi^{\times}_{i}\ M$ extracts the ith component of the tuple to which M evaluates. Variants are constructed via $\big(\textbf{in}^{+}_{i}\ M\big)$, which injects the value of M into a variant with positional tag i. The symbol $+$ serves to distinguish these from *virtual variants*, which will be introduced later. Variants are deconstructed via $\textbf{case}^{+}$ expressions. In particular, a term of the form $\textbf{case}^{+}\ M_0 \textbf{ bind } x \textbf{ in } M_1, \ldots, M_n$ discriminates on the variant value denoted by M_0, which should reduce to a subterm of the form $\big(\textbf{in}^{+}_{i}\ V\big)$, where $1 \leq i \leq n$, and V is a value (i.e., constant, abstraction, tuple of values, or variant of a value). The value of the $\textbf{case}^{+}$ expression is the value of the clause M_i in a context where x is bound to V (the same bound variable is used in all clauses for convenience). The $\textbf{case}^{+}$ term within g evaluates one of three tuple terms depending on the tag of s. Each of these terms pairs an abstraction with an argument value to which it will be applied (in h). Thus, the tuples have the form of thunks (nullary functions) that have been closure-converted (Dimock *et al.*, 1997). The term $\hat{M}_b$ reduces to the normal form $\times(4, 12, 1)$.

2.2 The Typed Calculus λ^{CIL}

The typed language λ^{CIL} is an extension of $\lambda^{\text{CIL}}_{\text{ut}}$ in which all occurrences of variables (except the binding occurrences of **case**-bound variables) and variants are annotated with a type. λ^{CIL} supports the familiar types: base types, function, product, variant and recursive types. In addition, λ^{CIL} includes two features for representing information that approximates the flow of values from their sources to their sinks: (1) intersection and union types and (2) flow annotations on terms and types.

As an illustration of intersection types, consider the following explicitly typed version of $\hat{M}_a$: [4]

$$\begin{aligned} M_a \equiv\ & \textbf{let } f^{\wedge[\text{int}\to\text{int},\text{bool}\to\text{bool}]} = \bigwedge\big(\lambda x^{\text{int}}.x, \lambda x^{\text{bool}}.x\big) \\ & \textbf{in } \times((\pi_1^{\wedge}\ f)\ @\ 17, (\pi_1^{\wedge}\ f)\ @\ 23, (\pi_2^{\wedge}\ f)\ @\ \textbf{true}) \end{aligned}$$

The term $\bigwedge\big(\lambda x^{\text{int}}.x, \lambda x^{\text{bool}}.x\big)$ denotes a *virtual tuple* — a value of intersection type. Intuitively, a virtual tuple is an entity that represents a polymorphic value as multiple copies of a term that differ only in their type annotations. The virtual projection $\pi_i^{\wedge}\ M$ selects one of the type-annotated copies from the virtual tuple. Virtual tuples and projections are entirely compile-time constructions whose purpose is to facilitate type-checking by tracking the different types at which a polymorphic value is used. All components of a virtual tuple denote the same run-time value; no code will be generated to construct or access the slots of a virtual tuple at run-time. Because λ^{CIL} uses virtual copies of terms as a kind of type annotation, we refer to it as a *duplicating* calculus. An implementation using λ^{CIL} has the responsibility of performing as much sharing as it can between the virtual copies.

Terms and types in λ^{CIL} are annotated with flow labels taken from a given set. To simplify notation, when this set is a singleton, the flow labels are omitted, as they were in the preceding example. As an illustration of non-trivial flow labels, consider another typed term corresponding to $\hat{M}_a$:

$$\begin{aligned} M'_a \equiv\ & \textbf{let } f^{\bigwedge\left[\text{int}\xrightarrow[\{4,5\}]{\{1\}}\text{int},\text{bool}\xrightarrow[\{6\}]{\{3\}}\text{bool}\right]} = \bigwedge\Big(\lambda^1_{\{4,5\}} x^{\text{int}}.x, \lambda^3_{\{6\}} x^{\text{bool}}.x\Big) \\ & \textbf{in } \times\Big(\ (\textbf{coerce}\Big(\text{int} \xrightarrow[\{4,5\}]{\{1\}} \text{int}, \text{int} \xrightarrow[\{4\}]{\{1\}} \text{int}\Big)\ (\pi_1^{\wedge}\ f))\ @_4^{\{1\}}\ 17, \\ & \qquad\quad (\textbf{coerce}\Big(\text{int} \xrightarrow[\{4,5\}]{\{1\}} \text{int}, \text{int} \xrightarrow[\{5\}]{\{1\}} \text{int}\Big)\ (\pi_1^{\wedge}\ f))\ @_5^{\{1\}}\ 23, \\ & \qquad\quad (\pi_2^{\wedge}\ f)\ @_6^{\{3\}}\ \textbf{true}\Big) \end{aligned}$$

Each source term (abstraction) is annotated with a single source label and a set of sink labels that approximate the sink terms (applications) to which values produced at the source may flow. Each sink term is annotated with a single sink label and a set of source labels that approximate the source terms from which the values consumed by the sink may flow. Arrow types are annotated with a set of source labels and a set of sink labels that approximate the sources and sinks of the values that they specify.[5] The **coerce** terms are explicit subtyping coercions that can add source labels to and/or remove sink labels from a type.

[4] To aid readability, the types of most variable occurrences have been elided; they can be determined from the type annotation on the corresponding binding occurrences.

[5] In the calculus of this paper, the only labelled sources are abstractions, the only labelled sinks are applications, and the only labelled types are arrow types. In our compiler implementation, we have extended the calculus to support labelling of all type constructors and their introduction and elimination forms.

An intersection type represents flow information in the sense that it approximates how a value at one point of a program (the intersection term) fans out to other parts of the program (the projection terms). In λ^{CIL}, there must be at least one component of an intersection type for each usage type of a polymorphic value, but the analysis may be even more fine-grained. For example, this is yet another typing of the untyped term $\hat{M}_a$:

$$\begin{aligned} M''_a \equiv\ & \mathbf{let}\ f^{\wedge[\mathrm{int}\to\mathrm{int},\mathrm{int}\to\mathrm{int},\mathrm{bool}\to\mathrm{bool}]} = \bigwedge\left(\lambda x^{\mathrm{int}}.x, \lambda x^{\mathrm{int}}.x, \lambda x^{\mathrm{bool}}.x\right) \\ & \mathbf{in}\ \times((\pi_1^{\wedge}\, f) \mathbin{@} 17, (\pi_2^{\wedge}\, f) \mathbin{@} 23, (\pi_3^{\wedge}\, f) \mathbin{@} \mathbf{true}) \end{aligned}$$

Here there are two virtual copies of the ($\mathrm{int} \to \mathrm{int}$) identity: one destined to be applied at 17, the other destined to be applied at 23. Intersection types can be used in this way to track different flows of *any* value, even one that would be given a monomorphic type by a traditional type system. The ability to represent different analyses of a single term used in multiple contexts is the hallmark of *polyvariant* flow analysis (Nielson & Nielson, 1997; Banerjee, 1997; Jagannathan *et al.*, 1997; Palsberg & Pavlopoulou, 2001; Amtoft & Turbak, 2000). The above example illustrates how intersection types can represent polyvariant flow analyses in λ^{CIL}.

Functions that are type polymorphic in the type system of the source language may become type monomorphic in a representation type system. For example, if the compiler chooses to represent both integer and boolean values as 32-bit words then the types of the components of the virtual tuple in M'_a and M''_a are identical and the virtual tuple may be merged into a single abstraction.

$$\begin{aligned} M'''_a \equiv\ & \mathbf{let}\ f^{\mathrm{word} \xrightarrow[\{2,3,4\}]{\{1\}} \mathrm{word}} = \lambda^1_{\{2,3,4\}} x^{\mathrm{word}}.x \\ & \qquad f_2{}^{\mathrm{word} \xrightarrow[\{2\}]{\{1\}} \mathrm{word}} = \mathbf{coerce}\left(\mathrm{word} \xrightarrow[\{2,3,4\}]{\{1\}} \mathrm{word}, \mathrm{word} \xrightarrow[\{2\}]{\{1\}} \mathrm{word}\right) f \\ & \qquad f_3{}^{\mathrm{word} \xrightarrow[\{3\}]{\{1\}} \mathrm{word}} = \mathbf{coerce}\left(\mathrm{word} \xrightarrow[\{2,3,4\}]{\{1\}} \mathrm{word}, \mathrm{word} \xrightarrow[\{3\}]{\{1\}} \mathrm{word}\right) f \\ & \qquad f_4{}^{\mathrm{word} \xrightarrow[\{4\}]{\{1\}} \mathrm{word}} = \mathbf{coerce}\left(\mathrm{word} \xrightarrow[\{2,3,4\}]{\{1\}} \mathrm{word}, \mathrm{word} \xrightarrow[\{4\}]{\{1\}} \mathrm{word}\right) f \\ & \mathbf{in}\ \times\left(f_2 \mathbin{@}^{\{1\}}_2 17, f_3 \mathbin{@}^{\{1\}}_3 23, f_4 \mathbin{@}^{\{1\}}_4 \mathbf{true}\right) \end{aligned}$$

The coercions in the above example can be interpreted as moves of the code pointer of the function to the registers used at the respective call sites.

Although the polymorphism in the above examples can be expressed in the Hindley-Milner system (Damas & Milner, 1982), intersection types can represent polymorphism that is not expressible in **let**-style polymorphism. For example, a polymorphic function can be returned as a result or passed as an argument as in the following example.

$$\begin{aligned} M_c \equiv\ & \mathbf{let}\ p^{\wedge[\wedge[\tau_1,\tau_2]\to\times[\mathrm{int},\mathrm{int},\mathrm{bool}],\wedge[\tau_3,\tau_4]\to\times[\mathrm{real},\mathrm{real},\mathrm{real}]]} = \\ & \quad \bigwedge(\ \lambda f^{\wedge[\tau_1,\tau_2]}.\times((\pi_1^{\wedge}\, f) \mathbin{@} 17, (\pi_1^{\wedge}\, f) \mathbin{@} 23, (\pi_2^{\wedge}\, f) \mathbin{@} \mathbf{true}), \\ & \qquad \lambda f^{\wedge[\tau_3,\tau_4]}.\times((\pi_1^{\wedge}\, f) \mathbin{@} 17, (\pi_1^{\wedge}\, f) \mathbin{@} 23, (\pi_2^{\wedge}\, f) \mathbin{@} \mathbf{true})) \\ & \mathbf{in} \\ & \quad \times(\ (\pi_1^{\wedge}\, p) \mathbin{@} \bigwedge\left(\lambda x^{\mathrm{int}}.x, \lambda x^{\mathrm{bool}}.x\right), \\ & \qquad (\pi_2^{\wedge}\, p) \mathbin{@} \bigwedge\left(\lambda y^{\mathrm{int}}.3.141, \lambda y^{\mathrm{bool}}.3.141\right)) \end{aligned}$$

where $\tau_1 \equiv \mathrm{int} \to \mathrm{int}$, $\tau_2 \equiv \mathrm{bool} \to \mathrm{bool}$, $\tau_3 \equiv \mathrm{int} \to \mathrm{real}$ and $\tau_4 \equiv \mathrm{bool} \to \mathrm{real}$. There are two levels of polymorphism here: one for the identity and constant functions, and one for the function p that is applied to these functions.

Whereas intersection types represent fan-out in flow paths (i.e., a value that flows to multiple destinations), union types represent fan-in of flow paths (i.e., multiple values flowing to a single destination). Union types are necessary for expressing the untyped sample term $\hat{M}_b$ in λ^{CIL}:

$$
\begin{array}{lll}
M_b & \equiv & \textbf{let } g^{+[\mathrm{int},\mathrm{int},\mathrm{bool}] \to \vee[\rho_1,\rho_2]} = \\
& & \quad \lambda s^{+[\mathrm{int},\mathrm{int},\mathrm{bool}]}.\ \textbf{case}^{+}\ s\ \textbf{bind}\ w\ \textbf{in} \\
& & \qquad \mathrm{int} \Rightarrow \left(\mathbf{in}_1^{\vee} \times \left(\lambda x^{\mathrm{int}}.x+1, w^{\mathrm{int}}\right)\right)^{\vee[\rho_1,\rho_2]}, \\
& & \qquad \mathrm{int} \Rightarrow \left(\mathbf{in}_1^{\vee} \times \left(\lambda y^{\mathrm{int}}.y*2, w^{\mathrm{int}}+1\right)\right)^{\vee[\rho_1,\rho_2]}, \\
& & \qquad \mathrm{bool} \Rightarrow \left(\mathbf{in}_2^{\vee} \times \left(\lambda z^{\mathrm{bool}}.\textbf{if}\ z\ \textbf{then}\ 1\ \textbf{else}\ 0, w^{\mathrm{bool}}\right)\right)^{\vee[\rho_1,\rho_2]} \\
& & \textbf{in}\quad \textbf{let } h^{+[\mathrm{int},\mathrm{int},\mathrm{bool}] \to \mathrm{int}} = \\
& & \qquad \lambda a^{+[\mathrm{int},\mathrm{int},\mathrm{bool}]}.\ \textbf{let } p^{\vee[\rho_1,\rho_2]} = g\ @\ a \\
& & \qquad\qquad\qquad\qquad \textbf{in case}^{\vee}\ p\ \textbf{bind}\ r\ \textbf{in} \\
& & \qquad\qquad\qquad\qquad\quad \rho_1 \Rightarrow (\pi_1^{\times}\ r^{\rho_1})\ @\ (\pi_2^{\times}\ r^{\rho_1}) \\
& & \qquad\qquad\qquad\qquad\quad \rho_2 \Rightarrow (\pi_1^{\times}\ r^{\rho_2})\ @\ (\pi_2^{\times}\ r^{\rho_2}) \\
& & \qquad \textbf{in} \times\left(h\ @\ \left(\mathbf{in}_1^{+}\ 3\right), h\ @\ \left(\mathbf{in}_2^{+}\ 5\right), h\ @\ \left(\mathbf{in}_3^{+}\ \textbf{true}\right)\right)
\end{array}
$$

where $\rho_1 \equiv \times[\mathrm{int} \to \mathrm{int}, \mathrm{int}]$ and $\rho_2 \equiv \times[\mathrm{bool} \to \mathrm{int}, \mathrm{bool}]$. In λ^{CIL}, each clause a of $\textbf{case}^{+}$ term and a $\textbf{case}^{\vee}$ term is introduced with the notation $\tau \Rightarrow$. This notation indicates that the bound variable declared by the **case** term has type τ within the clause. The two incompatible types returned by the body of g are merged into the union type $\vee[\rho_1, \rho_2]$. Terms of union type are constructed by injecting a term into a *virtual variant*. Virtual variants are analyzed by *virtual cases* (i.e., $\textbf{case}^{\vee}$ terms), which are the duals of virtual tuples. A virtual case contains multiple copies of clauses that differ only in their type annotations. As with intersection components, the case analysis of a $\textbf{case}^{\vee}$ is a compile-time operation that implies no run-time computation. All the clauses of a $\textbf{case}^{\vee}$ represent the same computation.

Our framework requires that differently typed values flowing to a polymorphic context must be injected into virtual variants of the same union type with different virtual tags. However, finer grained flow can be represented by injecting values of the same type into values of the same union type with different virtual tags. For instance, in the above example, $\times\left(\lambda x^{\mathrm{int}}.x+1, w^{\mathrm{int}}\right)$ and $\times\left(\lambda y^{\mathrm{int}}.y*2, w^{\mathrm{int}}+1\right)$ could be injected with different virtual tags, which would allow customization of the corresponding $\textbf{case}^{\vee}$ clauses to be made based on flow information finer-grained than the type information.

2.3 An Example of Flow-Based Customization

By combining the fan-out flow of intersection types with the fan-in flow of union types, it is possible to construct networks of flow paths connecting the sources and sinks of values. These flow path networks can guide flow-based customization (Dimock *et al.*, 1997).

As a concrete example of such customization, we illustrate how flow types can guide an uncurrying transformation. Consider the following $\lambda^{\mathrm{CIL}}_{\mathrm{ut}}$ term:

$$\begin{array}{rll} \hat{M}_d \equiv & \mathbf{let} & k{=}\lambda x.\lambda y.x \\ & & g{=}\lambda x.\ \mathbf{let}\ y = x * x\ \mathbf{in}\ \lambda z.y \\ & & h{=}\lambda f.f \mathbin{@} 1 \mathbin{@} 2 \\ & \mathbf{in} & \times(k \mathbin{@} 3 \mathbin{@} 4, h \mathbin{@} k, h \mathbin{@} g) \end{array}$$

The function named k is a curried function of two arguments. We want to investigate transformations that would allow uncurrying this function to the form $\lambda[x,y].x$[6]. Of course, any application sites to which k flows must also be transformed in a consistent manner. Matters are complicated by the fact that k can flow to application sites to which other functions can flow. For example, both k and the function g, which cannot be uncurried, are arguments to the h function. In traditional approaches to uncurrying, e.g., (Appel, 1992; Tarditi, 1996; Hannan & Hicks, 1998), this fact would prohibit k from being uncurried. This is an instance of a *representation pollution problem*, in which (1) the assumption that every source term has a single representation for all usage contexts and (2) the fact that an unoptimizable representation flows to some usage context together preclude using customized representations for some of the contexts.

Flow types are an effective language for addressing the pollution problem. We show this by presenting two strategies for customizations based on the following λ^{CIL} term, which is a typing for the untyped term above:

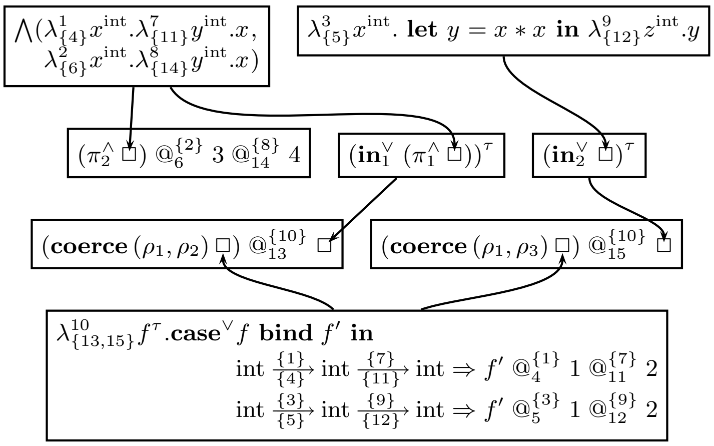
$$\begin{array}{rll} M_d \equiv & \mathbf{let} & k{=}\bigwedge\Big(\lambda^{1}_{\{4\}}x^{\mathrm{int}}.\lambda^{7}_{\{11\}}y^{\mathrm{int}}.x,\ \lambda^{2}_{\{6\}}x^{\mathrm{int}}.\lambda^{8}_{\{14\}}y^{\mathrm{int}}.x\Big) \\ & & g{=}\Big(\mathbf{in}^{\vee}_{2}\ (\lambda^{3}_{\{5\}}x^{\mathrm{int}}.\ \mathbf{let}\ y = x * x\ \mathbf{in}\ \lambda^{9}_{\{12\}}z^{\mathrm{int}}.y)\Big)^{\tau} \\ & & h{=}\lambda^{10}_{\{13,15\}}f^{\tau}.\mathbf{case}^{\vee} f\ \mathbf{bind}\ f'\ \mathbf{in} \\ & & \qquad\qquad \mathrm{int} \xrightarrow[\{4\}]{\{1\}} \mathrm{int} \xrightarrow[\{11\}]{\{7\}} \mathrm{int} \Rightarrow f' \mathbin{@}^{\{1\}}_{4} 1 \mathbin{@}^{\{7\}}_{11} 2 \\ & & \qquad\qquad \mathrm{int} \xrightarrow[\{5\}]{\{3\}} \mathrm{int} \xrightarrow[\{12\}]{\{9\}} \mathrm{int} \Rightarrow f' \mathbin{@}^{\{3\}}_{5} 1 \mathbin{@}^{\{9\}}_{12} 2 \\ & \mathbf{in} & \times((\pi^{\wedge}_{2}\ k) \mathbin{@}^{\{2\}}_{6} 3 \mathbin{@}^{\{8\}}_{14} 4, \\ & & \quad (\mathbf{coerce}\,(\rho_1,\rho_2)\,h) \mathbin{@}^{\{10\}}_{13} \big(\mathbf{in}^{\vee}_{1}\ (\pi^{\wedge}_{1}\ k)\big)^{\tau}, \\ & & \quad (\mathbf{coerce}\,(\rho_1,\rho_3)\,h) \mathbin{@}^{\{10\}}_{15}\ g) \end{array}$$

where $\tau = \bigvee\Big[\mathrm{int} \xrightarrow[\{4\}]{\{1\}} \mathrm{int} \xrightarrow[\{11\}]{\{7\}} \mathrm{int},\ \mathrm{int} \xrightarrow[\{5\}]{\{3\}} \mathrm{int} \xrightarrow[\{12\}]{\{9\}} \mathrm{int}\Big]$, $\rho_1 = \tau \xrightarrow[\{13,15\}]{\{10\}} \mathrm{int}$, $\rho_2 = \tau \xrightarrow[\{13\}]{\{10\}} \mathrm{int}$ and $\rho_3 = \tau \xrightarrow[\{15\}]{\{10\}} \mathrm{int}$. The typed abstractions $\lambda^{1}_{\{4\}}x^{\mathrm{int}}.\lambda^{7}_{\{11\}}y^{\mathrm{int}}.x$ and $\lambda^{2}_{\{6\}}x^{\mathrm{int}}.\lambda^{8}_{\{14\}}y^{\mathrm{int}}.x$ are components of a virtual tuple that represent the untyped abstraction $\lambda x.\lambda y.x$. (The correspondence between typed and untyped expressions is formalized by a notion of type erasure presented later in section 4.3.1.) The virtual tuple represents a polyvariant flow analysis in which the abstraction is analyzed at

[6] The deconstruction of a tuple argument via pattern matching is not supported by the formal $\lambda^{\mathrm{CIL}}_{\mathrm{ut}}$ syntax, but is used informally here to highlight the connection between the curried and uncurried forms of the function.

two call sites. The typed applications $f' \mathbin{@}_4^{\{1\}} 1 \mathbin{@}_{11}^{\{7\}} 2$ and $f' \mathbin{@}_5^{\{3\}} 1 \mathbin{@}_{12}^{\{9\}} 2$ are the code parts of clauses of a virtual **case** expression. They represent the untyped applications $\lambda f.f \mathbin{@} 1 \mathbin{@} 2$. The virtual **case** expression represents a polyvariant flow analysis in which the application $\lambda f.f \mathbin{@} 1 \mathbin{@} 2$ is analyzed with respect to the two abstractions k and g.

To better highlight the flow-based nature of the customizations, the typed term is presented using a more graphical notation in figure 1. In this notation, arrows denote the flow of values from their source to their destinations. The product combining the components of the resulting triple has also been elided.



where

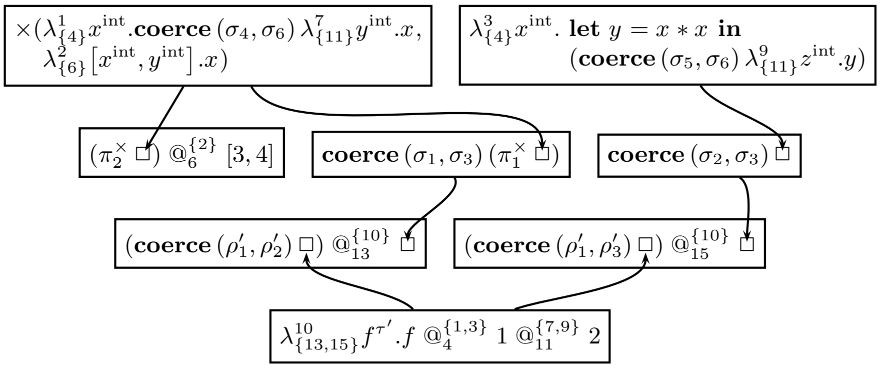
$$\tau = \bigvee\Big[\mathrm{int} \xrightarrow[\{4\}]{\{1\}} \mathrm{int} \xrightarrow[\{11\}]{\{7\}} \mathrm{int},\ \mathrm{int} \xrightarrow[\{5\}]{\{3\}} \mathrm{int} \xrightarrow[\{12\}]{\{9\}} \mathrm{int}\Big]$$

$$\rho_1 = \tau \xrightarrow[\{13,15\}]{\{10\}} \mathrm{int},\ \rho_2 = \tau \xrightarrow[\{13\}]{\{10\}} \mathrm{int},\ \rho_3 = \tau \xrightarrow[\{15\}]{\{10\}} \mathrm{int}$$

Fig. 1. Uncurrying Term before Transformation.

Using λ^{CIL}, the representation pollution in the uncurrying example can be resolved by one of two of *splitting* strategies. The first strategy, which we call *source splitting*, is to reify the virtual tuple as a real tuple whose two components have different representations (see figure 2). The component flowing to application site 6 has been uncurried, while the component flowing to application site 4 has been left in a curried form to match the type of the other function flowing there. The virtual projections associated with the reified virtual tuple have also been reified as real projections. The virtual **case** expression and its associated injections have been eliminated as a result of merging case clauses that manipulate the same representation.

The second strategy, which we call *sink splitting*, is to reify the virtual **case** expression and its associated injections. The injections tag the two incompatible function representations that will arrive at the application sites 4 and 5 (figure 3). In this case, the two components of the virtual tuple are merged into a single uncurried abstraction labelled 1. The flow path from this abstraction to the discriminant of the reified **case** expression includes an injection that tags the abstraction, in order distinguish it from the curried abstraction 3. The flow path from abstraction 1 to call site 6 does not need to be tagged, since no other representation reaches that site.

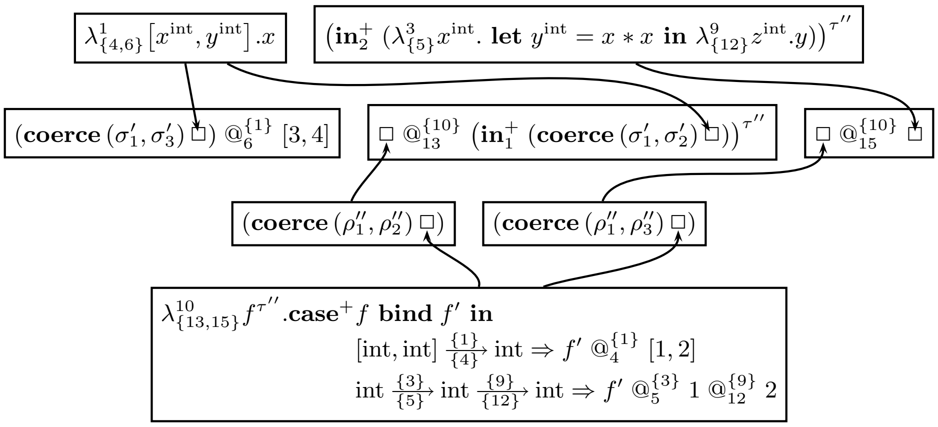


$$\sigma_1 = \text{int} \xrightarrow[\{4\}]{\{1\}} \text{int} \xrightarrow[\{11\}]{\{7,9\}} \text{int},\ \sigma_2 = \text{int} \xrightarrow[\{4\}]{\{3\}} \text{int} \xrightarrow[\{11\}]{\{7,9\}} \text{int},\ \sigma_3 = \text{int} \xrightarrow[\{4\}]{\{1,3\}} \text{int} \xrightarrow[\{11\}]{\{7,9\}} \text{int}$$

$$\sigma_4 = \text{int} \xrightarrow[\{11\}]{\{7\}} \text{int},\ \sigma_5 = \text{int} \xrightarrow[\{11\}]{\{9\}} \text{int},\ \sigma_6 = \text{int} \xrightarrow[\{11\}]{\{7,9\}} \text{int},\ \tau' = \text{int} \xrightarrow[\{4\}]{\{1,3\}} \text{int} \xrightarrow[\{11\}]{\{7,9\}} \text{int}$$

$$\rho'_1 = \tau \xrightarrow[\{13,15\}]{\{10\}} \text{int},\ \rho'_2 = \tau \xrightarrow[\{13\}]{\{10\}} \text{int},\ \rho'_3 = \tau \xrightarrow[\{15\}]{\{10\}} \text{int}$$

Fig. 2. Uncurrying term with split sources and merged sinks.



$$\sigma'_1 = [\text{int},\text{int}] \xrightarrow[\{4,6\}]{\{1\}} \text{int},\ \sigma'_2 = [\text{int},\text{int}] \xrightarrow[\{4\}]{\{1\}} \text{int},\ \sigma'_3 = [\text{int},\text{int}] \xrightarrow[\{6\}]{\{1\}} \text{int}$$

$$\tau'' = +\left[[\text{int},\text{int}] \xrightarrow[\{4\}]{\{1\}} \text{int},\ \text{int} \xrightarrow[\{5\}]{\{3\}} \text{int} \xrightarrow[\{12\}]{\{9\}} \text{int}\right]$$

$$\rho''_1 = \tau'' \xrightarrow[\{13,15\}]{\{10\}} \text{int},\ \rho''_2 = \tau'' \xrightarrow[\{13\}]{\{10\}} \text{int},\ \rho''_3 = \tau'' \xrightarrow[\{15\}]{\{10\}} \text{int}$$

Fig. 3. Uncurrying term with split sinks and merged sources.

In systems that enforce one representation per source or sink term, the representation of a single unoptimizable term can prohibit optimizations elsehwere in the program by dictating the representation of any term connected to it by some sequence of flow paths. The splitting strategies sketched above can contain this representation pollution by allowing a single source or sink term to be implemented with multiple representations: some optimized, some not. As illustrated by the example, the program plumbing information represented in λ^{CIL} terms and types allows the splitting strategies to be implemented by reifying virtual terms. Which strategy to use in practice depends on heuristics followed by the compiler; different strategies can be used in different parts of the same program.

The data customization illustrated by the above example is only one of many optimizations enabled by a flow type system. The source and sink labels in λ^{CIL} allow definition and use points to be matched up as required by conventional dataflow-based optimizations. Indeed, when the introduction and elimination forms for primitive data types are annotated with flow information, typed terms carry information similar to *static single-assignment* (SSA) form (Cytron *et al.*, 1991; Briggs *et al.*, 1998): source labels in λ^{CIL} play the role of SSA's uniquely named source points. A ϕ-node denotes a confluence of flow paths. In λ^{CIL} these points can be represented either as union **case** forms or as coercions. Unlike SSA, λ^{CIL} supports the dual notions of sink labels and intersection types.

3 Design Issues

In this section, we discuss the rationale behind various decisions made in the design of λ^{CIL}. Along the way, we also discuss related calculi, type systems, and intermediate languages.

3.1 Explicit Types

In each stage of type-directed compilation, it is important to be able to verify that terms are well typed and to use these types to guide translations. In order to derive a type for a term whenever needed, λ^{CIL} annotates variables, variants and coercions with explicit types. Such type annotations can impair readability, but this is not a major drawback since λ^{CIL} is intended to be an intermediate language, not a source language.

Since compiler transformations may produce typings beyond the range of computable automatic type inference, automatic type inference is also not a goal of our design. We assume that any type inference is performed before or as part of the translation from the source language into the intermediate language. Recent work on encoding flow information via intersection and union types (Palsberg & Pavlopoulou, 2001; Amtoft & Turbak, 2000) demonstrates how to automatically translate between flow analyses and type derivations similar to those of λ^{CIL} terms. Modulo the issue of shallow subtyping (see section 3.5), these translations suggest a simple approach to type inference in λ^{CIL}: perform one of several popular flow analyses for a term and then translate the flow analysis into a λ^{CIL} term.

3.2 Finitary Polymorphic Types

A central goal of our work is to encode precise information obtained from program analysis into the type systems of TILs. This information should be conveniently accessible for use in guiding program transformations. In the case of λ^{CIL}, types are annotated with information that tracks the flow of functions between abstractions and applications in order to support customization at these sites.

Some type system designs conflict with the goal of encoding precise program analysis information within the type system in an accessible manner. For example, type polymorphism for functions is usually represented by abstractions over types, which by themselves do not specify the types of arguments at which such functions may be called. This information *is* available in the typing derivation of the whole program, but it is not convenient to access. Dually, abstract data types are typically encoded by existential types, which do not directly provide representation information to the clients of such abstractions. In effect, universal and existential types are a *promise* of a very general implementation — a promise kept by boxing. The dynamic-dispatch problem of object-oriented languages is similar to boxing; in both cases, a wrapper is used to access potentially incompatible representations via a single protocol.

In order to expose concrete type information hidden by universal and existential types, λ^{CIL} supports type-polymorphic functions with intersection types and abstract data types with union types. An intersection type lists the concrete types at which a polymorphic function may be used in a particular program. It is the finitary (listing-based) version of the infinitary (schema-based) universal type, which corresponds to an infinite intersection of types. Dually, the finitary versions of infinitary existential types are union types, which list the concrete types of the implementations of an abstract data type.

There are several advantages of using intersection and union types in place of universal and existential types. First, they enable customization by indicating the types at which polymorphic values are used. Second, they can encode data flow; intersection types represent the possible destinations of a value while union types represent its possible sources (Palsberg & Pavlopoulou, 2001; Amtoft & Turbak, 2000). This enables type-based customizations that are more fine-grained than those possible using System F types; flow types can distinguish usage contexts for a value with a given System F type. Finally, for certain classes of languages, finitary polymorphism is strictly more powerful than infinitary polymorphism, in the sense that it can type more terms. Intersection types can type every strongly normalizing lambda calculus term, while the terms typable in System F are a proper subset of the strongly normalizing terms. As a concrete example, consider the term

$$(\lambda x.z(x(\lambda fu.fu))(x(\lambda vg.gv)))(\lambda y.yyy)\ .$$

This term is shown in (Urzyczyn, 1997) to be untypable in System F_ω, considered to be the most powerful type system with universal quantifiers. In contrast, it is typable not only in λ^{CIL}, but even in a very limited version of λ^{CIL} satisfying the so-called *rank 3* restriction (Kfoury & Wells, 1999). For extensions to the lambda calculus that model more programming language features (such as term-level recursion), the relationship between finitary and infinitary forms of polymorphism is not known, but it is likely that the finitary forms of polymorphism type more terms than the infinitary ones. The intuition behind this claim is that finitary polymorphism requires proving properties for a finite list of types whereas infinitary polymorphism effectively requires proving properties for an infinite number of types, and, in general, it is easier to prove a finite set of properties than an infinite set.

As noted in Section 1, there are several disadvantages of using finitary polymorphism, including larger terms and types and an assumption of whole-program compilation. It may be possible to address some of these drawbacks by combining finitary and infinitary polymorphic types into a single type system.

It is worthwhile to compare the finitary polymorphism of λ^{CIL}'s flow types to other approaches for exposing the concrete type information hidden by infinitary polymorphism. A common technique for improving polymorphic functions is *type specialization*, which makes monomorphic copies of a polymorphic function for the types at which it is (or might be) used. Type specialization of polymorphic functions is usually achieved either by a *monomorphization* pass that removes all polymorphism (Tolmach & Oliva, 1998; Benton *et al.*, 1998; Cejtin *et al.*, 2000) or by aggressive inlining (Tarditi *et al.*, 1996). Type specialization has also been used to compile data parallelism (Blelloch, 1993), to resolve overloading in Haskell (Jones, 1994), to optimize method invocation in object-oriented languages (Chambers & Ungar, 1989a; Chambers & Ungar, 1989b; Agesen, 1995; Dean *et al.*, 1995; Chambers *et al.*, 1996; Plevyak & Chien, 1995; Plevyak, 1996). Although the copying implied by type specialization introduces the threat of code blowup, the increase in code size observed in practice is often quite modest due to the fact that the specialized code is more amenable to traditional optimizations (Jones, 1994; Tolmach & Oliva, 1998; Benton *et al.*, 1998; Cejtin *et al.*, 2000).

The finitary polymorphism of λ^{CIL} is similar to the monomorphization approach to type specialization except for two key differences. First, because flow types encode flow information, they permit flow-based specializations that are more fine-grained than the type-based specializations of traditional type systems. For instance, a higher-order filtering function can be specialized according to the filtering predicate in addition to the element type of the filtered list. In this case, similar specialization can be achieved by inlining the filtering function at each call site. But the flow-based specialization can be performed even in situations where inlining is not performed (e.g., when multiple functions flow to the operator position of a given call site).

Second, whereas monomorphization effectively commits to duplicate copies of code at run-time, virtual tuples and case clauses only represent the *potential* of run-time duplication. If that potential is not realized, no duplicate code is created. For example, suppose that a filtering function is applied to both a list of integers and a list of characters. Monomorphizing this code based on source language types would yield two run-time copies even though the machine-level representation of integers and characters might be exactly the same. In contrast, if a virtual tuple containing the two copies of the filtering function were not reified by the compiler, only a single run-time copy would be generated. To reduce the number of unnecessary copies of polymorphic functions, monomorphizing compilers typically instantiate polymorphic functions to machine-level types rather than source language types.

Another approach to removing the overhead of polymorphism is *dynamic type dispatch*, in which a polymorphic function can dispatch to monomorphic code based on a type argument that is passed separately from a value argument of that type. In the TIL compiler, polymorphic functions are handled efficiently by dispatching to monomorphic code based on all possible representation types at which the function can be used (Harper & Morrisett, 1995; Morrisett, 1995). Although the type dispatch in general may take place at run-time, it usually can be performed at compile-time, yielding code without a run-time overhead. Duggan's *refinement kinds* extend dynamic type dispatch to user-defined types (Duggan, 1999). The main advantage of dynamic type dispatch over finitary polymorphism is that it is compatible with separate compilation. However, this strength turns into a weakness when it comes to customization. The range of representation types must be fixed in advance to allow separately compiled modules to interface with each other, and representations for values exported by a module must be chosen without any knowledge of how those values might be used in other modules.

3.3 Explicit Syntax for Intersection and Union Types

Systems with intersection types are ordinarily implicitly typed using the following typing rule for introducing intersection types:

$$\frac{A \vdash M : \sigma;\ A \vdash M : \tau}{A \vdash M : \sigma \wedge \tau}\ (\wedge\ \text{intro})$$

This typing rule is incompatible with the decision to annotate variables with explicit types. For instance, how can we show that an identity function has the type $(\text{int} \to \text{int}) \wedge (\text{bool} \to \text{bool})$? The derivation might look something like:

$$\frac{A \vdash \lambda x^{\text{int}}.x : (\text{int} \to \text{int});\ A \vdash \lambda x^{\text{bool}}.x : (\text{bool} \to \text{bool})}{A \vdash \lambda x^{???}.x : (\text{int} \to \text{int}) \wedge (\text{bool} \to \text{bool})}$$

There are two problems with this approach:

1. The derivation does not match the ($\wedge$ intro) rule above because there is not a single M, but three different versions of M that differ only in type annotations.
2. It is not clear how to annotate the bound variable(s) in a term of intersection type (as reflected by the ??? in the example).

The first problem can be solved by a rule that uses three terms that are not the same but the same modulo type annotations:

$$\frac{\begin{array}{l}A \vdash M_1 : \sigma;\ A \vdash M_2 : \tau; \\ M_3 \text{ is the "combination" of } M_1 \text{ and } M_2; \\ M_1,\ M_2, \text{ and } M_3 \text{ are "the same modulo type annotations"}\end{array}}{A \vdash M_3 : \sigma \wedge \tau}$$

In section 4.3, we introduce a notion of *type erasure* that formalizes the notion of "the same modulo type annotations".

With regard to the second problem, there are several approaches to dealing with the problem of annotating bound variables for terms of intersection type. The approach used by Reynolds in the language Forsythe (Reynolds, 1996) annotates the binding of an abstraction $(\lambda x.M)$ with a list of possible types, as in $(\lambda x{:}\,\sigma_1|\cdots|\sigma_n.M)$. If the body M of the abstraction is typable with the same type τ for each possible type σ_i of the bound variable x, then the abstraction is assigned the type $(\sigma_1 \to \tau) \wedge \cdots \wedge (\sigma_n \to \tau)$. However, this method is not sufficient to represent dependencies between the types of nested variable bindings. For instance, $(\lambda x.\lambda y.x)$ cannot be given the type $(\sigma \to (\sigma \to \sigma)) \wedge (\tau \to (\tau \to \tau))$.

Pierce uses a more general term-level **for** construct specifying that a type variable ranges over the types in a finite set (Pierce, 1991). For example, using this method the term $(\lambda x.\lambda y.x)$ can be annotated as $(\mathbf{for}\ \alpha \in \{\sigma,\tau\}.\lambda x{:}\,\alpha.\lambda y{:}\,\alpha.x)$, which has the type $(\sigma \to (\sigma \to \sigma)) \wedge (\tau \to (\tau \to \tau))$. However, this method is insufficient to represent some typings, such as giving the term $\hat{M}_f \equiv \lambda x.\lambda y.\lambda z.(xy, xz)$ the type $(((\alpha \to \delta) \wedge (\beta \to \epsilon)) \to \alpha \to \beta \to (\delta \times \epsilon)) \wedge ((\gamma \to \gamma) \to \gamma \to \gamma \to (\gamma \times \gamma))$. By extending Pierce's **for** notation with a notion of simultaneous substitution of type variables, it is possible to handle more complex dependencies. For example, an explicitly typed version of $\hat{M}_f$ can be written:

$$\begin{array}{l}\mathbf{for}\ \{[\theta \mapsto \alpha, \kappa \mapsto \beta, \eta \mapsto \delta, \nu \mapsto \epsilon], [\theta \mapsto \gamma, \kappa \mapsto \gamma, \eta \mapsto \gamma, \nu \mapsto \gamma]\}. \\ \quad \lambda x : (\theta \to \eta) \wedge (\kappa \to \nu)\,.\,\lambda y : \theta\,.\,\lambda z : \kappa\,.\,(xy, xz)\end{array}$$

There are two problems with both the original and extended **for** notations. First, they depend on intersection types being associative, commutative, and idempotent (ACI). But recent research suggests that non-ACI intersection and union types are needed to faithfully encode flow analyses (Amtoft & Turbak, 2000). Second, the type information does not satisfy the convenient accessibility goal; it is not locally obvious but is determined by enclosing type variable bindings. Although the types can be instantiated by a tree-walking process, this is not a convenient representation for flow-based compiler transformations, which need to reference terms at arbitrary locations in the program by their source and sink program points.

In λ^{CIL}, we solve the two problems with a new approach for giving explicit type annotations to terms of intersection type. Since every implicitly typed term of intersection type must have a type derivation tree, we can encode the structure of the type derivation tree in the term itself. That is, we treat each term of intersection type as a combination of component terms (which must be the same modulo type annotations) whose types are combined to form the intersection type.

$$\frac{A \vdash M_1 : \sigma;\ A \vdash M_2 : \tau;\quad M_1 \text{ and } M_2 \text{ are "the same modulo type annotations"}}{A \vdash \wedge(M_1, M_2) : \sigma \wedge \tau}$$

We call the term $\wedge(M_1, M_2)$ a *virtual tuple* and prefix it with the "$\wedge$" symbol to distinguish it from an ordinary tuple. The intended meaning is that M_1, M_2, and $\wedge(M_1, M_2)$ are merely different type-annotated versions of the *same* untyped term. We also introduce an explicit projection $\pi_i^\wedge$ to extract a component out of a value of intersection type:

$$\frac{A \vdash M : \wedge(\tau_1, \ldots, \tau_n);\ 1 \le i \le n}{A \vdash \pi_i^\wedge\, M : \tau_i}\ (\wedge\ \text{elim})$$

An implication of this approach is that constructors for intersection types and virtual tuples are not ACI.

Recording all type derivation choices in the syntax of λ^{CIL} makes it possible to use ordinary type annotations on variable bindings within each component of a virtual tuple. For example, below are the $\lambda^{\text{CIL}}_{\text{ul}}$ type-annotated terms (without flow labels) for several examples considered above. Note how type information is locally accessible at each term.

Untyped Term	$\lambda^{\text{CIL}}_{\text{ul}}$ **Type**	$\lambda^{\text{CIL}}_{\text{ul}}$ **Term**
$\lambda x.x$	$\wedge[\text{int} \to \text{int}, \text{bool} \to \text{bool}]$	$\bigwedge(\lambda x^{\text{int}}.x^{\text{int}}, \lambda x^{\text{bool}}.x^{\text{bool}})$
$\lambda x.\lambda y.x$	$\wedge[\sigma \to (\sigma \to \sigma),$ $\tau \to (\tau \to \tau)]$	$\wedge(\lambda x^\sigma.\lambda y^\sigma.x^\sigma, \lambda x^\tau.\lambda y^\tau.x^\tau)$
$\lambda x.$ $\lambda y.$ $\lambda z.\times(x @ y, x @ z)$	$\bigwedge[\wedge[\alpha \to \delta, \beta \to \epsilon]$ $\to \alpha \to \beta \to \times[\delta, \epsilon],$ $(\gamma \to \gamma) \to$ $\gamma \to \gamma \to \times[\gamma, \gamma]]$	$\bigwedge(\lambda x^{\wedge[\alpha\to\delta,\beta\to\epsilon]}.\lambda y^\alpha.\lambda z^\beta.$ $\times((\pi_1^\wedge\, x^{\wedge[\alpha\to\delta,\beta\to\epsilon]}) @ y^\alpha,$ $(\pi_2^\wedge\, x^{\wedge[\alpha\to\delta,\beta\to\epsilon]}) @ z^\beta),$ $\lambda x^{\gamma\to\gamma}.\lambda y^\gamma.\lambda z^\gamma.$ $\times(x^{\gamma\to\gamma} @ y^\gamma, x^{\gamma\to\gamma} @ z^\gamma))$

We emphasize that virtual tuple constructors and projections are purely compile-time notions introduced for typing purposes. At run-time, computation is essentially performed on the single untyped term that is the type erasure of all the type-annotated components of a virtual tuple.

In λ^{CIL}, terms of union type are handled in a manner dual to terms of intersection type. An explicit injection $\mathbf{in}_i^\vee$ is used to create a *virtual variant.* If M_0 denotes a virtual variant (i.e., is a term of union type) then it is discriminated on via the construct

$$\mathbf{case}^\vee\, M_0\ \mathbf{bind}\ x\ \mathbf{in}\ \tau_1 \Rightarrow M_1, \ldots, \tau_n \Rightarrow M_n$$

where x is bound to the "untagged" portion of the variant at type τ_i within term M_i. The purpose of $\mathbf{case}^{\vee}$ is to encode the type derivation tree for unions within the term structure of λ^{CIL}. Since the terms $M_1, \ldots, M_n$ represent the same run-time term, they must be be the same modulo type annotations.

An advantage of recording type derivation choices in the syntax of λ^{CIL} is that it simplifies the expression of representation transformations that use flow information to transform the sources and sinks of values to be consistent with a changes in the representations of those values. For example, even though a polymorphic function used at two different types is a single value, it is possible that a type-directed transformation will transform the function in incompatible ways for each type, in which case it must be represented as a pair of function values. With λ^{CIL}, this sort of transformation can easily be expressed by transforming a virtual tuple to a real tuple — i.e., changing the appropriate occurrences of $\wedge$ and $\pi_i^{\wedge}$ to $\times$ and $\pi_i^{\times}$, respectively as shown in Figure 2. Similarly, λ^{CIL} faciliates transforming virtual variants to real variants as shown in Figure 3.

One drawback of our approach to handling terms of intersection and union type is that reduction of typed terms must essentially work on typing derivations, a notion that is non-trivial to formulate. Since all the components of a virtual tuple stand for the same run-time term, any computation step in one component of a virtual tuple must be taken in parallel by *all* components of the virtual tuple. A similar constraint holds for the clauses of a $\mathbf{case}^{\vee}$. Section 4.3 introduces *parallel contexts* to formalize this notion of parallel computation step.

3.4 Explicit Coercions

Explicit subtyping coercions are another example of how aspects of type derivations are recorded in the term syntax of λ^{CIL}. The usual rule for subtyping is

$$\frac{A \vdash M : \sigma;\ \sigma \leq \tau}{A \vdash M : \tau}\ (\text{subsumption})$$

In λ^{CIL}, all uses of subtyping are indicated by an explicit **coerce**:

$$\frac{A \vdash M : \sigma;\ \sigma \leq \tau}{A \vdash \mathbf{coerce}\,(\sigma, \tau)\,M : \tau}\ (\text{coerce})$$

Explicit coercions can facilitate the expression of representation transformations. Given a source term in which $\sigma \leq \tau$, a type-directed transformation T may produce a target term in which $T[\sigma] \not\leq T[\tau]$. In such a target term, the subtyping coercion of the source term may be represented by manipulations of run-time data structures. Even though it implies no run-time overhead, an explicit subtyping coercion in the source term records the position at which a transformation may insert code that performs coercions between different data representations. This position would not be apparent if subtyping were implicit.

3.5 Shallow Subtyping

The only subtyping rule in λ^{CIL} is on arrow types:

$$(\text{arrow-}\leq)\ \frac{\phi \subseteq \phi';\ \psi' \subseteq \psi}{\sigma \xrightarrow[\psi]{\phi} \tau \leq \sigma \xrightarrow[\psi']{\phi'} \tau}$$

Because this rule is invariant in the argument and result types, it is said to be a *shallow* subtyping rule. In contrast, a *deep* subtyping rule would be contravariant in the argument type and covariant in the result type.

We avoid deep subtyping in λ^{CIL} because we do not know how to formulate it in such a way that it is compatible with our goal of using flow types to guide representation transformations in a strongly typed framework. For example, consider the following types:

$$\begin{aligned} \sigma' \equiv \quad \mathrm{int} \xrightarrow[\{3\}]{\{1\}} \mathrm{int} \quad &\leq \ \mathrm{int} \xrightarrow[\{3\}]{\{1,2\}} \mathrm{int} \ \equiv \sigma \\ \tau \equiv \mathrm{bool} \xrightarrow[\{5,6\}]{\{4\}} \mathrm{bool} &\leq \mathrm{bool} \xrightarrow[\{5\}]{\{4\}} \mathrm{bool} \equiv \tau' \end{aligned}$$

In a language with deep subtyping, the following term would be well typed:

$$\mathbf{coerce}\left(\sigma \xrightarrow[\psi]{\phi} \tau, \sigma' \xrightarrow[\psi]{\phi} \tau'\right) g^{\sigma \xrightarrow[\psi]{\phi} \tau}$$

In many type systems that support subtyping, when a term M has a type σ that is a subtype of τ, we expect that we can transform M to a term M' that can be shown to have type τ *without* using a subtyping rule as the last step of the proof showing that M' has type τ. In this example, we expect that we can "lower the depth" of the subtyping represented by the **coerce** by making a new abstraction that performs coercions on its argument and result:

$$\lambda^{7}_{\psi} f^{\sigma'}.\mathbf{coerce}\,(\tau, \tau')\,(g^{\sigma \xrightarrow[\{8\}]{\phi} \tau}\ @^{\phi}_{8}\ (\mathbf{coerce}\,(\sigma', \sigma)\, f^{\sigma'}))$$

However, this transformation introduces a new abstraction, labelled 7, and a new application site, labelled 8. The new application site consumes all sources in ϕ but does not pass them on to the sinks in ψ. Instead, the set $\{7\}$ takes the place of ϕ. This is problematic because representation decisions made for $\sigma \xrightarrow[\psi]{\phi} \tau$ in the untransformed term may not be valid for either $\sigma \xrightarrow[\{8\}]{\phi} \tau$ or $\sigma' \xrightarrow[\psi]{\{7\}} \tau'$ in the transformed term. For example, if all the abstractions in ϕ were closed (i.e., had no free variables), and these were the only values flowing to the application sites in ψ, it might be assumed that those sites could use a customized calling convention more efficient than the standard closure invocation (Wand & Steckler, 1994; Dimock *et al.*, 1997). But in the above translation, abstraction 7 is open (it contains the free variable g); this thwarts the attempted customization.

3.6 Parallel Reduction

The fact that typed terms in λ^{CIL} are isomorphic to typing derivations makes it impossible to use the ordinary definition of reduction. For example, consider the following untyped terms:

$$\begin{aligned}\hat{M}_e \equiv\ & \textbf{let } g = \lambda y.(\lambda z.y) \mathbin{@} 1 \textbf{ in } \times(g \mathbin{@} 6.001, g \mathbin{@} \textbf{true})\\ \hat{N}_e \equiv\ & \textbf{let } g = \lambda y.y \textbf{ in } \times(g \mathbin{@} 6.001, g \mathbin{@} \textbf{true})\end{aligned}$$

In one call-by-value β-reduction step, $\hat{M}_e$ reduces to $\hat{N}_e$ in $\lambda^{\text{CIL}}_{\text{ut}}$. Now consider a typed term M_e whose type erasure is $\hat{M}_e$:

$$\begin{aligned}M_e \equiv\ & \textbf{let } g^{\wedge[\text{real}\to\text{real},\text{bool}\to\text{bool}]} = \textstyle\bigwedge\big(\lambda y^{\text{real}}.(\lambda z^{\text{int}}.y) \mathbin{@} 1, \lambda y^{\text{bool}}.(\lambda z^{\text{int}}.y) \mathbin{@} 1\big)\\ & \textbf{in } \times((\pi_1^{\wedge}\, g) \mathbin{@} 6.001, (\pi_1^{\wedge}\, g) \mathbin{@} \textbf{true})\end{aligned}$$

It takes *two* call-by-value β-reduction steps to transform M_e into the typed term which corresponds to $\hat{N}_e$:

$$\begin{aligned}N_e \equiv\ & \textbf{let } g^{\wedge[\text{real}\to\text{real},\text{bool}\to\text{bool}]} = \textstyle\bigwedge\big(\lambda y^{\text{real}}.y, \lambda y^{\text{bool}}.y\big)\\ & \textbf{in } \times((\pi_1^{\wedge}\, g) \mathbin{@} 6.001, (\pi_1^{\wedge}\, g) \mathbin{@} \textbf{true})\end{aligned}$$

Furthermore, if these steps are performed sequentially, the intermediate result is ill-typed and corresponds to *no* $\lambda^{\text{CIL}}_{\text{ut}}$ term (because the type erasure of all components of a virtual tuple must be identical).

To solve this problem, λ^{CIL} uses a notion of *parallel context* to force each reduction step at the typed level to correspond to a single reduction step at the untyped level (Kfoury & Wells, 1995). A parallel context is a typed context, possibly containing multiple holes, whose type erasure contains a single hole. All typed terms $M_1, \ldots, M_k$ filling the holes of a parallel context type erase to the same untyped term $\hat{M}$. If $\hat{M}$ is a redex, then a typed reduction step can take place in which each of $M_1, \ldots, M_k$ simultaneously takes a computation step corresponding to the one taken by $\hat{M}$. This notion of reduction allows M_e to reduce to N_e in one call-by-value β step, thereby avoiding the undefined intermediate state noted above.

3.7 Subject Reduction for Union Types and Call-by-Value Reduction

It is difficult to formulate an implicitly typed calculus with union types that has the subject reduction property. For an explicitly typed calculus, this problem manifests itself as a difficulty in guaranteeing the property that any computation that can be performed on an untyped program can be duplicated on a typed version of the same program. For a language with union types, this property does not hold in the presence of general β reduction, but can hold for a call-by-value version of the β rule, where variables are not considered values.

Here we motivate the call-by-value restriction of λ^{CIL} in the context of an example. Consider the following $\lambda^{\text{CIL}}_{\text{ut}}$ term:

$$\hat{M}_1 \equiv (\lambda f.(\pi_1^{\times}\, f) \mathbin{@} (\pi_2^{\times}\, f)) \mathbin{@} (\textbf{if } b \textbf{ then } \times((\lambda e.e+1), 5) \textbf{ else } \times((\lambda e.2), \times()))$$

If $\lambda^{\mathrm{CIL}}_{\mathrm{ut}}$ did not have the value restriction on the β rule, then $\hat{M}_1$ could reduce to

$$\begin{aligned}\hat{M}_2 \equiv \quad & (\pi^{\times}_1\ (\textbf{if}\ b\ \textbf{then}\ \times((\lambda e.e+1),5)\ \textbf{else}\ \times((\lambda e.2),\times())))\\ @\ & (\pi^{\times}_2\ (\textbf{if}\ b\ \textbf{then}\ \times((\lambda e.e+1),5)\ \textbf{else}\ \times((\lambda e.2),\times())))\end{aligned}$$

and, assuming b is **true**, this in turn could reduce to

$$\hat{M}_3 \equiv (\pi^{\times}_1\ \times((\lambda e.e+1),5))\ @\ (\pi^{\times}_2\ (\textbf{if}\ b\ \textbf{then}\ \times((\lambda e.e+1),5)\ \textbf{else}\ \times((\lambda e.2),\times())))$$

However, we shall see that this term is not typable.

We now consider two formulations of a $\vee$-elimination typing rule that we shall compare in the context of the above example. In an implicitly typed calculus, the $\vee$-elimination rule is usually formulated as (Barbanera *et al.*, 1995):

$$\frac{A, x{:}\sigma \vdash \hat{M} : \rho;\ \ A, x{:}\tau \vdash \hat{M} : \rho;\ \ A \vdash \hat{N} : \sigma \vee \tau}{A \vdash \hat{M}[x{:=}\hat{N}] : \rho}\ (\vee\ \text{elim a})$$

This rule is unlike typical typing rules in that the $\hat{M}$ and $\hat{N}$ mentioned in the premises are not immediate subterms of the term mentioned in the conclusion. This implies the need to search for a way to decompose the conclusion term into an appropriate $\hat{M}$ and $\hat{N}$. The term $\hat{M}_1$ can be typed by instantiating ($\vee$ elim a) with

$$\begin{aligned}\hat{M} &\equiv (\pi^{\times}_1\ x)\ @\ (\pi^{\times}_2\ x)\\ \hat{N} &\equiv f\\ \sigma &\equiv \times[\mathrm{int} \to \mathrm{int}, \mathrm{int}]\\ \tau &\equiv \times[\times[\,] \to \mathrm{int}, \times[\,]]\\ \rho &\equiv \mathrm{int}\end{aligned}$$

and $\hat{M}_2$ can be typed using the same except

$$\hat{N} \equiv (\textbf{if}\ b\ \textbf{then}\ \times((\lambda e.e+1),5)\ \textbf{else}\ \times((\lambda e.2),\times())).$$

However, it is impossible to construct a type derivation for $\hat{M}_3$ that uses ($\vee$ elim a). The union type introduced by the **if** subterm cannot be eliminated no matter how the term is decomposed. This type can be eliminated in $\hat{M}_2$ because both copies of the **if** subterm are effectively shared via the substitution for x. But the reduction step from $\hat{M}_2$ to $\hat{M}_3$ reduces only one of these copies, thereby destroying the sharing.

An alternative formulation of the $\vee$-elimination rule is:

$$\frac{A, x{:}\sigma \vdash \hat{M} : \rho;\ \ A, x{:}\tau \vdash \hat{M} : \rho;\ \ A \vdash \hat{N} : \sigma \vee \tau}{A \vdash (\lambda x.\hat{M})\ @\ \hat{N} : \rho}\ (\vee\ \text{elim b})$$

This formulation has the advantage that the $\hat{M}$ and $\hat{N}$ appearing in the premises are immediate subterms of the conclusion term. $\hat{M}_1$ can be typed via ($\vee$ elim b), but this rule cannot be used to type either $\hat{M}_2$ or $\hat{M}_3$ because neither contains the β redex required by the conclusion. Essentially, ($\vee$ elim b) requires the sharing of a term of union type to be explicit in the syntax of the language, and is not applicable to terms like $\hat{M}_2$ where such sharing is implicit.

In λ^{CIL}, we avoid the decomposition problem associated with the ($\vee$ elim a) rule by adopting the ($\vee$ elim b) rule and address the sharing problem by stipulating call-by-value reduction. The rule ($\vee$ elim b) is the $\vee$-elimination rule for the implicitly typed language $\lambda_{\mathrm{i}}^{\mathrm{CIL}}$. The corresponding $\vee$-elimination rule for the explicitly typed language λ^{CIL} involves the $\mathbf{case}^{\vee}$ construct. We address this in sections 4.3 and 4.4. Values in $\lambda_{\mathrm{ut}}^{\mathrm{CIL}}$ are constants and abstractions and the set of values is closed under tuple and variant formation. Values in λ^{CIL} are similar but also include virtual tuples and variants. Requiring the operand of a β redex to be a value guarantees that in the explicitly typed language λ^{CIL} a term of union type cannot be copied via substitution unless it is a union introduction form (i.e., $\left(\mathbf{in}_i^{\vee}\ V\right)^{\vee[\tau_1,\ldots,\tau_n]}$). In the implicitly typed language $\lambda_{\mathrm{i}}^{\mathrm{CIL}}$ (see section 4.4) this implies that when ($\vee$ elim b) is used to type a β-value redex, it occurs in the following pattern:

$$\frac{A, x{:}\sigma \vdash \hat{M} : \rho; \;\; A, x{:}\tau \vdash \hat{M} : \rho; \;\; \dfrac{A \vdash \hat{N} : \sigma}{A \vdash \hat{N} : \sigma \vee \tau}\,(\vee \text{ intro})}{A \vdash (\lambda x.\hat{M}) \mathbin{@} \hat{N} : \rho}\,(\vee \text{ elim b}) \tag{1}$$

But any typing derivation pattern of this form can be replaced by the following pattern, for which subject reduction is straightforward to prove:

$$\frac{\dfrac{A, x{:}\sigma \vdash \hat{M} : \rho}{A \vdash (\lambda x.\hat{M}) : \sigma \to \rho}\,(\to \text{ intro}) \;\; ; \;\; A \vdash \hat{N} : \sigma}{A \vdash (\lambda x.\hat{M}) \mathbin{@} \hat{N} : \rho}\,(\to \text{ elim}) \tag{2}$$

So in $\lambda_{\mathrm{i}}^{\mathrm{CIL}}$, the reduction from $\hat{M}_1$ to $\hat{M}_2$ is illegal. Instead, again assuming b is **true**, the only legal reduction from $\hat{M}_1$ is to

$$\hat{M}_2' \equiv (\lambda f.(\pi_1^{\times}\ f) \mathbin{@} (\pi_2^{\times}\ f)) \mathbin{@} \times((\lambda e.e+1), 5)$$

and thence to

$$\hat{M}_3' \equiv (\pi_1^{\times} \times((\lambda e.e+1), 5)) \mathbin{@} (\pi_2^{\times} \times((\lambda e.e+1), 5))$$

Note that in the explicitly typed language λ^{CIL}, the statement that values of union type must be union introduction forms is only true if variables are not considered to be values. While variables are typically considered values in other call-by-value calculi (e.g., (Plotkin, 1975)), they cause trouble in λ^{CIL} because they can invalidate the equivalence between the two typing derivation patterns (1) and (2) shown above. As a concrete example of this trouble, consider the following terms:

$$\begin{array}{rcl} \hat{M}_{\mathbf{if}} & \equiv & (\mathbf{if}\ b\ \mathbf{then}\ \times((\lambda e.e+1), 5)\ \mathbf{else}\ \times((\lambda e.2), \times())) \\ \hat{M}_4 & \equiv & (\lambda z.z \mathbin{@} \hat{M}_{\mathbf{if}}) \mathbin{@} (\lambda y.(\lambda f.(\pi_1^{\times}\ f) \mathbin{@} (\pi_2^{\times}\ f)) \mathbin{@} y) \\ \hat{M}_5 & \equiv & (\lambda z.z \mathbin{@} \hat{M}_{\mathbf{if}}) \mathbin{@} (\lambda y.(\pi_1^{\times}\ y) \mathbin{@} (\pi_2^{\times}\ y)) \end{array}$$

If variables were considered values, then $\hat{M}_4$ could reduce to $\hat{M}_5$. But whereas $\hat{M}_4$ can be typed using ($\vee$ elim b), $\hat{M}_5$ cannot, because it does not contain the β redex required by the conclusion.

Subject reduction is achieved in both the explicitly and implicitly typed versions of λ^{CIL} by requiring call-by-value reduction and not treating variables as values. This ensures that every reduction at the untyped level will have a corresponding reduction at the typed level. In this way, the implicitly typed language $\lambda_{\mathrm{i}}^{\mathrm{CIL}}$ inherits subject reduction from the explicitly typed language λ^{CIL}.

$\lambda_{\mathrm{i}}^{\mathrm{CIL}}$ appears to be the first implicitly typed calculus with union types that has the subject reduction property for a single call-by-value β-reduction step. The loss of subject reduction in the presence of union types and unrestricted reduction has been noted before. Barbanera, Dezani-Ciancaglini, and de'Liguoro (Barbanera *et al.*, 1995) report the following example (due to Pierce): the term $(\lambda x.\lambda y.\lambda z.x((\lambda t.t)yz)((\lambda t.t)yz))$ can be given the type $((\sigma \to \sigma \to \tau) \wedge (\rho \to \rho \to \tau)) \to (\pi \to (\sigma \vee \rho)) \to \pi \to \tau$, but the term $(\lambda x.\lambda y.\lambda z.x(yz)((\lambda t.t)yz))$ to which it reduces cannot. To regain subject reduction, they adopt a notion of parallel reduction based on complete developments, but this needs to perform multiple β-reduction steps simultaneously at the untyped level.

Inspired in part by an earlier version of the work reported here, Palsberg and Pavlopoulou (Palsberg & Pavlopoulou, 2001) developed a language with union and intersection types that uses the call-by-value restriction to achieve a property that is similar to subject reduction, but weaker. In particular, they show the preservation of types across an evaluation relation rather than the more general reduction relation considered here.

The fact that λ^{CIL} uses call-by-value reduction and does not treat variables as values means that some common local transformations cannot be proven correct within the calculus. For example, consider the following transformations:

$$(\lambda z^{\tau}.\times(z,z)) \mathbin{@} M \text{ is transformed to } \times(M,M) \tag{3}$$

$$\lambda y^{\tau}.((\lambda z^{\tau}.\times(z,z)) \mathbin{@} y) \text{ is transformed to } \lambda y^{\tau}.\times(y,y) \tag{4}$$

Although transformation (3) preserves meaning for any M in the purely functional calculus λ^{CIL}, this can only be proven via the calculus when M is a value. Similarly, transformation (4) is always safe, but it cannot be justified by λ^{CIL} because variables are not values. Proving meaning preservation in these cases requires reasoning outside of the calculus. The sort of limitation illustrated by transformation (3) is exhibited by *any* call-by-value calculus, but the limitation illustrated by transformation (4) is specific to λ^{CIL}. In the typed calculus, the restriction that no variables are values could be loosened to say that no variables *of union type* are values; variables of all other types would be considered values. In this case, transformation (4) would be provable in the calculus if τ were not a union type. We have not taken this approach in this paper because it complicates the relationship between the typed and untyped calculi.

4 Formal Language Definition

The formal definition of the language λ^{CIL} proceeds in several steps:

- Section 4.1 introduces notation and terminology for the formal development.
- Section 4.2 defines the untyped calculus $\lambda_{\mathrm{ut}}^{\mathrm{CIL}}$, which is later used to define reduction and evaluation on the typed calculus. $\lambda_{\mathrm{ut}}^{\mathrm{CIL}}$ is a call-by-value version

of the pure λ-calculus extended with tuples, variants, and recursion. We show that $\lambda^{\mathrm{CIL}}_{\mathrm{ut}}$ is confluent. Work we have reported elsewhere shows standardization for $\lambda^{\mathrm{CIL}}_{\mathrm{ut}}$. These imply that $\lambda^{\mathrm{CIL}}_{\mathrm{ut}}$ is computationally sound: any calculus step is meaning-preserving relative to the operational semantics.

- Section 4.3 defines the explicitly typed language λ^{CIL} using product, intersection, sum, and union types and flow-annotated function types. First, type-annotated and flow-annotated contexts and terms of λ^{CIL} are defined along with a notion of type erasure mapping the annotated terms back into $\lambda^{\mathrm{CIL}}_{\mathrm{ut}}$. Then, reduction and evaluation rules are defined that provide the expected correspondence between λ^{CIL} and $\lambda^{\mathrm{CIL}}_{\mathrm{ut}}$. We also show that λ^{CIL} satisfies a subject reduction property, and use this property to prove type soundness.
- Section 4.4 observes that an implicitly typed language $\lambda^{\mathrm{CIL}}_{\mathrm{i}}$ is automatically obtained by taking typing derivations of an unlabelled version of $\lambda^{\mathrm{CIL}}_{\mathrm{ut}}$ and erasing types from the terms in these derivations.

4.1 General Notation and Terminology

A *context* is a term containing holes, where each hole is denoted by $\Box$. However, in this paper, it is simpler to view *terms* as contexts without holes. The expression $C[M_1, \ldots, M_n]$ denotes the result of placing terms $M_1, \ldots, M_n$ in the n holes of the context C from left to right, possibly capturing free variables. For terms, $M \equiv N$ denotes that M and N are the same term after renaming bound variables. We identify terms up to such renaming. For contexts, $C_1 \equiv C_2$ is similar but only allows renaming bound variables whose scopes do not include a hole. The statement $X \lhd Y$ means that the syntactic entity X occurs properly within the syntactic entity Y; $X \unlhd Y$ has the same meaning except X and Y may be the same. The expression $M[x{:=}N]$ denotes the result of replacing all free occurrences of x in M by N after first renaming the bound variables of M to be distinct from the free variables of N. For types, $\tau[\alpha{:=}\sigma]$ has an analogous meaning. The expression $\mathrm{FV}(X)$ denotes the set of free (unbound) variables of the syntactic entity X, where X is a term or type.

Our presentation generalizes *notions of reduction* (n.o.r.) (Barendregt, 1984). A *simple* n.o.r. R is a pair $(\rightsquigarrow, \mathbf{C})$ of a redex/contractum relation $\rightsquigarrow$ and a set of reduction contexts $\mathbf{C}$.[7] Given $R = (\rightsquigarrow, \mathbf{C})$, the statement $M \rightsquigarrow N$ means that M is an *R-redex* and N is the *R-contractum* of M. Given $R = (\rightsquigarrow, \mathbf{C})$, the statement $M \longrightarrow_R N$ means that M is transformed into N by contracting R-redexes in positions in M specified by an *R-reduction context*, i.e., there is a context $C \in \mathbf{C}$ with k holes and there are terms M_i and N_i for $i \in \{1, \ldots, k\}$ such that $M \equiv C[M_1, \ldots, M_k]$ and $N \equiv C[N_1, \ldots, N_k]$ and $M_i \rightsquigarrow N_i$ for $i \in \{1, \ldots, k\}$. A *composite* n.o.r. R is a rule composing reduction steps of simple n.o.r.'s; in this case $M \longrightarrow_R N$ means M and N are related by the rule (see figure 9 for an example of

[7] Barendregt's definition sec. 3.1.1 is a special case of our definition for simple n.o.r.s which is equivalent to requiring for any n.o.r. $R = (\rightsquigarrow, \mathbf{C})$ that $\mathbf{C}$ is the set of all single-hole contexts. In this case, Barendregt's formulation yields $\longrightarrow_R$ as the *compatible* closure of $\rightsquigarrow$.

a composite n.o.r.). The symbol $\twoheadrightarrow_R$ denotes the transitive and reflexive closure of $\longrightarrow_R$. A term M is in *normal form* with respect to R, written R-nf(M), when there is no term N such that $M \longrightarrow_R N$. The statement $M \xrightarrow{\text{nf}}_R N$ means $M \twoheadrightarrow_R N$ and R-nf(N).

4.2 Untyped Language $\lambda^{\text{CIL}}_{\text{ut}}$

4.2.1 Syntax and Semantics of $\lambda^{\text{CIL}}_{\text{ut}}$

Figure 4 shows the syntax and semantics of the untyped language $\lambda^{\text{CIL}}_{\text{ut}}$. The syntactic categories **UntContext**, **UntTerm**, **UntValue** and **UntEvalContext** are respectively the untyped *contexts*, *terms*, *values* and *evaluation contexts*.

$\lambda^{\text{CIL}}_{\text{ut}}$ includes constants, but no primitive operators on constants. The reason for this is that values at ground type are necessary for some formal statements (including some statements in other papers relying on this one), but the presentation is simpler without primitive operators on these values.

4.2.2 Confluence of $\lambda^{\text{CIL}}_{\text{ut}}$

We will prove confluence of $\lambda^{\text{CIL}}_{\text{ut}}$ by translating it into a *regular combinatory reduction system* (CRS). The notion of a CRS and what it means for a CRS to be regular is defined in appendix A. We define a CRS Σ_{ut} using the following set of function symbols.[8]

$$\begin{aligned}\mathcal{F} = \{&\lambda^{(1)}, @^{(2)}, \mu^{(1)}, \mathbf{val}^{(1)}, \mathbf{notval}^{(1)}\} \cup \mathbf{Constant}\\ &\cup \{\, \times_i^{(i)}, \pi_i^{\times(1)}, \mathbf{in}_i^{+(1)}, \mathbf{case}_i^{+(i+1)} \mid i \in \mathbb{N}\,\}\end{aligned}$$

We define the function $\mathcal{C}_{\text{ut}} : \mathbf{UntTerm} \to \text{Ter}(\mathcal{F})$ together with an auxiliary function $\mathcal{B}_{\text{ut}} : \mathbf{UntTerm} \to \text{Ter}(\mathcal{F})$ to translate untyped terms into CRS terms:

$$\begin{aligned}
\mathcal{C}_{\text{ut}}(\hat{M}) &= \begin{cases}\mathbf{val}(\mathcal{B}_{\text{ut}}(\hat{M})) & \text{if } \hat{M} \in \mathbf{UntValue},\\ \mathbf{notval}(\mathcal{B}_{\text{ut}}(\hat{M})) & \text{if } \hat{M} \notin \mathbf{UntValue}.\end{cases}\\
\mathcal{B}_{\text{ut}}(c) &= c\\
\mathcal{B}_{\text{ut}}(x) &= x\\
\mathcal{B}_{\text{ut}}(\mathbf{rec}\ x.\hat{M}) &= \mu([x]\mathcal{C}_{\text{ut}}(\hat{M}))\\
\mathcal{B}_{\text{ut}}(\lambda x.\hat{M}) &= \lambda([x]\mathcal{C}_{\text{ut}}(\hat{M}))\\
\mathcal{B}_{\text{ut}}(\hat{M} \mathbin{@} \hat{N}) &= @(\mathcal{C}_{\text{ut}}(\hat{M}), \mathcal{C}_{\text{ut}}(\hat{N}))\\
\mathcal{B}_{\text{ut}}(\times\Big(\hat{M}_1, \ldots, \hat{M}_n\Big)) &= \times_n(\mathcal{C}_{\text{ut}}(\hat{M}_1), \ldots, \mathcal{C}_{\text{ut}}(\hat{M}_n))\\
\mathcal{B}_{\text{ut}}(\pi_i^{\times}\ \hat{M}) &= \pi_i^{\times}(\mathcal{C}_{\text{ut}}(\hat{M}))\\
\mathcal{B}_{\text{ut}}(\mathbf{in}_i^{+}\ \hat{M}) &= \mathbf{in}_i^{+}(\mathcal{C}_{\text{ut}}(\hat{M}))\\
\mathcal{B}_{\text{ut}}(\mathbf{case}^{+}\ \hat{M}\ \mathbf{bind}\ x\ \mathbf{in}\ \hat{M}_1, \ldots, \hat{M}_n) &= \mathbf{case}_n^{+}(\mathcal{C}_{\text{ut}}(\hat{M}), [x]\mathcal{C}_{\text{ut}}(\hat{M}_1), \ldots, [x]\mathcal{C}_{\text{ut}}(\hat{M}_n))
\end{aligned}$$

Now we give reduction rules for the CRS to simulate reduction in $\lambda^{\text{CIL}}_{\text{ut}}$. The key technical challenge here is to specify the value restriction of the application, projection, and case analysis reduction rules using only non-ambiguous CRS reduction

[8] All members of **Constant** are assumed to have arity 0.

Untyped Syntax

$$
\begin{array}{rcl}
x, y, z \in \textbf{Variable} & & c \in \textbf{Constant} \\
\hat{C} \in \textbf{UntContext} & ::= & \square \mid c \mid x \mid \textbf{rec}\, x.\hat{C} \mid \lambda x.\hat{C} \mid \hat{C}_1 \mathbin{@} \hat{C}_2 \\
& & \mid \times\left(\hat{C}_1, \ldots, \hat{C}_n\right) \mid \pi_i^{\times}\, \hat{C} \\
& & \mid \textbf{in}_i^{+} \hat{C} \mid \textbf{case}^{+}\, \hat{C}\, \textbf{bind}\, x\, \textbf{in}\, \hat{C}_1, \ldots, \hat{C}_n \\
\hat{M}, \hat{N} \in \textbf{UntTerm} & = & \{\, \hat{C} \mid \square \not\trianglelefteq \hat{C} \,\} \\
\hat{V} \in \textbf{UntValue} & ::= & c \mid \lambda x.\hat{M} \mid \times\left(\hat{V}_1, \ldots, \hat{V}_n\right) \mid \textbf{in}_i^{+} \hat{V}
\end{array}
$$

Untyped Redex/Contractum Relation

$$
\begin{array}{rcll}
(\lambda x.\hat{M}) \mathbin{@} \hat{V} & \rightsquigarrow_{\mathrm{ut}} & \hat{M}[x{:=}\hat{V}] & \\
\pi_i^{\times} \times\left(\hat{V}_1, \ldots, \hat{V}_n\right) & \rightsquigarrow_{\mathrm{ut}} & \hat{V}_i & \text{if } 1 \leq i \leq n \\
\textbf{case}^{+}\, (\textbf{in}_i^{+} \hat{V})\, \textbf{bind}\, x\, \textbf{in}\, \hat{M}_1, \ldots, \hat{M}_n & \rightsquigarrow_{\mathrm{ut}} & (\lambda x.\hat{M}_i) \mathbin{@} \hat{V} & \text{if } 1 \leq i \leq n \\
\textbf{rec}\, x.\hat{M} & \rightsquigarrow_{\mathrm{ut}} & \hat{M}[x{:=}(\textbf{rec}\, x.\hat{M})] &
\end{array}
$$

Untyped Reduction Contexts

$$
\textbf{UntRedContext} \;=\; \{\, \hat{C} \mid \hat{C} \in \textbf{UntContext} \text{ and } \hat{C} \text{ has exactly one hole} \,\}
$$

Untyped Evaluation Contexts

$$
\begin{array}{rcl}
\hat{E} \in \textbf{UntEvalContext} & ::= & \hat{F} \mid \times\left(\hat{V}_1, \ldots, \hat{V}_n, \hat{E}, \hat{M}_1, \ldots, \hat{M}_m\right) \mid \textbf{in}_j^{+} \hat{E} \\
\hat{F} & ::= & \square \mid \hat{F} \mathbin{@} \hat{M} \mid (\lambda x.\hat{M}) \mathbin{@} \hat{E} \\
& & \mid \pi_i^{\times}\, \hat{F} \mid \pi_i^{\times} \times\left(\hat{V}_1, \ldots, \hat{V}_n, \hat{E}, \hat{M}_1, \ldots, \hat{M}_m\right) \\
& & \mid \textbf{case}^{+}\, \hat{F}\, \textbf{bind}\, x\, \textbf{in}\, \hat{M}_1, \ldots, \hat{M}_n \\
& & \mid \textbf{case}^{+}\, (\textbf{in}_j^{+} \hat{E})\, \textbf{bind}\, x\, \textbf{in}\, \hat{M}_1, \ldots, \hat{M}_n
\end{array}
$$

Notions of Reduction

$$
\begin{array}{ll}
\text{Untyped Reduction} & \mathrm{ut} = (\rightsquigarrow_{\mathrm{ut}}, \textbf{UntRedContext}) \\
\text{Evaluation} & \hat{\mathrm{e}} = (\rightsquigarrow_{\mathrm{ut}}, \textbf{UntEvalContext})
\end{array}
$$

Fig. 4. Untyped language $\lambda_{\mathrm{ut}}^{\mathrm{CIL}}$.

rules. We also desire to use simple rule schemas rather than rule schemas with one rule for each possible shape of a value. We will define the set of rules R_v to propagate the value status of a term and the set of rules R_{ut} to simulate the reduction rules of $\lambda_{\mathrm{ut}}^{\mathrm{CIL}}$. Let $n \in \mathbb{N}$.

$$
R_v = \begin{cases}
\textbf{notval}(\lambda(Z)) \to \textbf{val}(\lambda(Z)) \\
\textbf{notval}(c) \to \textbf{val}(c) \\
\textbf{notval}(\times_n(\textbf{val}(Z_1), \ldots, \textbf{val}(Z_n))) \to \textbf{val}(\times_n(\textbf{val}(Z_1), \ldots, \textbf{val}(Z_n))) \\
\textbf{notval}(\textbf{in}_i^{+}(\textbf{val}(Z))) \to \textbf{val}(\textbf{in}_i^{+}(\textbf{val}(Z))) \quad \text{where } i \in \mathbb{N}
\end{cases}
$$

$$
R_{\mathrm{ut}} = \begin{cases}
\textbf{notval}(@(\textbf{val}(\lambda([x]Z(x))), \textbf{val}(Z'))) \to Z(Z') \\
\textbf{notval}(\pi_i^{\times}(\textbf{val}(\times_n(Z_1, \ldots, Z_n)))) \to Z_i \quad \text{where } 1 \leq i \leq n \\
\textbf{notval}(\textbf{case}_n^{+}(\textbf{val}(\textbf{in}_i^{+}(\textbf{val}(Z))), [x]Z_1(x), \ldots, [x]Z_n(x))) \\
\quad \to \textbf{notval}(@(\textbf{val}(\lambda([x]Z_i(x))), \textbf{val}(Z))) \quad \text{where } 1 \leq i \leq n \\
\textbf{notval}(\mu([x]Z(x))) \to Z(\mu([x]Z(x)))
\end{cases}
$$

Let Σ_{ut} be the CRS with function symbols $\mathrm{Fun}(\Sigma_{\mathrm{ut}}) = \mathcal{F}$ and the reduction rules $\mathrm{Red}(\Sigma_{\mathrm{ut}}) = R_v \cup R_{\mathrm{ut}}$.

Remark 4.1. The CRS Σ_{ut} meets the *structure-preserving* criteria of (Bloo & Rose, 1996), since every argument of a RHS metavariable occurs as a subterm of the corresponding LHS. Thus, the techniques of (Bloo & Rose, 1996) can easily give an explicit-substitution version of Σ_{ut}. In turn, from this it is possible to derive an abstract machine implementation. □

Now we prove confluence of Σ_{ut} and use this result to prove confluence of $\lambda_{\mathrm{ut}}^{\mathrm{CIL}}$.

Lemma 4.2. *The CRS Σ_{ut} is regular, i.e., its rules are left-linear and unambiguous. (See the appendix for a definition of these terms.)* □

Proof. Simple checking reveals that the rules are left-linear. To see that the rules are unambiguous, first observe that the root symbol of the LHS of every rule is **notval** and none of the LHS's contain **notval** anywhere else. Thus, if two distinct redexes overlap, the overlap must occur at the root of both redexes. Simple inspection of each rule pair reveals that no such overlaps are possible. □

Corollary 4.3. *Reduction in Σ_{ut} is confluent.* □

Now we need to show a correspondence between reduction in $\lambda_{\mathrm{ut}}^{\mathrm{CIL}}$ and the CRS Σ_{ut}. First, we define a function $\mathcal{E} : \mathrm{Ter}(\mathcal{F}) \to \mathrm{Ter}(\mathcal{F})$ which erases **val** and **notval** from terms:

$$\begin{aligned}
\mathcal{E}(F(u)) &= \begin{cases} F(\mathcal{E}(u)) & \text{if } F \notin \{\mathbf{val}, \mathbf{notval}\}, \\ \mathcal{E}(u) & \text{otherwise.} \end{cases} \\
\mathcal{E}(F(u_1, \dots, u_n)) &= F(\mathcal{E}(u_1), \dots, \mathcal{E}(u_n)) \quad \text{where } n = 0 \text{ or } n > 1 \\
\mathcal{E}([x]u) &= [x]\mathcal{E}(u) \\
\mathcal{E}(x) &= x
\end{aligned}$$

Lemma 4.4. *If $\mathcal{E}(\mathcal{C}_{\mathrm{ut}}(\hat{M}_1)) = \mathcal{E}(\mathcal{C}_{\mathrm{ut}}(\hat{M}_2))$ then $\hat{M}_1 = \hat{M}_2$.* □

Next we define a partial function $\mathcal{C}_{\mathrm{ut}}^{-1} : \mathrm{Ter}(\mathcal{F}) \to \mathbf{UntTerm}$ which contains the inverse of $\mathcal{C}_{\mathrm{ut}}$:

$$\mathcal{C}_{\mathrm{ut}}^{-1}(u) = \begin{cases} \hat{M} & \text{if } \hat{M} \text{ is the unique term s.t. } \mathcal{E}(\mathcal{C}_{\mathrm{ut}}(\hat{M})) = \mathcal{E}(u), \\ \text{undefined} & \text{if no such } \hat{M} \text{ exists.} \end{cases}$$

Lemma 4.5. *If $\mathcal{C}_{\mathrm{ut}}(\hat{M}) \twoheadrightarrow_{\Sigma_{\mathrm{ut}}} u$, then $\mathcal{C}_{\mathrm{ut}}^{-1}(u)$ is defined.* □

Lemma 4.6. *Both of the following statements hold:*

1. *If $\hat{M} \twoheadrightarrow_{\mathrm{ut}} \hat{N}$, then $\mathcal{C}_{\mathrm{ut}}(\hat{M}) \twoheadrightarrow_{\Sigma_{\mathrm{ut}}} \mathcal{C}_{\mathrm{ut}}(\hat{N})$.*
2. *If $\mathcal{C}_{\mathrm{ut}}(\hat{M}) \twoheadrightarrow_{\Sigma_{\mathrm{ut}}} u$, then $\hat{M} \twoheadrightarrow_{\mathrm{ut}} \mathcal{C}_{\mathrm{ut}}^{-1}(u)$.* □

Theorem 4.7 (Confluence of Untyped Reduction). *If $\hat{M} \twoheadrightarrow_{\mathrm{ut}} \hat{N}_1$ and $\hat{M} \twoheadrightarrow_{\mathrm{ut}} \hat{N}_2$, then there exists $\hat{M}'$ such that $\hat{N}_1 \twoheadrightarrow_{\mathrm{ut}} \hat{M}'$ and $\hat{N}_2 \twoheadrightarrow_{\mathrm{ut}} \hat{M}'$.* □

Proof. By constructing this diagram:

$$
\begin{tikzcd}
 & \hat{N}_1 \arrow[r, dashed, bend left] \arrow[drr, "\mathrm{ut}"] & \mathcal{C}_{\mathrm{ut}}(\hat{N}_1) \arrow[l, dotted] \arrow[dr, "\Sigma_{\mathrm{ut}}"] & \\
\hat{M} \arrow[ur, "\mathrm{ut}"] \arrow[dr, "\mathrm{ut}"'] \arrow[r, dashed] & \mathcal{C}_{\mathrm{ut}}(\hat{M}) \arrow[ur, "\Sigma_{\mathrm{ut}}"] \arrow[dr, "\Sigma_{\mathrm{ut}}"'] & \mathcal{C}_{\mathrm{ut}}^{-1}(v) & v \arrow[l, dotted] \\
 & \hat{N}_2 \arrow[r, dashed, bend left] \arrow[urr, "\mathrm{ut}"'] & \mathcal{C}_{\mathrm{ut}}(\hat{N}_2) \arrow[l, dotted] \arrow[ur, "\Sigma_{\mathrm{ut}}"'] &
\end{tikzcd}
$$

□

4.2.3 Standardization for $\lambda^{\mathrm{CIL}}_{\mathrm{ut}}$

The $\lambda^{\mathrm{CIL}}_{\mathrm{ut}}$ calculus has the property that if a term $\hat{M}$ reduces to a value $\hat{V}$, then there exists a value $\hat{V}_0$ and a reduction sequence from $\hat{M}$ to $\hat{V}_0$ in which the reduced redexes occur in the restricted contexts **UntEvalContext**. The proof of this theorem is presented elsewhere.

Theorem 4.8 (Standardization for $\lambda^{\mathrm{CIL}}_{\mathrm{ut}}$). *If $\hat{M} \twoheadrightarrow_{\mathrm{ut}} \hat{V}$, then there exists $\hat{V}_0 \in$* **UntValue** *such that $\hat{M} \twoheadrightarrow_{\hat{\mathrm{e}}} \hat{V}_0 \twoheadrightarrow_{\mathrm{ut}} \hat{V}$.* □

Proof. See (Muller & Wells, 2000). □

4.3 Explicitly Typed Language λ^{CIL}

4.3.1 Type/Flow-Annotated Term Syntax

Figure 5 shows the syntax of the explicitly typed language λ^{CIL}. The syntactic categories **Context**, **Term**, and **Value** are respectively the type and flow-annotated versions of **UntContext**, **UntTerm**, and **UntValue**.

In this presentation, only abstractions, applications, and function types are given flow labels. We could have similarly annotated product, sum, intersection, union, and base types along with the introduction and elimination terms for these types.[9] However, we avoid these additional annotations in order to simplify the presentation. The absence of these additional annotations in no way effects the class of flow analyses that can be encoded in λ^{CIL}.

All sets of flow labels are assumed to be non-empty and finite. The requirement of non-empty label sets is imposed by certain representation transformations in the compiler framework based on λ^{CIL}(Dimock *et al.*, 1997). Even under the whole-program assumption, it is possible to have terms and types without sources or without sinks (due to dead code). To represent this, we use distinguished "no source" and "no sink" labels.

[9] In fact, our implementation of λ^{CIL} includes annotations on all of these types.

Syntax Shared between Types and Terms

$Q ::= P \mid S \qquad S ::= \vee \mid + \qquad P ::= \wedge \mid \times \qquad l, k \in \textbf{Label} = \mathbb{N} \qquad \varnothing \neq \phi, \psi \subset \textbf{Label}$

Types

$$\begin{array}{lll}
o \in \textbf{BaseType} & & \\
\alpha \in \textbf{TypeVariable} & & \\
\xi \in \textbf{OpenType} & ::= & o \mid \upsilon_1 \xrightarrow[\psi]{\phi} \upsilon_2 \mid Q[\upsilon_1, \ldots, \upsilon_n] \mid \mu\alpha.\xi \\
\upsilon & ::= & \alpha \mid \xi \\
\rho, \sigma, \tau \in \textbf{Type} & = & \{\xi \mid \mathrm{FV}(\xi) = \varnothing\}
\end{array}$$

Type Equality

$\sigma = \tau$ iff the infinite unfoldings of σ and τ are identical

Type-Annotated Contexts

$$\begin{array}{lll}
C \in \textbf{Context} & ::= & \square \mid c \mid x^\tau \mid \textbf{rec}\ x^\tau.C \mid \lambda^l_\psi x^\tau.C \mid C_1 \mathbin{@}^\phi_k C_2 \\
& \mid & P(C_1, \ldots, C_n) \mid \pi^P_i\ C \mid \textbf{coerce}\,(\sigma, \tau)\, C \mid \textbf{let}\ x^\tau = C_1\ \textbf{in}\ C_2 \\
& \mid & \left(\textbf{in}^S_i\ C\right)^\tau \mid \textbf{case}^S\ C\ \textbf{bind}\ x\ \textbf{in}\ \tau_1 \Rightarrow C_1, \ldots, \tau_n \Rightarrow C_n
\end{array}$$

Type Erasure (a partial function from **Context** to **UntContext**)

$$\begin{array}{rlrl}
|\square| & \equiv \square & |c| & \equiv c \\
|x^\tau| & \equiv x & |\textbf{rec}\ x^\tau.C| & \equiv \textbf{rec}\ x.|C| \\
\left|\lambda^l_\psi x^\tau.C\right| & \equiv \lambda x.|C| & \left|C_1 \mathbin{@}^\phi_k C_2\right| & \equiv |C_1| \mathbin{@} |C_2| \\
|\times(C_1, \ldots, C_n)| & \equiv \times(|C_1|, \ldots, |C_n|) & |\textbf{coerce}\,(\sigma, \tau)\, C| & \equiv |C| \\
\left|\pi^\times_i\ C\right| & \equiv \pi^\times_i\ |C| & |\pi^\wedge_i\ C| & \equiv |C| \\
\left|\left(\textbf{in}^+_i\ C\right)^\tau\right| & \equiv \textbf{in}^+_i |C| & \left|(\textbf{in}^\vee_i\ C)^\tau\right| & \equiv |C|
\end{array}$$

$$|\textbf{let}\ x^\tau = C_1\ \textbf{in}\ C_2| \equiv (\lambda x.|C_2|) \mathbin{@} |C_1|$$

$$\left|\textbf{case}^+\ C\ \textbf{bind}\ x\ \textbf{in}\ \tau_1 \Rightarrow C_1, \ldots, \tau_n \Rightarrow C_n\right| \equiv \textbf{case}^+\ |C|\ \textbf{bind}\ x\ \textbf{in}\ |C_1|, \ldots, |C_n|$$

$$|\textbf{case}^\vee\ C\ \textbf{bind}\ x\ \textbf{in}\ \tau_1 \Rightarrow C_1, \ldots, \tau_n \Rightarrow C_n| \equiv \begin{cases} (\lambda x.|C_1|) \mathbin{@} |C| & \text{if } |C_1| \equiv \cdots \equiv |C_n|, \\ \text{undefined} & \text{otherwise.} \end{cases}$$

$$|\wedge(C_1, \ldots, C_n)| \equiv \begin{cases} |C_1| & \text{if } |C_1| \equiv \cdots \equiv |C_n|, \\ \text{undefined} & \text{otherwise.} \end{cases}$$

Type-Annotated Terms, Values, Parallel Contexts

$$\begin{array}{rl}
M, N \in & \textbf{Term} = \{\, C \mid \text{the type erasure } |C| \in \textbf{UntTerm} \,\} \\
V \in & \textbf{Value} = \{\, C \mid \text{the type erasure } |C| \in \textbf{UntValue} \,\} \\
Cp \in & \textbf{ParallelContext} = \{\, C \mid \text{the type erasure } |C| \text{ has exactly one hole} \,\}
\end{array}$$

Syntactic Sugar for Examples

$\text{bool} = +[\times[\,], \times[\,]] \qquad \text{true} \equiv \left(\textbf{in}^+_1\ \times(\,)\right)^{\text{bool}} \qquad \text{false} \equiv \left(\textbf{in}^+_2\ \times(\,)\right)^{\text{bool}}$

$(\textbf{if}\ M_1\ \textbf{then}\ M_2\ \textbf{else}\ M_3) \equiv \textbf{case}^+\ M_1\ \textbf{bind}\ x\ \textbf{in}\ \times[\,] \Rightarrow M_2, \times[\,] \Rightarrow M_3 \quad x \text{ fresh}$

Fig. 5. Syntax of explicitly typed language λ^{CIL}.

The recursive binding construct "μ" is used to build recursive types. We use a standard definition of type equality (Amadio & Cardelli, 1993), in which two types are considered equal only if the regular trees that result from unfolding all recursive types (potentially an infinite number of times) within the two types are equal. While it is possible to axiomatize this notion of type equality (Amadio & Cardelli, 1993), there is no benefit to doing so in our calculus; we do not care which particular finite representation is used to represent a given infinite regular tree. Since we do not distinguish between equal recursive types in any context, we do not include any rules for folding or unfolding recursive types. All type variables appearing in a type must be μ-bound; the syntax forbids free type variables from occurring in typing derivations and terms.

The syntax ($\mathbf{coerce}\,(\sigma, \tau)\, M$) explicitly records at the term level the use of the subtyping rule in figure 6 to coerce the type of M from σ to τ. As explained in section 3.5, there is only a single "shallow" subtyping rule that can add labels to the set of source labels and remove labels from the set of sink labels of a "$\rightarrow$"-type. If flow annotations were added to other types (e.g., products, sums, etc.), shallow subtyping would be extended accordingly.

Although **let** could be defined as syntactic sugar for the application of an abstraction, the desugaring would have to invent unnecessary flow labels. Furthermore, an explicit **let** construct makes it easier for a transformation to handle this pattern differently from how it would handle an abstraction within an application (e.g., there is no need to closure convert an applied abstraction).

The type erasure $|C|$ of a type-annotated context C (defined in figure 5) is the corresponding untyped and unlabelled context. The fact that virtual tuples, virtual **case** expressions, and coercions are erased by type erasure underscores the virtual nature of these constructs. Type erasure does not entirely eliminate virtual **case** expressions, but instead leaves behind the application of an abstraction. This is a consequence of the formulation of the union elimination typing rule, as discussed in section 3.7. Some contexts do not have a type erasure, i.e., those containing virtual tuples like $\wedge(C_1, \ldots, C_n)$ or virtual case expressions like

$$\mathbf{case}^{\vee}\, C\ \mathbf{bind}\ x\ \mathbf{in}\ \tau_1 \Rightarrow C_1, \ldots, \tau_1 \Rightarrow C_1$$

where the type erasures of C_1, …, C_n are not identical. In the definition of $|C|$, it is assumed that if any immediate subcontext of C has an undefined type erasure, then the type erasure of C is also undefined.

Lemma 4.9 (Properties of Sub-Contexts).

1. *If $C_1 \trianglelefteq C_2$ and $|C_2|$ is defined, then $|C_1|$ is defined.*
2. *If $C \trianglelefteq Cp$, then either $C \in$ **ParallelContext** or $C \in$ **Term**.*
3. *If $C \trianglelefteq M$, then $C \in$ **Term**.* □

Lemma 4.10 (Contexts Are Injective Functions). *Both typed and untyped one-holed contexts can be seen as one-to-one functions from contexts to contexts (after identifying α-equivalence classes), i.e.,*

1. $\hat{C}[\hat{C}_1] \equiv \hat{C}[\hat{C}_2] \iff \hat{C}_1 \equiv \hat{C}_2$.
2. $C[C_1] \equiv C[C_2] \iff C_1 \equiv C_2$. $\square$

Lemma 4.11 ((De)Composing Parallel Contexts).

1. *Let $C \equiv Cp[C_1, \ldots, C_n]$. If $|C_1| \equiv \cdots \equiv |C_n|$, then $|C|$ is defined if and only if $|C_i|$ is defined, and C is a parallel context if and only if $|C_i|$ has exactly one hole.*
2. *If $|Cp[C_1, \ldots, C_n]|$ is defined, then $|C_1| \equiv \cdots \equiv |C_n|$, and $|Cp[C_1, \ldots, C_n]| \equiv |Cp|\,[|C_1|]$.* $\square$

Lemma 4.12. *If $|C_1| \equiv |C_2|$, then for any one-holed context C, either $|C[C_1]| \equiv |C[C_2]|$ or both $|C[C_1]|$ and $|C[C_2]|$ are undefined.* $\square$

4.3.2 Typing Derivations and Well Typed Terms

$$\text{(const)}\ \frac{\mathrm{ConstType}(c) = o}{A \vdash c : o} \qquad \text{(var)}\ \frac{}{A, x{:}\tau \vdash x^{\tau} : \tau}$$

$$(\rightarrow \text{elim})\ \frac{A \vdash M : \sigma \xrightarrow[\{k\}]{\phi} \tau;\ A \vdash N : \sigma}{A \vdash M \,@^{\phi}_{k}\, N : \tau} \qquad (\rightarrow \text{intro})\ \frac{A, x{:}\sigma \vdash M : \tau}{A \vdash \lambda^{l}_{\psi} x^{\sigma}.M : \sigma \xrightarrow[\psi]{\{l\}} \tau}$$

$$(\times\ \text{intro})\ \frac{\forall_{i=1}^{n}.\ A \vdash M_i : \tau_i}{A \vdash \times(M_1, \ldots, M_n) : \times[\tau_1, \ldots, \tau_n]} \qquad \text{(rec)}\ \frac{A, x{:}\tau \vdash M : \tau}{A \vdash \mathbf{rec}\ x^{\tau}.M : \tau}$$

$$(\wedge\ \text{intro})\ \frac{\forall_{i=1}^{n}.\ A \vdash M_i : \tau_i;\ \ |M_1| \equiv \cdots \equiv |M_n|}{A \vdash \wedge(M_1, \ldots, M_n) : \wedge[\tau_1, \ldots, \tau_n]} \qquad \text{(coerce)}\ \frac{A \vdash M : \sigma;\ \sigma \leq \tau}{A \vdash \mathbf{coerce}\,(\sigma, \tau)\, M : \tau}$$

$$(+,\vee\ \text{intro})\ \frac{A \vdash M : \tau_i;\ 1 \leq i \leq n}{A \vdash \left(\mathbf{in}_i^{S}\ M\right)^{S[\tau_1, \ldots, \tau_n]} : S[\tau_1, \ldots, \tau_n]} \qquad (\rightarrow\leq)\ \frac{\phi \subseteq \phi';\ \psi' \subseteq \psi}{\sigma \xrightarrow[\psi]{\phi} \tau \leq \sigma \xrightarrow[\psi']{\phi'} \tau}$$

$$(\times,\wedge\ \text{elim})\ \frac{A \vdash M : P[\tau_1, \ldots, \tau_n];\ 1 \leq i \leq n}{A \vdash \pi_i^{P}\, M : \tau_i} \qquad \text{(let)}\ \frac{A, x{:}\sigma \vdash N : \tau;\ A \vdash M : \sigma}{A \vdash \mathbf{let}\ x^{\sigma} = M\ \mathbf{in}\ N : \tau}$$

$$(+\ \text{elim})\qquad \frac{A \vdash M : +[\tau_1, \ldots, \tau_n];\ \forall_{i=1}^{n}.\ A, x{:}\tau_i \vdash M_i : \tau}{A \vdash \mathbf{case}^{+}\ M\ \mathbf{bind}\ x\ \mathbf{in}\ \tau_1 \Rightarrow M_1, \ldots, \tau_n \Rightarrow M_n : \tau}$$

$$(\vee\ \text{elim})\ \frac{A \vdash M : \vee[\tau_1, \ldots, \tau_n];\ \forall_{i=1}^{n}.\ A, x{:}\tau_i \vdash M_i : \tau;\ \ |M_1| \equiv \cdots \equiv |M_n|}{A \vdash \mathbf{case}^{\vee}\ M\ \mathbf{bind}\ x\ \mathbf{in}\ \tau_1 \Rightarrow M_1, \ldots, \tau_n \Rightarrow M_n : \tau}$$

Fig. 6. Typing rules of explicitly typed language λ^{CIL}.

Figure 6 gives the typing rules of λ^{CIL}. The function ConstType assigns a base type to each constant. A *type environment* is a finite mapping from term variables to types, i.e., a set of variable/type pairs. If A is a type environment, then $A, x{:}\tau$ denotes A extended to map x to type τ. The domain of definition of A is $\mathrm{DomDef}(A)$. A triple $A \vdash M : \tau$ is a *judgement*.

A *derivation* $\mathcal{D}$ in language X is a tree in which each node N contains a judgement that follows by an instantiation of a typing rule whose conclusion is the judgement of N and whose hypotheses are the judgements of the children of N. To guarantee uniqueness of derivations (see theorem 4.13), the children of a node must be ordered, and their order must match the order of hypotheses in the typing rule. A derivation is said to *end with* a judgement if that judgement is the root of the derivation tree. We write "$A \vdash_X M : \tau$ via $\mathcal{D}$" to mean that derivation $\mathcal{D}$ is valid in language X and $\mathcal{D}$ ends with $A \vdash M : \tau$. In this case, $\mathcal{D}$ is a *typing* for M in X and M is *well typed* in X. The statement $A \vdash_X M : \tau$ means there exists some $\mathcal{D}$ such that $A \vdash_X M : \tau$ via $\mathcal{D}$.

The ($\wedge$ intro) rule requires the equivalence of the type erasure of all components of the virtual tuple, while the ($\vee$ elim) rule requires the equivalence of the type erasures of all clause bodies of a $\mathbf{case}^{\vee}$ expression. These two rules formalize the restrictions on virtual tuples and virtual variants mentioned earlier. The ($\times$,$\wedge$ elim) (resp. (+,$\vee$ intro)) rule works for both product and intersection (resp. sum and union) types, since P (resp. S) ranges over $\times$ and $\wedge$ (resp. + and $\vee$).

Addition and Subtraction of Type Environments

$$A \oplus B = \begin{cases} A \cup B & \text{if } A \cup B \text{ is a function,} \\ \text{undefined} & \text{otherwise.} \end{cases}$$

$$A \ominus B = \begin{cases} A - B & \text{if } A - B \text{ is a function,} \\ \text{undefined} & \text{otherwise.} \end{cases}$$

Environment Inference Function

$$\mathrm{Env} : \mathbf{Term} \hookrightarrow \mathrm{TypeEnvironment}$$

$$\begin{array}{rcl}
\mathrm{Env}(c) & = & \varnothing \\
\mathrm{Env}(x^{\tau}) & = & \{x : \tau\} \\
\mathrm{Env}(\lambda^{l}_{\psi} x^{\tau}.M) & = & \mathrm{Env}(M) \ominus \{x : \tau\} \\
\mathrm{Env}(M \mathbin{@}^{\phi}_{k} N) & = & \mathrm{Env}(M) \oplus \mathrm{Env}(N) \\
\mathrm{Env}(\mathbf{coerce}\,(\sigma, \tau)\, M) & = & \mathrm{Env}(M) \\
\mathrm{Env}(\mathbf{rec}\ x^{\tau}.M) & = & \mathrm{Env}(M) \ominus \{x : \tau\} \\
\mathrm{Env}(\mathbf{let}\ x^{\tau} = M\ \mathbf{in}\ N) & = & \mathrm{Env}(M) \oplus (\mathrm{Env}(N) \ominus \{x : \tau\}) \\
\mathrm{Env}(P(M_1, \ldots, M_n)) & = & \mathrm{Env}(M_1) \oplus \cdots \oplus \mathrm{Env}(M_n) \\
\mathrm{Env}(\pi^{P}_{i}\, M) & = & \mathrm{Env}(M) \\
\mathrm{Env}((\mathbf{in}^{S}_{i}\, M)^{\tau}) & = & \mathrm{Env}(M)
\end{array}$$

$$\mathrm{Env}(\mathbf{case}^{S}\, M\ \mathbf{bind}\ x\ \mathbf{in}\ \tau_1 \Rightarrow M_1, \ldots, \tau_n \Rightarrow M_n) = \\ \mathrm{Env}(M) \oplus (\mathrm{Env}(M_1) \ominus \{x : \tau_1\}) \oplus \ldots \oplus (\mathrm{Env}(M_n) \ominus \{x : \tau_n\})$$

Fig. 7. Definition of the Env function.

We will show that typing derivations and well typed terms are isomorphic, using the functions defined in figures 7 and 8. Env is a partial function that maps a λ^{CIL} term to a type environment pairing each free variable of the term to its type. Env

Type Inference Function

$\mathrm{Typ} : \mathbf{Term} \hookrightarrow \mathbf{Type}$

$$
\begin{array}{rll}
\mathrm{Typ}(c) &= \mathrm{ConstType}(c) & \\
\mathrm{Typ}(x^{\tau}) &= \tau & \\
\mathrm{Typ}(\lambda^{l}_{\psi} x^{\tau}.M) &= \tau \xrightarrow[\psi]{\{l\}} \mathrm{Typ}(M) & \\
\mathrm{Typ}(M \mathbin{@}^{\phi}_{k} N) &= \tau & \text{if } \mathrm{Typ}(M) = \mathrm{Typ}(N) \xrightarrow[\{k\}]{\phi} \tau \\
\mathrm{Typ}(\mathbf{coerce}\,(\sigma,\tau)\,M) &= \tau & \text{if } \mathrm{Typ}(M) = \sigma = \rho_1 \xrightarrow[\psi\cup\psi']{\phi} \rho_2 \\
 & & \text{and } \tau = \rho_1 \xrightarrow[\psi]{\phi\cup\phi'} \rho_2 \\
\mathrm{Typ}(\mathbf{rec}\ x^{\tau}.M) &= \tau & \text{if } \mathrm{Typ}(M) = \tau \\
\mathrm{Typ}(\mathbf{let}\ x^{\tau} = M\ \mathbf{in}\ N) &= \mathrm{Typ}(N) & \text{if } \mathrm{Typ}(M) = \tau \\
\mathrm{Typ}(\times(M_1,\ldots,M_n)) &= \times[\mathrm{Typ}(M_1),\ldots,\mathrm{Typ}(M_n)] & \\
\mathrm{Typ}(\wedge(M_1,\ldots,M_n)) &= \wedge[\mathrm{Typ}(M_1),\ldots,\mathrm{Typ}(M_n)] & \\
\mathrm{Typ}(\pi^{P}_{i}\ M) &= \tau_i & \text{if } \mathrm{Typ}(M) = P[\tau_1,\ldots,\tau_n] \\
 & & \text{and } 1 \le i \le n \\
\mathrm{Typ}((\mathbf{in}^{S}_{i}\ M)^{\tau}) &= \tau & \text{if } \tau = S[\tau_1,\ldots,\tau_n] \\
 & & 1 \le i \le n \text{ and } \mathrm{Typ}(M) = \tau_i \\
\mathrm{Typ}(\mathbf{case}^{+}\ M\ \mathbf{bind}\ x\ \mathbf{in}\ \tau_1 \Rightarrow M_1,\ldots,\tau_n \Rightarrow M_n) & & \\
 &= \mathrm{Typ}(M_1) & \text{if } \mathrm{Typ}(M) = +[\tau_1,\ldots,\tau_n] \\
 & & \text{and } \mathrm{Typ}(M_1) = \cdots = \mathrm{Typ}(M_n) \\
\mathrm{Typ}(\mathbf{case}^{\vee}\ M\ \mathbf{bind}\ x\ \mathbf{in}\ \tau_1 \Rightarrow M_1,\ldots,\tau_n \Rightarrow M_n) & & \\
 &= \mathrm{Typ}(M_1) & \text{if } \mathrm{Typ}(M) = \vee[\tau_1,\ldots,\tau_n] \\
 & & \text{and } \mathrm{Typ}(M_1) = \cdots = \mathrm{Typ}(M_n)
\end{array}
$$

Fig. 8. Definition of the Typ function.

is undefined if there are conflicting type assignments for some free variable within the term. The partial function Typ constructs the type of a λ^{CIL} term based on the explicit type information in the term. In the definition of Typ, if the value of $\mathrm{Typ}(M)$ is not explicitly specified, then it is undefined.

Theorem 4.13 (Uniqueness of Typings in λ^{CIL}).

1. *Every typing derivation for M ends with* $\mathrm{Env}(M) \oplus A \vdash_{\lambda^{\mathrm{CIL}}} M : \mathrm{Typ}(M)$ *for some A.*
2. *If* $\mathrm{Env}(M) \oplus A$ *and* $\mathrm{Typ}(M)$ *are defined, then there is a unique typing derivation $\mathcal{D}$ such that* $\mathrm{Env}(M) \oplus A \vdash_{\lambda^{\mathrm{CIL}}} M : \mathrm{Typ}(M)$ *via $\mathcal{D}$.* □

Proof. By induction on typing derivations. The important thing to observe is that together the functions Env and Typ encode all of the restrictions of the type system, so if M is not typable then either $\mathrm{Env}(M)$ or $\mathrm{Typ}(M)$ will be undefined. □

Thus, when desired, we may recover the type of any well typed term from the term itself. The notation M^{τ} asserts that M is well typed and $\mathrm{Typ}(M) = \tau$.

4.3.3 Reduction on Explicitly Typed Terms

The call-by-value reduction rules for the typed language λ^{CIL} are in figure 9. The main notion of reduction, r-reduction, is divided into three steps: simplifying type

Main Notion of Reduction for Type-Annotated Terms

$$M \longrightarrow_{\mathrm{r}} N \quad \text{iff} \quad \exists M', N'.\ (M \xrightarrow{\mathrm{nf}}_{\mathrm{t}} M' \longrightarrow_{\mathrm{c}} N' \xrightarrow{\mathrm{nf}}_{\mathrm{t}} N)$$

Computation Reduction $(\rightsquigarrow_{\mathrm{c}}, \mathbf{C}_{\mathrm{c}})$

$\mathbf{let}\ x^{\tau} = V\ \mathbf{in}\ M$	$\rightsquigarrow_{\mathrm{c}} M[x{:=}V]$	
$\pi_i^{\times} \times(V_1, \ldots, V_n)$	$\rightsquigarrow_{\mathrm{c}} V_i$	if $1 \le i \le n$
$\mathbf{case}^{+} \left(\mathbf{in}_i^{+}\ V\right)^{\tau}\ \mathbf{bind}\ x\ \mathbf{in}\ \tau_1 \Rightarrow M_1, \ldots, \tau_n \Rightarrow M_n$	$\rightsquigarrow_{\mathrm{c}} \mathbf{let}\ x^{\tau_i} = V\ \mathbf{in}\ M_i$	if $1 \le i \le n$
$\mathbf{rec}\ x^{\tau}.M$	$\rightsquigarrow_{\mathrm{c}} M[x{:=}(\mathbf{rec}\ x^{\tau}.M)]$	

Reduction contexts: $\mathbf{C}_{\mathrm{c}} = \mathbf{ParallelContext}$

Type-Annotation-Simplification Reduction $(\rightsquigarrow_{\mathrm{t}}, \mathbf{C}_{\mathrm{t}})$

$(\lambda_{\psi}^{l} x^{\tau}.N)\ @_k^{\phi}\ M$	$\rightsquigarrow_{\mathrm{t}} \mathbf{let}\ x^{\tau} = M\ \mathbf{in}\ N$	
$\pi_i^{\wedge} \wedge(M_1, \ldots, M_n)$	$\rightsquigarrow_{\mathrm{t}} M_i$	if $1 \le i \le n$
$\mathbf{case}^{\vee}\ (\mathbf{in}_i^{\vee}\ N)^{\tau}\ \mathbf{bind}\ x\ \mathbf{in}\ \tau_1 \Rightarrow M_1, \ldots, \tau_n \Rightarrow M_n$	$\rightsquigarrow_{\mathrm{t}} \mathbf{let}\ x^{\tau_i} = N\ \mathbf{in}\ M_i$	if $1 \le i \le n$
$(\mathbf{coerce}\,(\sigma, \tau)\,(\lambda_{\psi}^{l} x^{\rho}.M))\ @_k^{\phi}\ N$	$\rightsquigarrow_{\mathrm{t}} \mathbf{let}\ x^{\rho} = N\ \mathbf{in}\ M$	
$\mathbf{coerce}\,(\sigma_1, \tau)\ \mathbf{coerce}\,(\rho, \sigma_2)\, M$	$\rightsquigarrow_{\mathrm{t}} \mathbf{coerce}\,(\rho, \tau)\, M$	

Reduction contexts: $\mathbf{C}_{\mathrm{t}} = \{\, C \mid C \in \mathbf{Context}$ and C has exactly one hole $\}$

Typed Evaluation

$$M \longrightarrow_{\mathrm{e}} N \quad \text{iff} \quad M \longrightarrow_{\mathrm{r}} N \text{ and } |M| \longrightarrow_{\hat{\mathrm{e}}} |N|$$

Fig. 9. Reduction rules of explicitly typed language λ^{CIL}.

annotations, performing a computation step, and then simplifying type annotations again. Type annotations that might block a computation step are removed by t-reduction. Since t-reduction is terminating (lemma 4.15), it is convenient to go to t-normal form before and after computation steps. We assume terms are always kept in t-normal form. The notion of c-reduction performs real computation steps. In our term formulation, *parallel* c-redexes (i.e., different type-annotated versions of the *same* program phrase) must be contracted simultaneously. This is formalized using *parallel contexts* (members of **ParallelContext**), which require parallel c-redexes to fill holes that map to the same hole in the type-erased program.

Remark 4.14. In the t-reduction rules in figure 9, constraints that might be expected on the type and flow annotations are not imposed by the reduction rules but are instead a consequence of the typing rules in figure 6. For example, in the application rule, the typing rules imply that ϕ must be $\{l\}$ and ψ must be $\{k\}$. Similar constraints hold for the other t-reduction rules. □

Lemma 4.15. t*-reduction is terminating.* □

Proof. t-reduction reduces the size of a term, where the size is measured as the number of symbols appearing in the term. □

Remark 4.16. We have proven that t-reduction is confluent but do not present this fact, because it is not required for our subject reduction and confluence results for $\longrightarrow_{\mathrm{r}}$. These results only require that any term can be reduced to t-normal-form. □

Lemma 4.17.

1. *If* $M \leadsto_{\mathrm{c}} C$*, then* $C \in \mathbf{Term}$ *and* $|M| \leadsto_{\mathrm{ut}} |C|$*.*
2. *If* $M \leadsto_{\mathrm{t}} C$*, then* $C \in \mathbf{Term}$ *and* $|M| \equiv |C|$*.* □

Proof. By inspection of the reduction rules together with the type erasure rules. □

Lemma 4.18 (Redex/Contractum Relations Are Functions). *For each simple n.o.r.* $R \in \{\mathrm{ut}, \mathrm{c}, \mathrm{t}\}$*, for any syntactic entity* X*, there is at most one* Y *such that* $X \leadsto_R Y$*.* □

Proof. By inspection of the reduction rules. □

Lemma 4.19. *The set* **Term** *is closed under* c*-reduction and* t*-reduction. Also, each* c*-reduction step corresponds to a* ut*-reduction step on the type erasure while each* t*-reduction step preserves the type erasure. More specifically,*

1. *If* $M \longrightarrow_{\mathrm{c}} C$*, then* $C \in \mathbf{Term}$ *and* $|M| \longrightarrow_{\mathrm{ut}} |C|$*.*
2. *If* $M \longrightarrow_{\mathrm{t}} C$*, then* $C \in \mathbf{Term}$ *and* $|M| \equiv |C|$*.* □

Proof.

1. By the definition of c-reduction, we know that $M \equiv Cp[M_1, \ldots, M_n]$ and $C \equiv Cp[C_1, \ldots, C_n]$ where $M_i \leadsto_{\mathrm{c}} C_i$ for $1 \leq i \leq n$. By lemma 4.11, we know that $|M_1| \equiv \cdots \equiv |M_n|$ and $|M| \equiv |Cp|\,[|M_1|]$. By lemma 4.17, we know that $C_1, \ldots, C_n \in \mathbf{Term}$ and $|M_i| \leadsto_{\mathrm{ut}} |C_i|$ for $1 \leq i \leq n$. By lemma 4.18 we know that $|C_1| \equiv \cdots \equiv |C_n|$. By lemma 4.11, we know that $|C|$ is defined, implying $C \in \mathbf{Term}$, and that $|C| \equiv |Cp|\,[|C_1|]$, implying that $|M| \longrightarrow_{\mathrm{ut}} |C|$.
2. By definition of t-reduction, we know that $M \equiv C'[M']$ and $C \equiv C'[C'']$ where $M' \leadsto_{\mathrm{t}} C''$. By lemma 4.17, we know that $C'' \in \mathbf{Term}$ and $|M'| \equiv |C''|$. By lemma 4.12, we know that $|M| \equiv |C'[M']| \equiv |C'[C'']| \equiv |C|$ implying that $C \in \mathbf{Term}$. □

Lemma 4.20. *If* $\mathrm{Env}(M)$ *is defined and if* $x \in \mathrm{DomDef}(\mathrm{Env}(M))$ *implies* $(\mathrm{Env}(M))(x) = \tau$*, then*

1. $\mathrm{Env}(M[x{:=}N]) \subseteq (\mathrm{Env}(M) \ominus \{x : \tau\}) \oplus \mathrm{Env}(N)$*.*
2. *If* $\mathrm{Typ}(N) = \tau$ *and* $\mathrm{Typ}(M)$ *is defined, then* $\mathrm{Typ}(M[x{:=}N]) = \mathrm{Typ}(M)$*.* □

Proof. Both parts are by induction on the structure of M. □

Lemma 4.21. *If* $M \leadsto_R N$ *for* $R \in \{\mathrm{c}, \mathrm{t}\}$*, then*

1. *If* $\mathrm{Env}(M)$ *is defined, then* $\mathrm{Env}(N) \subseteq \mathrm{Env}(M)$*.*
2. *If* $\mathrm{Typ}(M)$ *is defined, then* $\mathrm{Typ}(N) = \mathrm{Typ}(M)$*.*
3. *If* $A \vdash_{\lambda^{\mathrm{CIL}}} M : \tau$*, then* $A \vdash_{\lambda^{\mathrm{CIL}}} N : \tau$*.* □

Proof. For 1 and 2, by cases on the reduction rule, using lemma 4.20 for the reduction of **let**. For 3, using 1 and 2 together with theorem 4.13. □

Lemma 4.22 (Subject c/t-Reduction).

1. *If $M \longrightarrow_{\mathrm{c}} N$ and $A \vdash_{\lambda^{\mathrm{CIL}}} M : \tau$, then $A \vdash_{\lambda^{\mathrm{CIL}}} N : \tau$.*
2. *If $M \longrightarrow_{\mathrm{t}} N$ and $A \vdash_{\lambda^{\mathrm{CIL}}} M : \tau$, then $A \vdash_{\lambda^{\mathrm{CIL}}} N : \tau$.* □

Proof.

1. We know that $M \equiv Cp[M_1, \dots, M_n]$ and $N \equiv Cp[N_1, \dots, N_n]$ where $M_i \rightsquigarrow_{\mathrm{c}} N_i$ for $1 \le i \le n$. Consider the typing derivation $\mathcal{D}$ which proves $A \vdash_{\lambda^{\mathrm{CIL}}} M : \tau$. For $1 \le i \le n$ the typing derivation $\mathcal{D}$ has subderivation $\mathcal{D}_i$ proving $A_i \vdash_{\lambda^{\mathrm{CIL}}} M_i : \tau_i$ for some A_i and τ_i. By lemma 4.21, for $1 \le i \le n$ there is a derivation $\mathcal{D}'_i$ proving $A_i \vdash_{\lambda^{\mathrm{CIL}}} N_i : \tau_i$. Consider the derivation $\mathcal{D}'$ formed from $\mathcal{D}$ by replacing $\mathcal{D}_i$ by $\mathcal{D}'_i$ for $1 \le i \le n$. The only typing rules which inspect the internal structure of the terms in the judgements in their premises are ($\wedge$ intro) and ($\vee$ elim), which merely verify that the type erasure of the term they are building is defined. Because $|N|$ is defined (since it is a term by lemma 4.19), we know that $\mathcal{D}'$ is a valid derivation, giving the desired result.
2. Similar reasoning to the previous case, only simpler. □

Theorem 4.23 (Subject r-Reduction for λ^{CIL}). *If $M \longrightarrow_{\mathrm{r}} N$ and $A \vdash_{\lambda^{\mathrm{CIL}}} M : \tau$, then $A \vdash_{\lambda^{\mathrm{CIL}}} N : \tau$.* □

Proof. The claim follows immediately from lemma 4.22 and the definition of r-reduction as the composition of c-reduction and t-reduction. □

Definition 4.24 (Erasable Form). A term M is an *erasable form* if it has the form $\wedge(M_1, \dots, M_n)$, $\pi_i^{\wedge} M'$, $\mathbf{coerce}\,(\sigma, \tau)\, M'$, or $\left(\mathbf{in}_i^{\vee}\, M'\right)^{\tau}$. □

Lemma 4.25. *If $\text{t-nf}(M)$ and $|M| \equiv \hat{C}[\hat{N}]$, then $M \equiv Cp[N_1, \dots, N_n]$ where $|Cp| \equiv \hat{C}$, $|N_1| \equiv \hat{N}$, and N_i is not an erasable form for $1 \le i \le n$.* □

Lemma 4.26. *If M is well typed, $\text{t-nf}(M)$, M is not an erasable form, and $|M|$ is a ut-redex, then M is a c-redex.* □

Proof. By case analysis on the form of M. □

Lemma 4.27. *If $\text{t-nf}(M)$, M is well typed, and $|M| \longrightarrow_{\mathrm{ut}} \hat{N}$, then there is a term N such that $M \longrightarrow_{\mathrm{c}} N$ and $|N| \equiv \hat{N}$.* □

Proof. Because $|M| \longrightarrow_{\mathrm{ut}} \hat{N}$, we know that $|M| \equiv \hat{C}[\hat{M}']$ and $\hat{N} \equiv \hat{C}[\hat{N}']$ where $\hat{M}' \rightsquigarrow_{\mathrm{ut}} \hat{N}'$. By lemmas 4.25 and 4.26, we know that $M \equiv Cp[M_1, \dots, M_n]$ where $|Cp| \equiv \hat{C}$, $|M_1| \equiv \cdots \equiv |M_n| \equiv \hat{M}'$, and $M_i \rightsquigarrow_{\mathrm{c}} N_i$ for $1 \le i \le n$. Thus, $M \longrightarrow_{\mathrm{c}} N$ where $N \equiv Cp[N_1, \dots, N_n]$. All that remains is to show that $|N| \equiv \hat{N}$.

By lemma 4.19, $|M| \longrightarrow_{\mathrm{ut}} |N|$. By lemma 4.11 and the above reasoning we know that $\hat{C}[\hat{M}'] \longrightarrow_{\mathrm{ut}} \hat{C}[|N_1|]$. Thus, $\hat{M}' \rightsquigarrow_{\mathrm{ut}} |N_1|$. By lemma 4.18, we know that $\hat{N}' \equiv |N_1|$. Thus, $|N| \equiv \hat{C}[|N_1|] \equiv \hat{C}[\hat{N}'] \equiv \hat{N}$, which is exactly the desired result. □

Theorem 4.28 (Typed/Untyped Reduction Correspondence).

1. *If* $M \longrightarrow_{\mathrm{r}} N$*, then* $|M| \longrightarrow_{\mathrm{ut}} |N|$*.*
2. *If* $|M| \longrightarrow_{\mathrm{ut}} \hat{N}$ *and* M *is well typed, then there exists a term* N *where* $M \longrightarrow_{\mathrm{r}} N$ *and* $|N| \equiv \hat{N}$*.* □

Proof.

1. This claim follows immediately from lemma 4.19 and the definition of r-reduction as the composition of c-reduction and t-reduction.
2. By lemma 4.15, $M \xrightarrow{\mathrm{nf}}_{\mathrm{t}} M'$. By lemma 4.22, M' is well typed. By lemma 4.19, $|M'| \equiv |M|$, implying that $|M'| \longrightarrow_{\mathrm{ut}} \hat{N}$. By lemma 4.27, $M' \longrightarrow_{\mathrm{c}} N'$ where $|N'| \equiv \hat{N}$. By lemmas 4.15 and 4.19, $N' \xrightarrow{\mathrm{nf}}_{\mathrm{t}} N$ where $|N'| \equiv |N| \equiv \hat{N}$. By definition of r-reduction, $M \longrightarrow_{\mathrm{r}} N$, showing the desired result. □

Theorem 4.29 (Confluence Modulo Type Erasure of Typed Reduction). *If* M_1 *and* M_2 *are well typed,* $|M_1| \equiv |M_2|$*,* $M_1 \twoheadrightarrow_{\mathrm{r}} N_1$*, and* $M_2 \twoheadrightarrow_{\mathrm{r}} N_2$*, then there exist* M_1' *and* M_2' *such that* $|M_1'| \equiv |M_2'|$*,* $N_1 \twoheadrightarrow_{\mathrm{r}} M_1'$ *and* $N_2 \twoheadrightarrow_{\mathrm{r}} M_2'$*.* □

Proof. By theorem 4.28, $|M_1| \twoheadrightarrow_{\mathrm{ut}} |N_1|$ and $|M_2| \twoheadrightarrow_{\mathrm{ut}} |N_2|$. By theorem 4.7, there exists $\hat{N}$ such that $|N_1| \twoheadrightarrow_{\mathrm{ut}} \hat{N}$ and $|N_2| \twoheadrightarrow_{\mathrm{ut}} \hat{N}$. By theorem 4.28, there exist terms M_1 and M_2 such that $|M_1| \equiv |M_2| \equiv \hat{N}$ and $N_1 \twoheadrightarrow_{\mathrm{r}} M_1$ and $N_2 \twoheadrightarrow_{\mathrm{r}} M_2$. □

Remark 4.30. Confluence modulo type erasure is not as strong a result as traditional confluence, in which $M_1 \equiv M_2$ and $M_1' \equiv M_2'$. However, since meaning in λ^{CIL} is entirely determined at the untyped level, confluence modulo type erasure is sufficient for the purpose of showing that transformations preserve meaning. We conjecture that λ^{CIL} is confluent in the traditional sense, but have not proven this fact. □

Lemma 4.31 (Value Characterization).

1. *If* M *closed, well-typed, and an evaluation normal form, then* M *is a value.*
2. *If* M *is a value, then it is an evaluation normal form.* □

Lemma 4.32 (Progress). *If* $\emptyset \vdash_{\lambda^{\mathrm{CIL}}} M : \tau$*, then either* M *is a value or there exists an* N *such that* $M \longrightarrow_{\mathrm{e}} N$*.* □

Definition 4.33 (Stuck Terms). $M \in$ **Term** is *stuck* iff it is an evaluation normal form that is not a value. □

Theorem 4.34 (Typing Soundness). *If* M *is a well-typed closed term in* λ^{CIL}*, then evaluating it "cannot go wrong". I.e., for all* N *such that* $M \twoheadrightarrow_{\mathrm{e}} N$*,* N *is not stuck.* □

Proof. By assumption, there is a τ such that $\emptyset \vdash_{\lambda^{\mathrm{CIL}}} M : \tau$. The proof is by induction on the length n of the reduction $M \twoheadrightarrow_{\mathrm{e}} N$. If $n = 0$, then by lemma 4.32, N is not stuck because it is either a value or can be evaluated. If $n > 0$, then there is an N' such that $M \longrightarrow_{\mathrm{e}} N' \twoheadrightarrow_{\mathrm{e}} N$. Since $\longrightarrow_{\mathrm{e}}$ is a subrelation of $\longrightarrow_{\mathrm{r}}$, subject reduction of $\longrightarrow_{\mathrm{r}}$ implies $\emptyset \vdash_{\lambda^{\mathrm{CIL}}} N' : \tau$, which is assumed true by the induction hypothesis. □

4.4 Implicitly Typed Language $\lambda_{\mathrm{i}}^{\mathrm{CIL}}$

The implicitly typed language $\lambda_{\mathrm{i}}^{\mathrm{CIL}}$ is obtained from $\lambda_{\mathrm{ut}}^{\mathrm{CIL}}$ and λ^{CIL}. The syntax and semantics of implicitly typed language $\lambda_{\mathrm{i}}^{\mathrm{CIL}}$ are the same as $\lambda_{\mathrm{ut}}^{\mathrm{CIL}}$ as given in figure 4. The types of $\lambda_{\mathrm{i}}^{\mathrm{CIL}}$ are obtained from those of λ^{CIL} by erasing labels. We will informally use the notation $\langle X\rangle$ to denote the label erasure of X where X is a term, type, or type environment. The typing rules of $\lambda_{\mathrm{i}}^{\mathrm{CIL}}$ are the rules of λ^{CIL} modified by replacing every judgement $A \vdash M : \tau$ mentioned in a λ^{CIL} rule by $\langle A\rangle \vdash |M| : \langle\tau\rangle$.

While the implicitly typed language is not as useful for a compiler intermediate language as the explicitly typed language, it is helpful for comparing our approach to intersection and union types with traditional approaches. As noted before, $\lambda_{\mathrm{i}}^{\mathrm{CIL}}$ appears to be the first implicitly typed lambda calculus with intersection and union types that has the subject reduction property for a single call-by-value β-reduction step.

Theorem 4.35 (Subject ut**-Reduction for $\lambda_{\mathrm{i}}^{\mathrm{CIL}}$).** *If $\hat{M} \longrightarrow_{\mathrm{ut}} \hat{N}$ and $\tilde{A} \vdash_{\lambda_{\mathrm{i}}^{\mathrm{CIL}}} \hat{M} : \tilde{\tau}$, then $\tilde{A} \vdash_{\lambda_{\mathrm{i}}^{\mathrm{CIL}}} \hat{N} : \tilde{\tau}$.* □

Proof. Because $\tilde{A} \vdash_{\lambda_{\mathrm{i}}^{\mathrm{CIL}}} \hat{M} : \tilde{\tau}$, we know by the definition of $\lambda_{\mathrm{i}}^{\mathrm{CIL}}$ there are A, M, and τ such that $\langle A\rangle \equiv \tilde{A}$, $|M| \equiv \hat{M}$, $\langle\tau\rangle \equiv \tilde{\tau}$, and $A \vdash_{\lambda^{\mathrm{CIL}}} M : \tau$. Because $\hat{M} \longrightarrow_{\mathrm{ut}} \hat{N}$, by theorem 4.28 there is an N such that $M \longrightarrow_{\mathrm{r}} N$ and $|N| \equiv \hat{N}$. By theorem 4.23, $A \vdash_{\lambda^{\mathrm{CIL}}} N : \tau$. By the definition of $\lambda_{\mathrm{i}}^{\mathrm{CIL}}$, this implies that $\tilde{A} \vdash_{\lambda_{\mathrm{i}}^{\mathrm{CIL}}} \hat{N} : \tilde{\tau}$. □

5 Epilog

We have implemented a whole-program compiler for core Standard ML using CIL, a typed intermediate language that is based on λ^{CIL}. For implementing features of core Standard ML, CIL extends the purely functional λ^{CIL} with primitive datatypes, references, arrays, and exceptions. These extensions are described in Appendix A of (Dimock *et al.*, 2001b). Although CIL is based on λ^{CIL}, CIL itself is not a calculus. We have implemented a semantics for CIL, but we have not written its formal counterpart. While we have proven formal properties like standardization, subject reduction, and type soundness for λ^{CIL}, we have not yet established any of these properties for CIL.

The key novel feature of our compiler is its use of flow types to choose customized representations for functions. To determine the effect of customizations and pollution removal on the dynamic costs of function representations, we have measured the run-time performance (relative to a cost model) of code generated using various function customization strategies (Dimock *et al.*, 2001a). Our experiments show that flow-based customization of closed functions can give significant improvements over uniform closure representations. The efficacy of using flow types to remove representation pollution is less clear. For the benchmarks tested and the types of representation pollution detected by our compiler, the pollution removal strategies we consider often cost more in overhead than they gain via enabled customizations.

However, some strategies that use defunctionalization and flow-based inlining often achieve significant customization benefits via aggressive pollution removal.

Although CIL's listing-based intersection and union types and its duplicating term representations raise the specter of compile-time space explosion at both the term and the type level, we have not observed such blowups in practice (Dimock *et al.*, 2001b). Our experiments show that space costs in our compiler can be made tractable by using sufficiently fine-grained flow analyses together with standard hash-consing techniques. A surprising result of our experiments is that they suggest that non-duplicating formulations of intersection and union types would not achieve significantly better space complexity than our duplicating term representation. However, only one of the flow analyses we have experimented with to date expresses a non-trivial form of polyvariance, so it remains to be seen whether these results hold up in the presence of flow analyses expressing more polyvariance.

References

Agesen, Ole. (1995). The Cartesian product algorithm. *Pages 2–26 of: Proceedings of ecoop'95, seventh european conference on object-oriented programming*, vol. 952. Springer-Verlag.

Aiken, Alexander S., & Wimmers, Edward L. (1993). Type inclusion constraints and type inference. *Pages 31–41 of: Fpca '93, conf. funct. program. lang. comput. arch.* ACM.

Aiken, Alexander S., Wimmers, Edward L., & Lakshman, T. K. (1994). Soft typing with conditional types. *In:* (POPL '94, 1994).

Amadio, Roberto, & Cardelli, Luca. (1993). Subtyping recursive types. *ACM trans. on prog. langs. & systs.*, **15**(4), 575–631.

Amtoft, Torben, & Turbak, Franklyn. (2000). Faithful translations between polyvariant flows and polymorphic types. *In:* (ESOP '00, 2000).

Appel, Andrew W. (1992). *Compiling with continuations*. Cambridge University Press.

Appel, Andrew W., & Felty, Amy. (2000). A semantic model of types and machine instructions for proof-carrying code. *Pages 243–253 of: Conf. rec. popl '00: 27th ACM symp. princ. of prog. langs.*

Ariola, Zena M., & Felleisen, Matthias. (1997). The call-by-need lambda calculus. *J. funct. programming*, **3**(7).

Banerjee, Anindya. (1997). A modular, polyvariant, and type-based closure analysis. *In:* (ICFP '97, 1997).

Barbanera, Franco, Dezani-Ciancaglini, Mariangiola, & de'Liguoro, Ugo. (1995). Intersection and union types: Syntax and semantics. *Inform. & comput.*, **119**, 202–230.

Barendregt, H[endrik] P[ieter]. (1984). *The lambda calculus: Its syntax and semantics*. Revised edn. North-Holland.

Benton, Nick, Kennedy, Andrew, & Russell, George. (1998). Compiling Standard ML to Java bytecodes. *In:* (ICFP '98, 1998).

Blelloch, Guy E. 1993 (Apr.). *NESL: A nested data-parallel language*. Tech. rept. CMU-CS-93-129. School of Computer Science, Carnegie Mellon University.

Bloo, Roel, & Rose, Kristoffer Høgsbro. (1996). Combinatory Reduction Systems with explicit substitution that preserve strong normalization. *Proc. 7th int'l conf. rewriting techniques and applications*.

Briggs, P., Cooper, K. D., Harvey, T. J., & Simpson, L. T. (1998). Practical improvements to the construction and destruction of static single assignment form. *Software practice and experience*, **28**(8), 859–881.

Cejtin, Henry, Jagannathan, Suresh, & Weeks, Stephen. (2000). Flow-directed closure conversion for typed languages. *In:* (ESOP '00, 2000).

Chambers, Craig, & Ungar, David. (1989a). Customization: Optimizing compiler technology for Self, a dynamically-typed object-oriented programming language. *Pages 146–160 of: Proc. ACM SIGPLAN '89 conf. prog. lang. design & impl.*

Chambers, Craig, & Ungar, David. (1989b). Iterative type analysis and extended message splitting: Optimizing dynamically-typed object-oriented programs. *Proc. ACM SIGPLAN '90 conf. prog. lang. design & impl.*

Chambers, Craig, Dean, Jeffrey, & Grove, David. 1996 (June). *Whole-program optimization of object-oriented languages.* Tech. rept. Technical Report 96-06-02. Department of Computer Science and Engineering, University of Washington.

Curtis, Pavel. (1990). *Constrained quantification in polymorphic type analysis.* Tech. rept. CSL-90-1. XEROX PARC, CSLPubs.parc@xerox.com.

Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., & Zadeck, F. K. (1991). Efficiently computing static single assignment form and the control dependence graph. *ACM trans. on prog. langs. & systs.*, **13**(4), 451–490.

Damas, L., & Milner, Robin. (1982). Principal type schemes for functional programs. *Pages 207–212 of: Conf. rec. 9th ann. ACM symp. princ. of prog. langs.* ACM.

Dean, Jeffrey, Chambers, Craig, & Grove, David. (1995). Selective specialization for object-oriented languages. *In:* (PLDI '95, 1995).

Dimock, Allyn, Muller, Robert, Turbak, Franklyn, & Wells, J. B. (1997). Strongly typed flow-directed representation transformations. *In:* (ICFP '97, 1997).

Dimock, Allyn, Westmacott, Ian, Muller, Robert, Turbak, Franklyn, & Wells, J. B. (2001a). Functioning without closure: Type-safe customized function representations for Standard ML. *Proc. 2001 int'l conf. functional programming.* ACM Press.

Dimock, Allyn, Westmacott, Ian, Muller, Robert, Turbak, Franklyn, Wells, J. B., & Considine, Jeffrey. 2001b (Mar.). *Program representation size in an intermediate language with intersection and union types.* Tech. rept. BUCS-TR-2001-02. Comp. Sci. Dept., Boston Univ.

Duggan, D. (1999). Dynamic typing for distributed programming in polymorphic languages. *ACM trans. on prog. langs. & systs.*, **21**(1), 11–45.

Eifrig, Jonathan, Smith, Scott, & Trifonov, Valery. (1995). Type inference for recursively constrained types and its application to OOP. *Proc. 1995 mathematical foundations of programming semantics conf.* Electronic Notes in Theoretical Computer Science, vol. 1. Elsevier.

ESOP '00. (2000). *Programming languages & systems, 9th european symp. programming.* LNCS, vol. 1782. Springer-Verlag.

Fernandez, Mary F. (1995). Simple and effective link-time optimization of Modula-3 programs. *In:* (PLDI '95, 1995).

Fitzgerald, R., Knoblock, T., Ruf, E., Steensgaard, B., & Tarditi, D. (1999). *Marmot: An optimizing compiler for Java.* Technical Report 99-33. Microsoft Research.

Girard, J[ean]-Y[ves]. (1972). *Interprétation fonctionnelle et elimination des coupures de l'arithmétique d'ordre supérieur.* Thèse d'Etat, Université de Paris VII.

Hannan, John, & Hicks, Patrick. (1998). Higher-order uncurrying. *In:* (POPL '98, 1998).

Harper, Robert, & Morrisett, Greg. (1995). Compiling polymorphism using intensional type analysis. *Conf. rec. 22nd ann. ACM symp. princ. of prog. langs.* ACM.

Heintze, Nevin. (1995). Control-flow analysis and type systems. *Pages 189–206 of: Proc. 2nd int'l static analysis symp.* LNCS, vol. 983.

ICFP '97. (1997). *Proc. 1997 int'l conf. functional programming.* ACM Press.

ICFP '98. (1998). *Proc. 1998 int'l conf. functional programming.* ACM Press.

Jagannathan, Suresh, Weeks, Stephen, & Wright, Andrew. (1997). Type-directed flow analysis for typed intermediate languages. *Proc. 4th int'l static analysis symp.* LNCS, vol. 1302. Springer-Verlag.

Jim, Trevor. (1996). What are principal typings and what are they good for? *Conf. rec. popl '96: 23rd ACM symp. princ. of prog. langs.* ACM.

Jones, Mark P. (1994). Dictionary-free overloading by partial evaluation. *Pepm '94 — ACM SIGPLAN workshop partial eval. & semantics-based prog. manipulation.*

Kfoury, A. J., & Wells, J. B. (1995). New notions of reduction and non-semantic proofs of β-strong normalization in typed λ-calculi. *Pages 311–321 of: Proc. 10th ann. IEEE symp. logic in computer sci.*

Kfoury, Assaf J., & Wells, J. B. (1999). Principality and decidable type inference for finite-rank intersection types. *Pages 161–174 of: Conf. rec. popl '99: 26th ACM symp. princ. of prog. langs.*

Klop, Jan Willem. (1980). *Combinatory Reduction Systems.* Amsterdam: Mathematisch Centrum. Ph.D. Thesis.

Klop, Jan Willem, van Oostrom, Vincent, & van Raamsdonk, Femke. (1993). Combinatory Reduction Systems: Introduction and survey. *Theoret. comput. sci.*, **121**(1–2), 279–308.

Leroy, Xavier. (1992). Unboxed objects and polymorphic typing. *Pages 177–188 of: Conf. rec. 19th ann. ACM symp. princ. of prog. langs.* ACM.

Morrisett, G., Walker, D., Crary, K., & Glew, N. (1999). From System F to typed assembly language. *ACM trans. on prog. langs. & systs.*, **21**(3), 528–569.

Morrisett, Greg. (1995). *Compiling with types.* Ph.D. thesis, Carnegie Mellon University.

Muller, Robert, & Wells, J. B. (2000). *Two applications of standardization and evaluation in Combinatory Reduction Systems.* Submitted for publication.

Necula, George C., & Lee, Peter. (1998). The design and implementation of a certifying compiler. *Pages 333–344 of: Proc. ACM SIGPLAN '98 conf. prog. lang. design & impl.*

Nielson, Flemming, & Nielson, Hanne Riis. (1997). Infinitary control flow analysis: A collecting semantics for closure analysis. *Pages 332–345 of: Conf. rec. popl '97: 24th ACM symp. princ. of prog. langs.*

Palsberg, Jens, & O'Keefe, Patrick. (1995). A type system equivalent to flow analysis. *ACM trans. on prog. langs. & systs.*, **17**(4), 576–599.

Palsberg, Jens, & Pavlopoulou, Christina. (2001). From polyvariant flow information to intersection and union types. *J. funct. programming*, **11**(3), 263–317.

Palsberg, Jens, & Smith, Scott. (1996). Constrained types and their expressiveness. *ACM trans. on prog. langs. & systs.*, **18**(5), 519–527.

Peyton Jones, Simon L. (1996). Compiling Haskell by program transformation: A report from the trenches. *Proc. european symp. on programming.*

Peyton Jones, Simon L., & Meijer, Erik. 1997 (June). Henk: A typed intermediate language. *In:* (TIC '97, 1997).

Pierce, Benjamin C. 1991 (Feb.). *Programming with intersection types, union types, and polymorphism.* Tech. rept. CMU-CS-91-106. Carnegie Mellon University.

PLDI '95. (1995). *Proc. ACM SIGPLAN '95 conf. prog. lang. design & impl.*

Plevyak, John. (1996). *Optimization of object-oriented and concurrent programs.* Ph.D. thesis, University of Illinois at Urbana-Champaign.

Plevyak, John, & Chien, Andrew. 1995 (Aug.). Type directed cloning for object-oriented programs. *Workshop for languages and compilers for parallel computers.*

Plotkin, G[ordon] D. (1975). Call-by-name, call-by-value and the lambda calculus. *Theoret. comput. sci.*, **1**, 125–159.

POPL '94. (1994). *Conf. rec. 21st ann. ACM symp. princ. of prog. langs.* ACM.

POPL '98. (1998). *Conf. rec. popl '98: 25th ACM symp. princ. of prog. langs.*

Reynolds, J. C. (1974). Towards a theory of type structure. *Pages 408–425 of: Colloque sur la programmation.* LNCS, vol. 19. Paris, France: Springer-Verlag.

Reynolds, John C. (1996). Design of the programming language Forsythe. O'Hearn, P., & Tennent, R. D. (eds), *Algol-like languages.* Birkhauser.

Shao, Zhong. 1997 (June). An overview of the FLINT/ML compiler. *In:* (TIC '97, 1997).

Shao, Zhong, League, Christopher, & Monnier, Stefan. (1998). Implementing typed intermediate languages. *In:* (ICFP '98, 1998).

Siskind, Jeffrey Mark. 1999 (Dec.). *Flow-directed lightweight closure conversion.* Tech. rept. 99-190R. NEC Research Institute, Inc.

Tang, Yan Mei, & Jouvelot, Pierre. (1994). Separate abstract interpretation for control-flow analysis. *Lncs*, **789**, 224–243.

Tarditi, D., Morrisett, G., Cheng, P., Stone, C., Harper, R., & Lee, P. (1996). TIL: A type-directed optimizing compiler for ML. *Proc. ACM SIGPLAN '96 conf. prog. lang. design & impl.*

Tarditi, David. 1996 (Dec.). *Design and implementation of code optimizations for a type-directed compiler for Standard ML.* Ph.D. thesis, Carnegie Mellon University.

TIC '97. 1997 (June). *Proc. first int'l workshop on types in compilation.* The printed TIC '97 proceedings is Boston Coll. Comp. Sci. Dept. Tech. Rep. BCCS-97-03. The individual papers are available at http://www.cs.bc.edu/~muller/TIC97/ or http://oak.bc.edu/~muller/TIC97/.

Tolmach, Andrew P., & Oliva, Dino. (1998). From ML to Ada: Strongly-typed language interoperability via source translation. *J. funct. programming*, **8**(4), 367–412.

Urzyczyn, Paweł. (1997). Type reconstruction in $\mathbf{F}_\omega$. *Math. structures comput. sci.*, **7**(4), 329–358.

Wand, Mitchell, & Steckler, Paul. (1994). Selective and lightweight closure conversion. *In:* (POPL '94, 1994).

A Combinatory Reduction Systems

We use the *functional* presentation of CRS's (Klop *et al.*, 1993). An alternative *applicative* presentation can be found in (Klop, 1980). Both ways of presenting CRS's have the same expressiveness; they only differ in the number of "garbage terms" that must be ignored.

A CRS Σ is specified by a set of *function symbols* $\text{Fun}(\Sigma)$ (in the applicative presentation, a set of constants) and a set of *reduction rules* $\text{Red}(\Sigma)$ (sometimes called *rewrite rules*). Each function symbol F has a fixed arity n, which we denote by writing $F^{(n)}$. We will often omit the arity from function symbols and metavariables when writing terms since it will be obvious from the context. The function symbols are the only part of the CRS's *alphabet* which can vary from CRS to CRS. The fixed part of the alphabet includes the set of variables Var and the set of metavariables MVar. The set of *metaterms* and the set of *terms* are determined by the set of function symbols $\mathcal{F}$ (where $\mathcal{F} = \mathrm{Fun}(\Sigma)$ for some CRS Σ) together with the fixed portion of the alphabet. Let u and v range over terms and let s and t range over metaterms. The set of metaterms $\mathrm{MTer}\mathcal{F}$ is the smallest set satisfying all of the following:

1. If $x \in \mathrm{Var}$ (i.e., x is a (ordinary) variable), then $x \in \mathrm{MTer}\mathcal{F}$.
2. If $x \in \mathrm{Var}$ and $s \in \mathrm{MTer}\mathcal{F}$, then $[x]s \in \mathrm{MTer}\mathcal{F}$. (This construct declares a variable x which may be used in s. The variable x is *bound* by this construct.)
3. If $F^{(n)} \in \mathcal{F}$ and $s_1, \ldots, s_n \in \mathrm{MTer}\mathcal{F}$, then $F^{(n)}(s_1, \ldots, s_n) \in \mathrm{MTer}\mathcal{F}$.
4. If $Z^{(n)} \in \mathrm{MVar}$ (i.e., Z is a metavariable with fixed arity n) and $s_1, \ldots, s_n \in \mathrm{MTer}\mathcal{F}$, then $Z^{(n)}(s_1, \ldots, s_n) \in \mathrm{MTer}\mathcal{F}$.

The set of terms $\mathrm{Ter}(\mathcal{F})$ is the subset of $\mathrm{MTer}(\mathcal{F})$ containing only those metaterms which do not mention metavariables. The notion of (one-holed) *context* is defined for metaterms and terms as usual.

A *valuation* $\nu : \mathrm{MVar} \to \mathrm{MTer}\mathcal{F}$ is a function mapping metavariables to metaterms such that for any metavariable $Z^{(n)}$, the metaterm $\nu(Z^{(n)})$ mentions only metavariables in the set $\{Z_1^{(0)}, \ldots, Z_n^{(0)}\}$. A valuation ν is automatically extended to a function from $\mathrm{MTer}\mathcal{F}$ to $\mathrm{Ter}(\mathcal{F})$ as follows[10]:

1. $\nu(x) = x$.
2. $\nu([x]s) = [x]\nu(s)$ (assuming by α-conversion that x is not mentioned in the range of ν).
3. $\nu(F^{(n)}(s_1, \ldots, s_n)) = F^{(n)}(\nu(s_1), \ldots, \nu(s_n))$.
4. $\nu(Z^{(n)}(s_1, \ldots, s_n)) = \nu'(\nu(Z^{(n)}))$ where $\nu'(Z_i^{(0)}) = \nu(s_i)$ for $1 \leq i \leq n$.

Each reduction rule r of a CRS is a pair $s \to t$ (where s is the left-hand side (LHS) and t is the right-hand side (RHS)) of metaterms obeying the following conditions:

1. Neither s nor t has free (ordinary) variables, i.e., each variable x occurs in the scope of a binder $[x]$.
2. The LHS is of the form $F(s_1, \ldots, s_n)$ for some function symbol F and some metaterms $s_1, \ldots, s_n$.
3. Any metavariable which occurs in the RHS also occurs in the LHS.
4. Any metavariable $Z^{(n)}$ (of arity n) occurs in the LHS only in the form $Z^{(n)}(x_1, \ldots, x_n)$ where $x_1, \ldots, x_n$ are n distinct (ordinary) variables.

Any reduction rule $r = s \to t$ automatically determines a *reduction relation* $\longrightarrow_r$ (sometimes called a *rewrite relation*) such that $C[\nu(s)] \longrightarrow_r C[\nu(t)]$ for every valuation ν and every term context C. Any set of reduction rules R determines a reduction relation $\longrightarrow_R = \bigcup_{r \in R} \longrightarrow_r$.

A reduction rule $s \to t$ is *left-linear* if every metavariable in s (the LHS) occurs in s exactly once. A set of reduction rules R is left-linear if every reduction rule $r \in R$ is left-linear.

Two metaterms s and t *interfere* iff for some valuations ν and ν' and some context

[10] This definition of valuation differs from that of (Klop *et al.*, 1993) and (Klop, 1980) (which differ from each other anyway), but produces equivalent results.

C it is the case that (1) $\nu(s) = C[\nu'(t)]$ and (2) the position[11] of the hole in C is a position in s which is not occupied by a metavariable. The interference is *at the root* iff C is the empty context. A pair of reduction rules $s \to t$ and $s' \to t'$ is *ambiguous* (sometimes called *overlapping*) iff s and s' interfere and either the two rules are distinct or the interference is not at the root. A set of reduction rules R is ambiguous iff there exists an ambiguous pair of rules $r, r' \in R$ (where r and r' may be the same rule).

A CRS Σ is *regular* (also called *orthogonal*) if and only if the set of reduction rules $\mathrm{Red}(\Sigma)$ are left-linear and non-ambiguous. We write $\longrightarrow_\Sigma$ as an abbreviation for $\longrightarrow_{\mathrm{Red}(\Sigma)}$.

Theorem A.1 (Confluence of Regular CRS's). *If Σ is a regular CRS, and $u \twoheadrightarrow_\Sigma v_1$, and $u \twoheadrightarrow_\Sigma v_2$, then there exists u' such that $v_1 \twoheadrightarrow_\Sigma u'$ and $v_2 \twoheadrightarrow_\Sigma u'$.* □

Proof. See (Klop, 1980) or (Klop *et al.*, 1993) for proofs that any regular (orthogonal) CRS is confluent. □

[11] We leave this notion of *position* unspecified. See (Klop *et al.*, 1993) or (Klop, 1980) for a more precise definition.