A Calculus with Polymorphic and Polyvariant Flow Types*

J. B. Wells†
University of Glasgow
jbw@dcs.gla.ac.uk

Allyn Dimock
Harvard University
dimock@das.harvard.edu

Robert Muller
Boston College
muller@cs.bc.edu

Franklyn Turbak
Wellesley College
fturbak@wellesley.edu

August 28, 1999

Abstract

We present λ^{CIL}, a typed λ-calculus which is intended to serve as the basis of an intermediate language for optimizing compilers for higher-order polymorphic programming languages. The language λ^{CIL} has function, product, sum and recursive types. In addition, λ^{CIL} has a novel formulation of intersection and union types and flow labels on both terms and types. The calculus is formulated to support the integration of polyvariant flow information in a polymorphically typed program representation. These *flow types* can guide a compiler in generating customized data representations in a strongly typed setting. We prove that the calculus satisfies confluence and subject reduction properties.

1 Introduction

Higher-order polymorphic programming languages are widely understood to be more expressive but less efficient than their first-order monomorphic counterparts. The inefficiencies arise in part from difficulties in computing efficient data and control representations in the presence of the abstraction facilities that characterize these languages and in part from a compiler organizing principle, the *uniform representation assumption* (URA), which dictates that each type in the language is associated with a single representation in the target machine language. Although some of the implementation costs of this class of languages may be unavoidable, improvements in run-time performance can be expected from relaxing the URA and supporting customized data and control representations.

In this paper we introduce λ^{CIL}, an explicitly typed λ-calculus which serves as a basis for an intermediate language in a compiler for a higher-order polymorphic programming language.[1] The

*This paper supersedes [WDMT97]. It has been accepted, subject to revision, in the *Journal of Functional Programming*.

†This author's work was done while at Boston University and the University of Glasgow and was partially supported by NSF grants CCR–9113196 and CCR–9417382 and EPSRC grant GR/L 36963.

[1] "CIL" is an acronym for "Church Intermediate Language". The authors are members of of the Church Project (`http://www.cs.bu.edu/groups/church/`) which is investigating the application of types in compiling ML–like languages.

key innovation of the language is the integration of polyvariant flow information and polymorphic type information into a single *flow type* system which supports transformations into customized data representations. In λ^{CIL}, terms and types have annotations that approximate the flow of values from the points where they are produced (their *sources*) to the points where they are consumed (their *sinks*). Flow annotations encode control and data flow information as it would be computed by one of several typed flow analyses in the literature [Hei95, Ban97, Mos97, JWW97]. These flow annotations can be used to construct data representations tailored specifically to source/sink pairs and to implement efficient control structures.

The calculus λ^{CIL} is an explicitly typed λ-calculus with function, product, sum and recursive types. In addition, λ^{CIL} has a novel formulation of intersection and union types and flow labels on both terms and types. Flow information is represented in λ^{CIL} using two features:

1. *Flow labels*: Source and sink terms are annotated with flow labels. Each type constructor (a function type in particular) is annotated with a set of source labels approximating the source terms producing the values of that type and a set of sink labels approximating the sink terms consuming the values of that type.

2. *Intersection and union types*: Intersection and union types [vB93, Pie91, BDCd95] are polymorphic types which can be used to encode polyvariant flow information. A value v flowing to m sinks can be assigned an intersection type with m conjuncts. In λ^{CIL}, this value would be represented as a *virtual tuple* containing m copies of v. When v is a polymorphic function, the intersection type can be viewed as a finitary form of a universal type in which the concrete types at which the polymorphic function is instantiated are represented explicitly. Polyvariance can also be represented with intersection types by using one conjunct for each of the multiple analyses of a function. Dually, a set of n values flowing to a given sink can be represented by a *virtual sum* (a value of union type) with n disjuncts. The sink can be expressed as *virtual case expression* which dispatches to one of n clauses based on the virtual injection tag. A union type can be viewed as a finitary form of an existential type [MP88, MMH96] in which each of the possible source types are represented explicitly.

   From a theoretical point of view, flow labels are unnecessary since any flow information representable with label-annotated types can instead be represented with intersection and union types. However, from an engineering point of view, flow labels are important for naming purposes and because sets of flow labels are more compact.

Virtual tuples and virtual case expressions provide support for the compiler in customizing data representations. For example, the compiler may choose to represent a function flowing to several application sites with several customized pieces of run-time code. If the function is polymorphic, the run-time code may be customized specifically for the type at which it is instantiated. This type of customization, called *type specialization* in the literature, is particularly effective since it avoids the overhead of boxing (see Section 2). Irrespective of whether the function is polymorphic, the run-time code may be customized for other context dependent properties such as callee-save register conventions, activation record layout, stack availability of free variable values, etc. When the compiler elects to customize a source, it *splits* the source by converting the virtual tuple into a real tuple and converting the corresponding virtual projections to real ones. 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 splits the sink by converting the

virtual sum to a real one (i.e., one with run-time tags) and converting the corresponding virtual case expression to a real one. See [DMTW97] for an extensive discussion of how λ^{CIL} supports customized function representations.

As a calculus, λ^{CIL} supports equational reasoning. We prove that it is confluent and satisfies the subject reduction property. Because flow information is represented in the types, the latter property guarantees that the flow annotations are sound with respect to the rewrite rules of the calculus.

λ^{CIL} is an example of a *typed intermediate language* (TIL). TILs have been used in several recent compilers for higher-order polymorphic languages [Sha94, Mor95, PJ96, PJM97]. The key motivation for the explicit types in these languages is to transmit type information available in the early stages of the compiler to the later stages of the compiler through a series of program transformations that preserve the well typedness of the intermediate language programs. In the later stages of the compiler, the type information can be used to guide data representation decisions and optimizations and to provide accurate type information needed at run-time (e.g., for garbage collection). Types also serve as an important debugging aid in the compiler development process.

While most other typed intermediate languages have been based on System F [Gir72, Rey74], λ^{CIL} is based on intersection and union types.[2] The integration of flow and type information in λ^{CIL}, made possible by the combination of flow labels and union and intersection types, has several benefits:

- The flow information embedded in types can be used to guide a wide range of optimizations and representation decisions that are not possible with traditional type information alone. Flow information is critical for distinguishing usage contexts that traditional types do not distinguish. For instance, flow information can:
  - indicate the instantiation types of a polymorphic function and enable specialization of the function at these types.
  - override inefficiencies implied by the universal representation assumption by exposing the opportunity for multiple representations of monomorphically typed data.
  - enable classical optimizations like constant propagation, dead code elimination, and loop transformations by detecting flow patterns in a maze of higher-order function calls.
- Type-preserving compiler transformations necessarily preserve the flow analysis embedded in the types, i.e., the flow analysis must continue to be a conservative approximation of the actual flow. This makes it unnecessary to recompute the flow analysis on the result of the transformation unless greater precision is desired.
- The additional flow information can aid in correctness proofs and in the debugging of compiler implementations.

[2] The omission of universal and existential quantifiers within the λ^{CIL} type system is possible largely because of an assumption of whole-program compilation, in which the instantiation types of polymorphic functions and representation types of abstract data are all known. Extensions to λ^{CIL} supporting separate compilation are likely to include some form of universal and existential quantification for the purpose of deciding whether a program is a legal member of the language.

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 used as the basis for a compiler intermediate language. Heintze coined the term "control flow types" in the context of a system in which function types are annotated with sets of source labels [Hei95]; the system does not support sink labels and is limited to the expression of monovariant flow analyses. Jouvelot and Tang's control flow effect system [TJ94] also involves source label annotations of arrow types but cannot express polyvariant flow analyses. Jim uses intersection types to model the **let**-style polymorphism of ML [Jim95, Jim96] in the context of type inference. Banerjee combines intersection types with source label annotations to encode a polyvariant flow analysis [Ban97], but again, the emphasis is on type inference, not on compilation. The first formal correspondence between a class of polyvariant flow analyses and intersection and union types was shown by Palsberg and Pavlopoulou [PP98]. Flow and type information are also merged in constrained types [Cur90, EST95, AW93, AWL94], but the class of flow analyses expressible in these systems is not clear.

The traditional approach to using program analysis in compilation involves maintaining the results of the analysis separately from the type information of a typed intermediate language. In this approach, any formalism connecting an analysis to the typed program is outside the type system. One consequence is that well typedness preserving program transformations are not guaranteed to preserve the results of an analysis. In general, it may be necessary to reanalyze the program after every transformation. Not only is reanalysis expensive, but analysis of a transformed program may yield less precise information than for the original program[3], possibly causing loss of well typedness.

The remainder of this paper is organized as follows. Section 2 reviews the sources of inefficiency in higher-order polymorphic languages and explains why both flow and type information are important for compiling these languages. Section 3 presents an informal overview of λ^{CIL} motivating the properties of the calculus with examples. Section 4, discusses design trade-offs in the formulation of the calculus. Section 5 presents (three versions of) the calculus and proves fundamental properties including confluence and subject reduction. Section 6 concludes with a discussion of future work. The appendix presents the details of combinatory reduction systems which are used in our proof of confluence.

2 Background

Higher-order polymorphic programming languages offer abstraction facilities that enhance expressiveness but are challenging to implement efficiently. The abstraction mechanisms simplify the construction of reusable software by hiding details about how values are produced and consumed — the very sort of implementation-specific information that compilers need to determine efficient concrete data representations.

Most compilers for this class of languages account for this opacity by adopting the URA. This is a reasonably simple organizing principle for compilers which provides a basis for correct implementations and which naturally supports separate compilation. Unfortunately, the URA has serious performance disadvantages: it leads to significant costs in time and space to build and take apart abstractions at run-time, it prevents the compiler from performing optimizing program

[3]This is particularly true in the case of modular analyses such as that of Banerjee [Ban97], 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. 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.

transformations that depend on context-specific information, and it hinders interoperability with other languages.

Below is a summary of some of the sources of inefficiency for implementations of higher-order polymorphic languages, followed by a discussion of how type and flow information can be used to improve performance. The key observation is that whereas traditional type information can be effective for reducing the cost of polymorphism, it is not particularly effective for reducing the costs of higher-order functions nor the one-representation-per-type costs implied by the URA. Flow information appears to be more valuable for reducing these other costs.

- *Heavy use of functions:* Function-oriented languages (even monomorphic ones) naturally encourage a programming style characterized by the use of many small functions. As is well-known, function calls complicate program analysis and their straightforward implementation can incur a substantial run-time overhead. Since function bodies are typically small, intraprocedural optimizations usually have little benefit. And since common control idioms like iteration are typically expressed via function calls, loop optimizations that would be intraprocedural in the presence of a manifest loop syntax instead require interprocedural analysis.

- *Function Values:* Higher-order functions make it difficult to compute the control and data flow graphs required for standard Dragon book [ASU85] optimizations. Furthermore, whereas closed functions can be represented by a code pointer, open functions (i.e., those with free variables) are typically represented as some sort of *closure* that combines a code pointer and an environment of free variable values. Closures are especially inefficient in compilers employing the URA, since all functions of a given type are assumed to have the same closure representation, regardless of whether they are open. So not only are functions extremely common, but the overhead for creating and calling them is particularly high.

- *Type polymorphism:* In System F [Gir72, Rey74], a function f used at different types is assigned a universal type $\forall\alpha.\tau$. For the compiler writer, this means that the code generated for f must work properly at the instantiated type $\tau[\alpha := \sigma]$ for *any* type σ. Since different data types have different representations on standard architectures, it is natural to generate one piece of code for f that manipulates values of uniform size (typically pointers). This approach, known as *boxing*, has been used for many years in compilers for LISP [Gre77, BGS82] and Scheme [Ste78].

  Since it is non-trivial to determine which values will be used in polymorphic contexts, the URA implies the stringent requirements that (1) *all* values be must boxed and (2) data representations are determined by type *constructor* and not just by *type*. For instance, lists of integers and lists of reals (i.e., double-precision floating point numbers) will have exactly the same node structure. Similarly, a function mapping integers to integers will have the same calling convention as a function mapping reals to reals. Boxing incurs both time and space costs relative to the more efficient representations that would be used by compilers for monomorphic languages like C and Pascal. It also serves as a barrier to traditional compiler optimizations and hinders interoperability with languages using unboxed representations.

- *Abstract data types:* Since the representation type of a value of an abstract data type (ADT) is hidden, the URA suggests that a compiler treat it with a uniform interface. So ADTs

imply boxing penalties similar to those for polymorphism. Indeed, ADTs can be viewed as polymorphism for *contexts* in which the contexts that manipulate the ADT values are like polymorphic functions.

Much recent work on compiling higher-order polymorphic languages has focused on relaxing the URA and supporting customized representations. The goal is to design compilers that take programs making heavy use of abstraction and translate them into code that uses efficient data and control representations similar to those that might be employed if the programmer wrote the program in a first-order, monomorphic language like C or Pascal.[4] A common technique in this work is to use information gleaned from static program analyses, particularly type and flow analysis, to guide the selection of data representations.

Type information has been widely used to optimize the implementation of polymorphism. Modeling boxing via coercions that mediate between unboxed and boxed types makes it possible to avoid the boxing of values that are used only in monomorphic contexts [PJL91, Ler92, HJ94, Sha94, SA95, Ler97]. While boxing coercions can improve the compilation of monomorphic functions and data structures, their polymorphic counterparts remain inefficient. The most common technique for improving polymorphic code is *specialization*, which makes monomorphic copies of the polymorphic code for each type at which it the polymorphic code is (or might be) used. Such copying introduces a threat of code blowup, but such a blowup may not be observed in practice [Jon94]. The specialization approach has been considered in the context of compiling NESL in [Ble93], in the context of resolving overloading in Haskell [Jon94] and particularly in optimizing method invocation in object-oriented languages [CU89a, CU89b, Age95, DCG95, PC95, Ple96].

Specialization techniques differ mainly in terms of how the instances of the polymorphic types are determined. If the range of instances can be enumerated independently from information about the instantiation sites, then a simple approach is to specialize to all of the types in the enumeration. For example, implementations of some object-oriented languages specialize each method for all subclasses of the type of the method receiver [CU89a, CU89b]. In *dynamic type dispatch* [HM95, Mor95] polymorphic functions dispatch to monomorphic code based on all possible representation types at which the function can be used; 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 for polymorphism.

An alternative to specializing a polymorphic function for all possible instances of its polymorphic type is to use flow analysis to better approximate the set of types at which polymorphic code is actually used.[5] For example, in the **let**-style polymorphism of ML, type instances determined by a trivial flow analysis have been used to specialize polymorphic functions [SA95]. More complex flow analyses have been used in object-oriented languages to reduce the number of specialized copies [Age95, DCG95, PC95, Ple96].

Flow analysis has also been used to reduce the implementation costs of higher-order functions. When it can be shown that only one function flows to a given call site (a so-called *known function*), the invocation protocol can be streamlined or the call site can be replaced by inlining the function body [App92, Ash96]. Sophisticated flow analysis can enable inlining that is not locally apparent or that involves the inlining of open functions [JW96, DMTW97]. Even when multiple functions

[4] Though both C and Pascal do support some aspects of higher-order functions, these are so limited that we will classify the languages as first-order.

[5] This approach generally assumes that the whole program is available for flow analysis.

flow to a call site, it is possible to replace the call site by code that dispatches to inlined versions of the functions based on a tag associated with the function being called [DMTW97]; this technique has been used to compile method invocation in object-oriented languages. Inlining is an especially important transformation because it exposes many opportunities for traditional local optimizations [App92, SA95, Tar96]. Flow analysis can also be used to detect candidates for optimization that are non-local, as in the case of loop detection across procedure boundaries [Ash96] and type recovery for the elimination of run-time type checks [Shi91, JW95].

3 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 some simple examples. We present three distinct versions of the calculus, which differ in terms of whether they include types and/or flow label annotations.[6]

1. The *untyped* language $\lambda^{\mathrm{CIL}}_{\mathrm{ut}}$ has neither types nor flow labels.
2. The *unlabelled* language $\lambda^{\mathrm{CIL}}_{\mathrm{ul}}$ has types but no flow labels.
3. The fully *typed* language λ^{CIL} has both types and flow labels.

We consider these three languages in turn.

3.1 The Untyped Language $\lambda^{\mathrm{CIL}}_{\mathrm{ut}}$

The untyped language $\lambda^{\mathrm{CIL}}_{\mathrm{ut}}$ 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 $\lambda^{\mathrm{CIL}}_{\mathrm{ut}}$ term:

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

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 λ^{CIL} (see section 5.2.1) are straightforward. Under these rules, $\hat{M}_a$ reduces to the normal form $\times(17, 23, \textbf{true})$.

[6] Section 5 presents yet another language, the implicitly typed language $\lambda^{\mathrm{CIL}}_{\mathrm{i}}$, which is different from each of the three languages listed here. Because $\lambda^{\mathrm{CIL}}_{\mathrm{i}}$ is not critical to the informal exposition, we omit it from the current section.

Our second example illustrates variants:

$$\begin{array}{rl}\hat{M}_b \equiv & \textbf{let } g = \lambda s.\ \textbf{case}^+ s \textbf{ bind } w \textbf{ in} \\ & \qquad \times(\lambda x.x+1, w), \\ & \qquad \times(\lambda y.y*2, w+1), \\ & \qquad \times(\lambda z.\textbf{if } z \textbf{ then } 1 \textbf{ else } 0, w) \\ & \textbf{in } \textbf{let } h = \lambda a.\ \textbf{let } p = g \mathbin{@} a \\ & \qquad\qquad \textbf{in}(\pi_1^\times\ p) \mathbin{@} (\pi_2^\times\ p) \\ & \quad \textbf{in } \times\big(h \mathbin{@} \big(\textbf{in}_1^+\ 3\big), h \mathbin{@} \big(\textbf{in}_2^+\ 5\big), h \mathbin{@} \big(\textbf{in}_3^+\ \textbf{true}\big)\big)\end{array}$$

In this example, g and h are **let**-bound to functions that take variants as arguments. The term $\pi_i^\times\ M$ extracts the ith component of the tuple to which M evaluates. Variant values 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 [DMTW97]. The term $\hat{M}_b$ reduces to the normal form $\times(4, 12, 1)$.

3.2 The Unlabelled Language $\lambda_{\text{ul}}^{\text{CIL}}$

The typed but unlabelled language $\lambda_{\text{ul}}^{\text{CIL}}$ is an extension of $\lambda_{\text{ut}}^{\text{CIL}}$ in which all occurrences of variables (except the binding occurrences of **case**-bound variables) and variants are annotated with a type. In addition to the more familiar types — base types, function types, tuple types, variant types, and recursive types — $\lambda_{\text{ul}}^{\text{CIL}}$ provides intersection and union types.

In intersection type systems, the types of polymorphic functions are generally represented as collections (intersections) of the types at which the functions are used. For example, the identity function $\lambda x.x$ appearing in untyped term $\hat{M}_a$ can be assigned the type $\bigwedge[\text{int} \to \text{int}, \text{bool} \to \text{bool}]$ because it is applied only to integers and booleans. In type systems based on System F [Gir72, Rey74], the identity function is normally assigned the universal type $\forall\alpha.(\alpha \to \alpha)$. Universal types do not provide any information about the actual types at which polymorphic functions are used. Universal types are suggestive of a uniform representation, which entail implementation overheads such as boxing. In contrast, intersection types expose the types at which polymorphic functions are used. This information can be used to guide representation decisions in a compiler (e.g., type-directed specialization for uses of a polymorphic function at different types).

In a type system with union types, abstract data types are generally represented as collections (unions) of the possible implementation types. Union types allow values of otherwise incompatible types to be used in the same context as long as they are only manipulated in a way that does not expose their incompatibilities. This is the essence of data abstraction, in which clients of an abstract data type use a single protocol that is compatible with all implementations of the abstraction. For example, in the untyped term $\hat{M}_b$, the thunks $\times(\lambda x.x+1, w)$ and $\times(\lambda y.y*2, w+1)$ can be

assigned the type $\times[\text{int} \to \text{int}, \text{int}]$, while the thunk $\times(\lambda z.\textbf{if } z \textbf{ then } 1 \textbf{ else } 0, w)$ can be assigned the type $\times[\text{bool} \to \text{int}, \text{bool}]$. Even though these types are different, the values with these types are used only in a way that does not expose the differences (i.e., applying the function in the first tuple component to the second component to yield an integer). Traditionally, this situation is modeled with existential types [MP88, MMH96], which hide the unobservable types. For example, both of the above types are instances of the existential type $\exists\tau.\times[\tau \to \text{int}, \tau]$. As with universal types, existential types hide usage information and imply unnecessary implementation overheads. Union types expose the implementation types of a data abstraction. For example, the union type for the thunks would be $\bigvee[\times[\text{int} \to \text{int}, \text{int}], \times[\text{bool} \to \text{int}, \text{bool}]]$.

Here is a $\lambda^{\text{CIL}}_{\text{ul}}$ term corresponding to the untyped term $\hat{M}_a$:[7]

$$\begin{aligned}\tilde{M}_a \equiv\ & \textbf{let } f^{\wedge[\text{int}\to\text{int},\text{bool}\to\text{bool}]} = \textstyle\bigwedge(\lambda x^{\text{int}}.x, \lambda x^{\text{bool}}.x) \\ & \textbf{in } \times((\pi^{\wedge}_1\, f) \mathbin{@} 17, (\pi^{\wedge}_1\, f) \mathbin{@} 23, (\pi^{\wedge}_2\, f) \mathbin{@} \textbf{true})\end{aligned}$$

The notation $\bigwedge(\lambda x^{\text{int}}.x, \lambda x^{\text{bool}}.x)$ designates a term of intersection type, which is known as a a *virtual tuple*. 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 $\pi^{\wedge}_i\, M$ 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 $\lambda^{\text{CIL}}_{\text{ul}}$ 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.

The notation for virtual tuples and virtual projections has two key benefits:

- It solves an important technical problem: how to annotate the bound variables of terms of intersection type in an explicitly typed language. Previous approaches that allow bound variables to range over instantiation types [Rey96a, Pie91] cannot express some of the typings expressible in our system. In essence, our term syntax is isomorphic to typing derivations, so every typing derivation can be expressed as a term. This is discussed in greater detail in section 4.3.

- The notational similarity between products and virtual products is specifically designed to suggest splitting transformations. In λ^{CIL}, a source term can be split by converting the virtual tuple and its corresponding projections are transformed into real tuples and projections. This is the formalization of type-based specialization in our system. For example, suppose that a function that swaps the components of a 2-tuple has the type

  $$\textstyle\bigwedge[\times[\text{int}, \text{bool}] \to \times[\text{bool}, \text{int}], \times[\text{real}, \text{real}] \to \times[\text{real}, \text{real}]]$$

  Although it mentions two usage types, this intersection type specifies a single polymorphic function. But if unboxed tuple component representations are desired, there must be two distinct functions, since different code will need to be executed for swapping integers and

[7] 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.

booleans than for swapping reals (assuming that reals have a different size from integers and booleans). This specialization is expressed by converting the $\wedge$ to $\times$ in the type and corresponding term and by converting the associated occurrences of $\pi_i^{\wedge}$ to $\pi_i^{\times}$. (Finding the corresponding occurrences is facilitated by the flow labels in the typed and labelled language.) Splitting is an important technique in the representation transformation framework based on λ^{CIL} [DMTW97].

An intersection type encodes 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 $\lambda_{\mathrm{ul}}^{\mathrm{CIL}}$, 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 an alternative typing of the untyped term $\hat{M}_a$:

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

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 a value which would be given a monomorphic type by a traditional type system. The ability to distinguish the flows of different values generated by a single source term is a hallmark of *polyvariant* flow analysis [Ban97, JWW97, PP98]. The above example illustrates how intersection types can encode polyvariance in $\lambda_{\mathrm{ul}}^{\mathrm{CIL}}$.

Although the above examples happen to require only polymorphism which can be encoded in the Hindley-Milner ("**let**-style") system, we stress that intersection types can handle complex flows not expressible in **let**-style polymorphism. For instance, here is a term illustrating how a polymorphic function can be returned as a result and passed as an argument — first-class features not supported by **let**-style polymorphism:

$$\begin{array}{rl}\tilde{M}_c \equiv & (\lambda f^{\wedge[\mathrm{int}\to\mathrm{int},\mathrm{bool}\to\mathrm{bool}]}.\times((\pi_1^{\wedge}\, f) \mathbin{@} 17, (\pi_1^{\wedge}\, f) \mathbin{@} 23, (\pi_2^{\wedge}\, f) \mathbin{@} \textbf{true})) \\ & \mathbin{@} ((\lambda z^{\mathrm{int}\to\wedge[\mathrm{int}\to\mathrm{int},\mathrm{bool}\to\mathrm{bool}]}.\bigwedge\big(\lambda x^{\mathrm{int}}.x, \lambda x^{\mathrm{bool}}.x\big)) \mathbin{@} 42)\end{array}$$

Here is an example that illustrates a more complex use of polymorphic functions as arguments:

$$\begin{array}{rl}\tilde{M}_d \equiv & \textbf{let } p^{\wedge[\wedge[\mathrm{int}\to\mathrm{int},\mathrm{bool}\to\mathrm{bool}]\to\times[\mathrm{int},\mathrm{int},\mathrm{bool}],\wedge[\mathrm{int}\to\mathrm{real},\mathrm{bool}\to\mathrm{real}]\to\times[\mathrm{real},\mathrm{real},\mathrm{real}]]} = \\ & \quad \bigwedge(\ \lambda f^{\wedge[\mathrm{int}\to\mathrm{int},\mathrm{bool}\to\mathrm{bool}]}.\times((\pi_1^{\wedge}\, f) \mathbin{@} 17, (\pi_1^{\wedge}\, f) \mathbin{@} 23, (\pi_2^{\wedge}\, f) \mathbin{@} \textbf{true}), \\ & \qquad \lambda f^{\wedge[\mathrm{int}\to\mathrm{real},\mathrm{bool}\to\mathrm{real}]}.\times((\pi_1^{\wedge}\, f) \mathbin{@} 17, (\pi_1^{\wedge}\, f) \mathbin{@} 23, (\pi_2^{\wedge}\, f) \mathbin{@} \textbf{true})) \\ & \textbf{in} \\ & \quad \times(\ (\pi_1^{\wedge}\, p) \mathbin{@} \bigwedge\big(\lambda x^{\mathrm{int}}.x, \lambda x^{\mathrm{bool}}.x\big), \\ & \qquad (\pi_2^{\wedge}\, p) \mathbin{@} \bigwedge\big(\lambda y^{\mathrm{int}}.3.141, \lambda y^{\mathrm{bool}}.3.141\big)\big)\end{array}$$

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 $\lambda^{\text{CIL}}_{\text{ul}}$:[8]

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

The two incompatible types returned by the body of g are merged into the union type $\bigvee[\rho_1, \rho_2]$. 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.[9] 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.

Sometimes it is desirable to customize the clauses of a $\textbf{case}^{\vee}$ to take advantage of type or flow differences between virtual variants that can reach the $\textbf{case}^{\vee}$. This type-based customization can be carried out by a splitting transformation that changes the $\textbf{case}^{\vee}$ to a $\textbf{case}^{+}$ and the corresponding occurrences of $\textbf{in}_i^{\vee}$ to $\textbf{in}_i^{+}$. The real variants resulting from this transformations carry run-time tags that are used to choose the appropriate clause code to execute.

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 encoded by injecting values of the same type into values of the same union type with different virtual tags. For instance, in the above example, $\times\left(\lambda x^{\text{int}}.x+1, w^{\text{int}}\right)$ and $\times\left(\lambda y^{\text{int}}.y*2, w^{\text{int}}+1\right)$ 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.

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 [DMTW97].

3.3 The Flow Typed Language λ^{CIL}

The flow typed language λ^{CIL} extends $\lambda^{\text{CIL}}_{\text{ul}}$ with flow label annotations on terms and types. Program flow information is encoded in two ways: (1) via flow label annotations that approximate

[8] In $\lambda^{\text{CIL}}_{\text{ul}}$ and 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.

[9] One way to phrase the duality between intersections and unions is to note that virtual tuples are polymorphic values while virtual cases are polymorphic continuations.

the flow of values from source expressions to sink expressions; and (2) via terms of intersection and union type. Each source term is annotated with a single source label and a set of sink labels that approximates the sink terms 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 approximates the source terms from which the values consumed by the sink may flow. 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.[10]

Here is a sample flow-annotated term:

$$
\begin{array}{rl}
M_e \equiv & \textbf{let } f^{\text{int} \xrightarrow[\{3,4\}]{\{1\}} \text{int}} = \lambda^1_{\{3,4\}} x^{\text{int}}.x * 2 \\
& \textbf{in } \textbf{let } g^{\text{int} \xrightarrow[\{4\}]{\{2\}} \text{int}} = \lambda^2_{\{4\}} y^{\text{int}}.y + a^{\text{int}} \\
& \quad \textbf{in } \times \Big(\textbf{coerce}\Big(\text{int} \xrightarrow[\{3,4\}]{\{1\}} \text{int}, \text{int} \xrightarrow[\{3\}]{\{1\}} \text{int}\Big) f \ @^{\{1\}}_3 \ 5, \\
& \qquad \Big(\textbf{if } b^{\text{bool}} \\
& \qquad \textbf{then coerce}\Big(\text{int} \xrightarrow[\{3,4\}]{\{1\}} \text{int}, \text{int} \xrightarrow[\{4\}]{\{1,2\}} \text{int}\Big) f \\
& \qquad \textbf{else coerce}\Big(\text{int} \xrightarrow[\{4\}]{\{2\}} \text{int}, \text{int} \xrightarrow[\{4\}]{\{1,2\}} \text{int}\Big) g\Big) \ @^{\{1,2\}}_4 \ 7\Big)
\end{array}
$$

For readability, only abstractions, applications, and arrow types have been annotated with explicit labels; other terms and types can be considered to be trivially labelled with a "don't care" label. The **coerce** terms are explicit subtyping coercions that are consistent with our strategy of encoding all aspects of a type derivation within the term structure. Coercions add source labels to and/or remove sink labels from a type.

Flow annotations summarize the results of a flow analysis on a term. The flow labels are sound with respect to the reduction rules of the calculus in the sense that in reductions that annihilate a source/sink pair, the source and sink labels on these terms must match exactly. Soundness follows from a subject reduction property on the calculus. See section 5.3.3 for details.

Of course, flow annotations are necessarily only conservative compile-time approximations of actual run-time flow. For example, it may be the case that no value produced by a particular source term can flow to a particular sink term whose label is in the sink set of the source. The trivial flow annotation, in which every term and type is labelled with the same "don't care" label is isomorphic to the unlabelled calculus. Often it is helpful to assume a *type/label consistency* (TLC) property in which the flow annotations are at least as refined as the types [DMTW97]; this corresponds to the notion of *type respecting* flow analysis in [JWW97].

We have used flow information in conjunction with the splitting transformations discussed above to manage the plumbing details associated with transformations that introduce multiple representations for a type [DMTW97]. An important stage of this framework is *flow separation*, which introduces intersection and union types and virtual tuples and variants to refine flow types and express finer-grained flow distinctions. Flow separation highlights the correspondence between flows and intersection and union types. We have also used flow types to construct DJ graphs [SGL96] for detecting loops in higher-order programs.

[10] 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. The calculus could be extended to support other labelled values.

4 Design Issues

Below, we discuss the rationale behind various decisions made in the design of λ^{CIL}.

4.1 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 facilitate the determination of the typing (i.e., type derivation) for a term, λ^{CIL} annotates variables and variants 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.

4.2 Finitary Polymorphic Types

A central goal of our work is to encode precise information obtained from program analysis into the type systems of typed intermediate languages. 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. For example, type polymorphism for functions is usually encoded using universal types, which provide no information about the types of arguments at which such functions may be called. Dually, abstract data types are typically encoded by existential types, which provide no 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 usage 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 version of the infinitary 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. A key feature of these types is that they can encode data flow: intersection types represent the possible destinations of a value while union types represent its possible sources.

4.3 Intersection and Union Type Notation

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 5.3, 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 [Rey96b] 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 $(\sigma_1 \to \tau) \wedge \cdots \wedge (\sigma_n \to \tau)$. Unfortunately, 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 gives a more general approach that uses a special term-level construct to bind a type variable to some set of types [Pie91]. For example, using this method the term $(\lambda x.\lambda y.x)$ can be annotated as $(\textbf{for}\ \alpha \in \{\sigma, \tau\}.\lambda x{:}\,\alpha.\lambda y{:}\,\alpha.x)$, which has the type $(\sigma \to (\sigma \to \sigma)) \wedge (\tau \to (\tau \to \tau))$. However, this method is insufficient to represent some typings, e.g., giving the term $(\lambda x.\lambda y.\lambda z.(xy, xz))$ the type $(((\alpha \to \alpha) \wedge (\beta \to \beta)) \to \alpha \to \beta \to (\alpha \times \beta)) \wedge ((\gamma \to \gamma) \to \gamma \to \gamma \to (\gamma \times \gamma))$. Another problem with Pierce's method is that subterms of a well typed term are not necessarily well typed out of their context. For instance, in $(\textbf{for}\ \alpha \in \{\text{int} \to \text{int}\}.\lambda f{:}\,\alpha.f5)$, the term $\lambda f{:}\,\alpha.f5$ is not well typed since it is not known whether α is $\text{int} \to \text{int}$.

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, which we now prefix with "$\times$". 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 an explicit projection $\pi_i^{\wedge}$ to extract a component out of a value of intersection type:

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

An implication of this approach is that constructors for intersection types and virtual tuples are neither associative nor commutative.

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:

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

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 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. Similarly, λ^{CIL} faciliates transforming virtual variants to real variants.

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 5.3 introduces *parallel contexts* to formalize this notion of parallel computation step.

4.4 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.

4.5 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:

$$\sigma' \equiv \quad \mathrm{int} \xrightarrow[\{3\}]{\{1\}} \mathrm{int} \quad \leq \quad \mathrm{int} \xrightarrow[\{3\}]{\{1,2\}} \mathrm{int} \quad \equiv \sigma$$

$$\tau \equiv \mathrm{bool} \xrightarrow[\{5,6\}]{\{4\}} \mathrm{bool} \leq \mathrm{bool} \xrightarrow[\{5\}]{\{4\}} \mathrm{bool} \equiv \tau'$$

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

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

It would be natural to make the required argument and result coercions explicit as follows:

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

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 [WS94, DMTW97]. But in the above translation, abstraction 7 is open (it contains the free variable g); this thwarts the attempted customization.

4.6 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 $\lambda^{\mathrm{CIL}}_{\mathrm{ut}}$ term:

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

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

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

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

$$\hat{M}_3 \equiv (\pi_1^{\times} \times((\lambda e.e+1),5)) \mathbin{@} (\pi_2^{\times}\ (\textbf{if}\ b\ \textbf{then}\ \times((\lambda e.e+1),5)\ \textbf{else}\ \times((\lambda e.2),\times(()))))$$

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:

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

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

$$\begin{aligned}\hat{M} &\equiv (\pi_1^{\times}\ x) \mathbin{@} (\pi_2^{\times}\ x)\\ \hat{N} &\equiv f\\ \sigma &\equiv \times[\text{int} \to \text{int}, \text{int}]\\ \tau &\equiv \times[\times[\,] \to \text{int}, \times[\,]]\\ \rho &\equiv \text{int}\end{aligned}$$

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 substitition 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 beta 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[11] and address the sharing problem by stipulating call-by-value reduction. At the untyped level, the base values are constants and abstractions and the set of values is closed under

[11] The rule ($\vee$ elim b) is the $\vee$-elimination rule for the implicitly typed language $\lambda_{\text{i}}^{\text{CIL}}$. The corresponding $\vee$-elimination rule for the explicitly typed language λ^{CIL} involves the $\textbf{case}^{\vee}$ construct. See sections 5.3 and 5.5 for details.

tuple and variant formation. Values at the typed level are similar, but also include virtual tuples and variants.[12] 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 $\lambda_{\text{i}}^{\text{CIL}}$ (see section 5.5) this implies that when ($\vee$ elim b) is used to type a β-value redex, it occurs in the following pattern:

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

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

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

So in $\lambda_{\text{i}}^{\text{CIL}}$, 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

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

and thence to

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

Note that in the explicitly typed language λ^{CIL}, the assumption 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., [Plo75]), they cause trouble in λ^{CIL} because they can invalidate the equivalence between the two typing derivation patterns shown above. As a concrete example of this trouble, consider the following terms:

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

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.

Thus, 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 $\lambda_{\text{i}}^{\text{CIL}}$ inherits subject reduction from the explicitly typed language λ^{CIL}.

[12] Values in the typed language are more precisely defined as terms that type erase to values in the untyped language. See section 5.3 for details.

$\lambda_{\mathrm{i}}^{\mathrm{CIL}}$ appears to be the first implicitly typed calculus with union types that has the subject reduction property. The loss of subject reduction in the presence of union types and unrestricted reduction has been noted before. Barbanera, Dezani-Ciancaglini, and de'Liguoro [BDCd95] 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. Inspired in part by an earlier version of the work reported here, Palsberg and Pavlopoulou [PP98] 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.

5 Formal Language Definition

The formal definition of the language λ^{CIL} proceeds in several steps. Section 5.2 defines the untyped language $\lambda_{\mathrm{ut}}^{\mathrm{CIL}}$ by adding tuples, variants, and recursion to the pure λ-calculus and providing call-by-value reduction rules. Section 5.3 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 $\lambda_{\mathrm{ut}}^{\mathrm{CIL}}$. Then, reduction rules are defined that provide the expected correspondence between λ^{CIL} and $\lambda_{\mathrm{ut}}^{\mathrm{CIL}}$. Section 5.4 defines the unlabelled language $\lambda_{\mathrm{ul}}^{\mathrm{CIL}}$ as a subset of λ^{CIL} that is closed under reduction and shares all the properties of λ^{CIL}. Section 5.5 observes that an implicitly typed language $\lambda_{\mathrm{i}}^{\mathrm{CIL}}$ is automatically obtained by taking typing derivations of $\lambda_{\mathrm{ul}}^{\mathrm{CIL}}$ and erasing types from the terms in these derivations. $\lambda_{\mathrm{i}}^{\mathrm{CIL}}$ is the first implicitly typed calculus with intersection and union types that has the subject reduction property.

5.1 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. 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; $X\trianglelefteq Y$ 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.)[Bar84]. A *simple* n.o.r. R is a pair $(\rightsquigarrow_R, \mathbf{C}_R)$ of a redex/contractum relation $\rightsquigarrow_R$ and a set of reduction contexts $\mathbf{C}_R$.[13] The statement $M\rightsquigarrow_R N$ means M is an *R-redex* and N is the *R-contractum* of M. For a simple n.o.r., $M\longrightarrow_R N$ means M is transformed into N by contracting R-redexes in positions in M specified by an *R-reduction context*, i.e., there are a context $C\in\mathbf{C}_R$ with k holes and terms M_i and N_i for

[13] Barendregt's definition [Bar84, sec. 3.1.1] identifies R with $\rightsquigarrow_R$ and requires $\mathbf{C}_R$ to always be the set of all single-hole contexts. Barendregt's formulation yields the *compatible* closure of $\rightsquigarrow_R$.

$i \in \{1, \ldots, k\}$ such that $M \equiv C[M_1, \ldots, M_k]$ and $N \equiv C[N_1, \ldots, N_k]$ and $M_i \rightsquigarrow_R N_i$ 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. The symbol $\longrightarrow\!\!\!\!\!\rightarrow_R$ 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 $M \longrightarrow\!\!\!\!\!\rightarrow_R N$ and R-nf(N).

5.2 Untyped Language $\lambda_{\text{ut}}^{\text{CIL}}$

Untyped Syntax

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

Untyped Reduction $(\rightsquigarrow_{\hat{c}}, \mathbf{C}_{\hat{c}})$

$$
\begin{array}{lll}
(\lambda x.\hat{M}) \mathbin{@} \hat{V} & \rightsquigarrow_{\hat{c}}\ \hat{M}[x{:=}\hat{V}] & \\
\pi_i^{\times} \times\left(\hat{V}_1, \ldots, \hat{V}_n\right) & \rightsquigarrow_{\hat{c}}\ \hat{V}_i & \text{if } 1 \le i \le n \\
\mathbf{case}^{+}\ (\mathbf{in}_i^{+}\hat{V})\ \mathbf{bind}\ x\ \mathbf{in}\ \hat{M}_1, \ldots, \hat{M}_n & \rightsquigarrow_{\hat{c}}\ (\lambda x.\hat{M}_i) \mathbin{@} \hat{V} & \text{if } 1 \le i \le n \\
\mu x.\hat{M} & \rightsquigarrow_{\hat{c}}\ \hat{M}[x{:=}(\mu x.\hat{M})] &
\end{array}
$$

Reduction contexts: $\mathbf{C}_{\hat{c}} = \{\, \hat{C} \mid \hat{C} \in \textbf{UntContext}$ and $\hat{C}$ has exactly one hole $\}$

Figure 1: Untyped language $\lambda_{\text{ut}}^{\text{CIL}}$.

5.2.1 Syntax and Semantics of $\lambda_{\text{ut}}^{\text{CIL}}$

Figure 1 shows the syntax and semantics of the untyped language $\lambda_{\text{ut}}^{\text{CIL}}$. The syntactic categories **UntContext**, **UntTerm**, and **UntValue** are respectively the untyped and unlabelled *contexts*, *terms*, and *values*.

$\lambda_{\text{ut}}^{\text{CIL}}$ includes constants, but no primitive operators on constants. The reason for this is that values at ground type are necessary for some formal statements, but the presentation is simpler without primitive operators on these values.

5.2.2 Confluence of $\lambda_{\text{ut}}^{\text{CIL}}$

We will prove confluence of $\lambda_{\text{ut}}^{\text{CIL}}$ 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.[14]

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

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

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

Now we give reduction rules for the CRS to simulate reduction in $\lambda_{\mathrm{ut}}^{\mathrm{CIL}}$. 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 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 $R_{\hat{c}}$ to simulate the reduction rules of $\lambda_{\mathrm{ut}}^{\mathrm{CIL}}$.

$$R_v = \left\{ \begin{array}{lll}
\mathbf{notval}(\lambda(Z)) & \to \mathbf{val}(\lambda(Z)) & \\
\mathbf{notval}(c) & \to \mathbf{val}(c) & \\
\mathbf{notval}(\times_n(\mathbf{val}(Z_1), \ldots, \mathbf{val}(Z_n))) & \to \mathbf{val}(\times_n(\mathbf{val}(Z_1), \ldots, \mathbf{val}(Z_n))) & \mid n \in \mathbb{N} \\
\mathbf{notval}(\mathbf{in}_i^{+}(\mathbf{val}(Z))) & \to \mathbf{val}(\mathbf{in}_i^{+}(\mathbf{val}(Z))) & \mid i \in \mathbb{N}
\end{array} \right\}$$

$$R_{\hat{c}} = \left\{ \begin{array}{lll}
\mathbf{notval}(@(\mathbf{val}(\lambda([x]Z(x))), \mathbf{val}(Z'))) & \to Z(Z') & \\
\mathbf{notval}(\pi_i^{\times}(\mathbf{val}(\times_n(Z_1, \ldots, Z_n)))) & \to Z_i & \mid 1 \le i \le n \in \mathbb{N} \\
\mathbf{notval}(\mathbf{case}_n^{+}(\mathbf{val}(\mathbf{in}_i^{+}(\mathbf{val}(Z))), [x]Z_1(x), \ldots, [x]Z_n(x))) & & \\
\qquad & \to \mathbf{notval}(@(\mathbf{val}(\lambda([x]Z_i(x))), \mathbf{val}(Z))) & \mid 1 \le i \le n \in \mathbb{N} \\
\mathbf{notval}(\mu([x]Z(x))) & \to Z(\mu([x]Z(x))) &
\end{array} \right\}$$

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

REMARK 5.1. Note that the CRS Σ_{ut} meets the *structure-preserving* criteria of [BR96], since every argument of a RHS metavariable occurs as a subterm of the corresponding LHS. Thus, the techniques of [BR96] can easily give an explicit-substitution version of Σ_{ut}. In turn, from this it is possible to derive an abstract machine implementation. □

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

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

Lemma 5.2. *The CRS Σ_{ut} is regular, i.e., its rules are left-linear and non-ambiguous.* □

Proof. Simple checking reveals that the rules are left-linear. To see that the rules are non-ambiguous, 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 5.3. *Reduction in Σ_{ut} is confluent.* □

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

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

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

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

Lemma 5.4. *If $\mathcal{C}_{\text{ut}}(\hat{M}) \twoheadrightarrow_{\Sigma_{\text{ut}}} u$, then $\mathcal{C}_{\text{ut}}^{-1}(u)$ is defined.* □

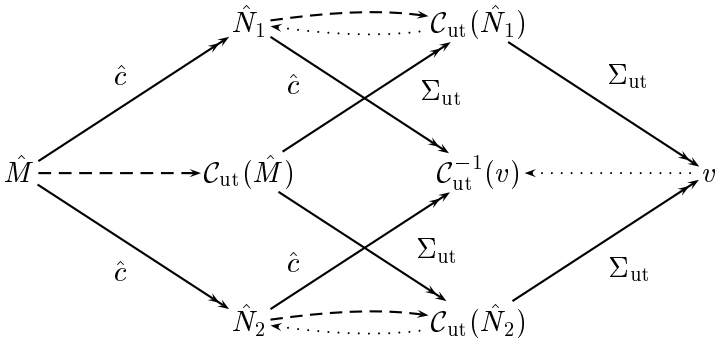
Lemma 5.5. *Both of the following statements hold:*

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

□

Theorem 5.6 (Confluence of Untyped Reduction). *If $\hat{M} \twoheadrightarrow_{\hat{c}} \hat{N}_1$ and $\hat{M} \twoheadrightarrow_{\hat{c}} \hat{N}_2$, then there exists $\hat{M}'$ such that $\hat{N}_1 \twoheadrightarrow_{\hat{c}} \hat{M}'$ and $\hat{N}_2 \twoheadrightarrow_{\hat{c}} \hat{M}'$.* □

Proof. By constructing this diagram:



□

5.3 Explicitly Typed Language λ^{CIL}

5.3.1 Type/Flow-Annotated Term Syntax

Figure 2 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.[15] 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}.

The requirement of non-empty label sets is imposed by certain representation transformations in the compiler framework based on λ^{CIL}[DMTW97]. This requirement can easily be satisfied for a top level $\lambda^{l}_{\psi} x^{\tau}.C$ context (or for dead code) by using distinguished "no source" and "no sink" labels in ψ or ϕ.

The syntax $(\textbf{coerce}\,(\sigma,\tau)\,M)$ explicitly records at the term level the use of the subtyping rule in figure 2 to coerce the type of M from σ to τ. In this presentation, there is only one subtyping rule: it 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.), the subtyping rules would be extended accordingly.

The recursive binding construct "μ" is used to build recursive types. Our definition of type equality on recursive types is standard [Bar92]. We do not distinguish between equal recursive types (i.e., types whose infinite unfoldings are identical) in any context, so we have no rules for folding or unfolding recursive types. The syntax forbids free type variables from occurring in typing derivations and terms.

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 facilitates transformations that handle this pattern differently from the purely structural translation of an abstraction within an application.

The type erasure $|C|$ of a type-annotated context C (defined in figure 2) is the corresponding untyped and unlabelled context. The fact that virutal 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 4.6. Some contexts do not have a type erasure, i.e., those containing virtual tuples like $\bigwedge(C_1,\ldots,C_n)$ or virtual case expressions like

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

where the type erasures of $C_1, \ldots, C_n$ 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 5.7 (Properties of Sub-Contexts).

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

Syntax Shared between Types and Terms

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

Types

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

Type Equality

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

Type-Annotated Contexts

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

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

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

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

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

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

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

Type-Annotated Terms, Values, Parallel Contexts

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

Syntactic Sugar for Examples

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

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

Figure 2: Syntax of explicitly typed language λ^{CIL}.

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 5.8 (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. $C[C_1] \equiv C[C_2] \iff C_1 \equiv C_2$.

□

Lemma 5.9 ((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* $|Cp[C_1, \ldots, C_n]| \equiv |Cp|\,[|C_1|]$.

□

Lemma 5.10. *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.* □

5.3.2 Typing Derivations and Well Typed Terms

Figure 3 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 sequence of judgements such that:

1. each judgement is obtained from the previous ones by the typing rules of X;

2. if a typing rule has k premises, then, for $1 \leq i < k$, the entire subderivation for the ith premise must precede any of the subderivation for the $i+1$th premise.[16]

[16] This condition guarantees that derivations are effectively trees. Without this condition, there could be many different derivation sequences that were topological sorts of the same derivation tree, invalidating the uniqueness claim of theorem 5.11.

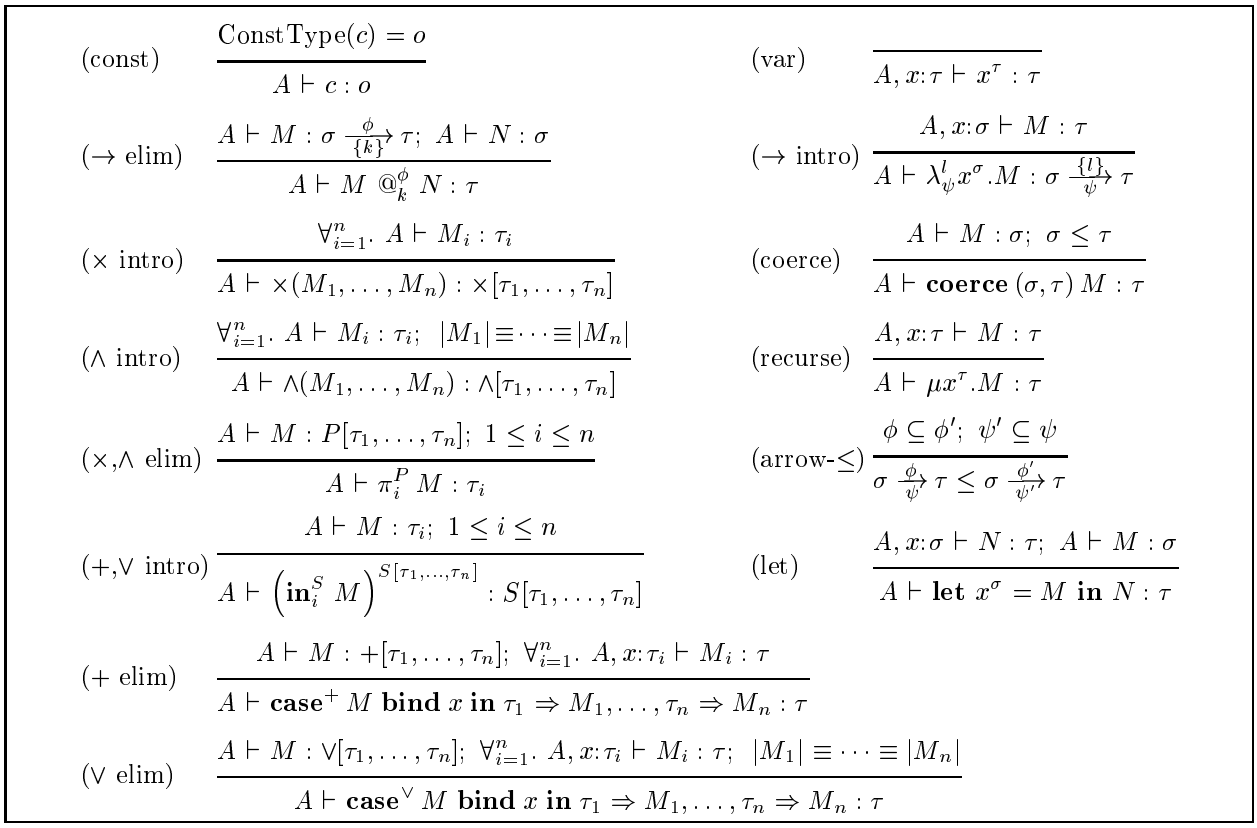


Figure 3: Typing rules of explicitly typed language λ^{CIL}.

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 $A \vdash_{\lambda^{\text{CIL}}} M : \tau$ means there exists some $\mathcal{D}$ such that $A \vdash_{\lambda^{\text{CIL}}} M : \tau$ 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 $\vee$).

We will show that typing derivations and well typed terms are isomorphic, using the functions defined in figure 4. 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 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 $\text{Typ}(M)$ is not explicitly specified, then it is undefined.

Theorem 5.11 (Uniqueness of Typings in λ^{CIL}).

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

5.3.3 Reduction on Explicitly Typed Terms

The call-by-value reduction rules for the typed language λ^{CIL} are in figure 5. The main notion of reduction, r-reduction, is divided into three steps: simplifying type 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 5.13), 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 5.12. In the t-reduction rules in figure 5, 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

Addition and Subtraction of Type Environments

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

Environment Inference Function

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

$$\begin{array}{ll}
\mathrm{Env}(c) & = \varnothing \\
\mathrm{Env}(x^\tau) & = \{x : \tau\} \\
\mathrm{Env}(\lambda^l_\psi x^\tau.M) & = \mathrm{Env}(M) \ominus \{x : \tau\} \\
\mathrm{Env}(M\ @^\phi_k\ N) & = \mathrm{Env}(M) \oplus \mathrm{Env}(N) \\
\mathrm{Env}(\mathbf{coerce}\ (\sigma,\tau)\ M) & = \mathrm{Env}(M) \\
\mathrm{Env}(\mu x^\tau.M) & = \mathrm{Env}(M) \ominus \{x : \tau\} \\
\mathrm{Env}(\mathbf{let}\ x^\tau = M\ \mathbf{in}\ N) & = \mathrm{Env}(M) \oplus (\mathrm{Env}(N) \ominus \{x : \tau\}) \\
\mathrm{Env}(P(M_1, \ldots, M_n)) & = \mathrm{Env}(M_1) \oplus \cdots \oplus \mathrm{Env}(M_n) \\
\mathrm{Env}(\pi^P_i\ M) & = \mathrm{Env}(M) \\
\mathrm{Env}(\left(\mathbf{in}^S_i\ M\right)^\tau) & = \mathrm{Env}(M) \\
\mathrm{Env}(\mathbf{case}^S\ M\ \mathbf{bind}\ x\ \mathbf{in}\ \tau_1 \Rightarrow M_1, \ldots, \tau_n \Rightarrow M_n) & \\
\qquad = \mathrm{Env}(M) \oplus (\mathrm{Env}(M_1) \ominus \{x : \tau_1\}) \oplus \ldots \oplus (\mathrm{Env}(M_n) \ominus \{x : \tau_n\}) &
\end{array}$$

Type Inference Function

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

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

Figure 4: Definitions of the Env and Typ functions.

Main Notion of Reduction for Type-Annotated Terms

$$M \longrightarrow_r N \quad \text{iff} \quad \exists M', N'.\ (M \xrightarrow{\text{nf}}_t M' \longrightarrow_c N' \xrightarrow{\text{nf}}_t N)$$

Computation Reduction $(\rightsquigarrow_c, \mathbf{C}_c)$

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

Reduction contexts: $\mathbf{C}_c = \mathbf{ParallelContext}$

Type-Annotation-Simplification Reduction $(\rightsquigarrow_t, \mathbf{C}_t)$

$(\lambda^l_\psi x^\tau.N)\ @^\phi_k\ M$	$\rightsquigarrow_t \mathbf{let}\ x^\tau = M\ \mathbf{in}\ N$	
$\pi_i^\wedge \wedge(M_1, \ldots, M_n)$	$\rightsquigarrow_t M_i$	if $1 \le i \le n$
$\mathbf{case}^\vee \left(\mathbf{in}_i^\vee\ N\right)^\tau\ \mathbf{bind}\ x\ \mathbf{in}\ \tau_1 \Rightarrow M_1, \ldots, \tau_n \Rightarrow M_n$	$\rightsquigarrow_t \mathbf{let}\ x^{\tau_i} = N\ \mathbf{in}\ M_i$	if $1 \le i \le n$
$(\mathbf{coerce}\ (\sigma, \tau)\ (\lambda^l_\psi x^\rho.M))\ @^\phi_k\ N$	$\rightsquigarrow_t \mathbf{let}\ x^\rho = N\ \mathbf{in}\ M$	
$\mathbf{coerce}\ (\sigma_1, \tau)\ \ \mathbf{coerce}\ (\rho, \sigma_2)\ M$	$\rightsquigarrow_t \mathbf{coerce}\ (\rho, \tau)\ M$	

Reduction contexts: $\mathbf{C}_t = \{\, C \mid C \in \mathbf{Context} \text{ and } C \text{ has exactly one hole} \,\}$

Figure 5: Reduction rules of explicitly typed language λ^{CIL}.

typing rules in figure 3. 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 5.13. *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 5.14. 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 $\longrightarrow_r$. These results only require that any term can be reduced to t-normal-form. □

Lemma 5.15.

1. *If $M \rightsquigarrow_c C$, then $C \in \mathbf{Term}$ and $|M| \rightsquigarrow_{\hat{c}} |C|$.*
2. *If $M \rightsquigarrow_t C$, then $C \in \mathbf{Term}$ and $|M| \equiv |C|$.*

□

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

Lemma 5.16 (Redex/Contractum Relations Are Functions). *For each simple n.o.r. $R \in \{\hat{c}, c, t\}$, for any syntactic entity X, there is at most one Y such that $X \rightsquigarrow_R Y$.* □

Proof. By inspection of the reduction rules. □

Lemma 5.17. *The set* **Term** *is closed under c-reduction and t-reduction. Also, each c-reduction step corresponds to a $\hat{c}$-reduction step on the type erasure while each t-reduction step preserves the type erasure. More specifically,*

1. *If $M \longrightarrow_c C$, then $C \in$* **Term** *and $|M| \longrightarrow_{\hat{c}} |C|$.*
2. *If $M \longrightarrow_t C$, then $C \in$* **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 $M_i \rightsquigarrow_c C_i$ for $1 \leq i \leq n$. By lemma 5.9, we know that $|M_1| \equiv \cdots \equiv |M_n|$ and $|M| \equiv |Cp|\,[|M_1|]$. By lemma 5.15, we know that $C_1, \ldots, C_n \in$ **Term** and $|M_i| \rightsquigarrow_{\hat{c}} |C_i|$ for $1 \leq i \leq n$. By lemma 5.16 we know that $|C_1| \equiv \cdots \equiv |C_n|$. By lemma 5.9, we know that $|C|$ is defined, implying $C \in$ **Term**, and that $|C| \equiv |Cp|\,[|C_1|]$, implying that $|M| \longrightarrow_{\hat{c}} |C|$.
2. By definition of t-reduction, we know that $M \equiv C'[M']$ and $C \equiv C'[C'']$ where $M' \rightsquigarrow_t C''$. By lemma 5.15, we know that $C'' \in$ **Term** and $|M'| \equiv |C''|$. By lemma 5.10, we know that $|M| \equiv |C'[M']| \equiv |C'[C'']| \equiv |C|$ implying that $C \in$ **Term**.

□

Lemma 5.18. *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 5.19. *If $M \rightsquigarrow_R N$ for $R \in \{c, t\}$, 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 5.18 for the reduction of **let**. For 3, using 1 and 2 together with theorem 5.11. □

Lemma 5.20 (Subject c/t-Reduction).

1. *If $M \longrightarrow_c N$ and $A \vdash_{\lambda^{\mathrm{CIL}}} M : \tau$, then $A \vdash_{\lambda^{\mathrm{CIL}}} N : \tau$.*

2. *If* $M \longrightarrow_t N$ *and* $A \vdash_{\lambda^{\text{CIL}}} M : \tau$, *then* $A \vdash_{\lambda^{\text{CIL}}} N : \tau$.

□

Proof.

1. We know that $M \equiv Cp[M_1, \ldots, M_n]$ and $N \equiv Cp[N_1, \ldots, N_n]$ where $M_i \rightsquigarrow_c N_i$ for $1 \leq i \leq n$. Consider the typing derivation $\mathcal{D}$ which proves $A \vdash_{\lambda^{\text{CIL}}} M : \tau$. For $1 \leq i \leq n$ the typing derivation $\mathcal{D}$ has subderivation $\mathcal{D}_i$ proving $A_i \vdash_{\lambda^{\text{CIL}}} M_i : \tau_i$ for some A_i and τ_i. By lemma 5.19, for $1 \leq i \leq n$ there is a derivation $\mathcal{D}'_i$ proving $A_i \vdash_{\lambda^{\text{CIL}}} N_i : \tau_i$. Consider the derivation $\mathcal{D}'$ formed from $\mathcal{D}$ by replacing $\mathcal{D}_i$ by $\mathcal{D}'_i$ for $1 \leq i \leq n$. 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 5.17), we know that $\mathcal{D}'$ is a valid derivation, giving the desired result.

2. Similar reasoning to the previous case, only simpler.

□

Theorem 5.21 (Subject r-Reduction for λ^{CIL}). *If* $M \longrightarrow_r N$ *and* $A \vdash_{\lambda^{\text{CIL}}} M : \tau$, *then* $A \vdash_{\lambda^{\text{CIL}}} N : \tau$. □

Proof. The claim follows immediately from lemma 5.20 and the definition of r-reduction as the composition of c-reduction and t-reduction. □

Definition 5.22 (Erasable Form). A term M is an *erasable form* if it has the form $\bigwedge(M_1, \ldots, M_n)$, $\pi_i^{\wedge} M'$, $\mathbf{coerce}\,(\sigma, \tau)\, M'$, or $\left(\mathbf{in}_i^{\vee}\ M'\right)^{\top}$. □

Lemma 5.23. *If* $t\text{-nf}(M)$ *and* $|M| \equiv \hat{C}[\hat{N}]$, *then* $M \equiv Cp[N_1, \ldots, N_n]$ *where* $|Cp| \equiv \hat{C}$, $|N_1| \equiv \hat{N}$, *and* N_i *is not an erasable form for* $1 \leq i \leq n$. □

Lemma 5.24. *If* M *is well typed,* $t\text{-nf}(M)$, M *is not an erasable form, and* $|M|$ *is a* $\hat{c}$*-redex, then* M *is a* c*-redex.* □

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

Lemma 5.25. *If* $t\text{-nf}(M)$, M *is well typed, and* $|M| \longrightarrow_{\hat{c}} \hat{N}$, *then there is a term* N *such that* $M \longrightarrow_c N$ *and* $|N| \equiv \hat{N}$. □

Proof. Because $|M| \longrightarrow_{\hat{c}} \hat{N}$, we know that $|M| \equiv \hat{C}[\hat{M}']$ and $\hat{N} \equiv \hat{C}[\hat{N}']$ where $\hat{M}' \rightsquigarrow_{\hat{c}} \hat{N}'$. By lemmas 5.23 and 5.24, we know that $M \equiv Cp[M_1, \ldots, M_n]$ where $|Cp| \equiv \hat{C}$, $|M_1| \equiv \cdots \equiv |M_n| \equiv \hat{M}'$, and $M_i \rightsquigarrow_c N_i$ for $1 \leq i \leq n$. Thus, $M \longrightarrow_c N$ where $N \equiv Cp[N_1, \ldots, N_n]$. All that remains is to show that $|N| \equiv \hat{N}$.

By lemma 5.17, $|M| \longrightarrow_{\hat{c}} |N|$. By lemma 5.9 and the above reasoning we know that $\hat{C}[\hat{M}'] \longrightarrow_{\hat{c}} \hat{C}[|N_1|]$. Thus, $\hat{M}' \rightsquigarrow_{\hat{c}} |N_1|$. By lemma 5.16, 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 5.26 (Typed/Untyped Reduction Correspondence).

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

□

Proof.

1. This claim follows immediately from lemma 5.17 and the definition of r-reduction as the composition of c-reduction and t-reduction.
2. By lemma 5.13, $M \xrightarrow{\text{nf}}_t M'$. By lemma 5.20, M' is well typed. By lemma 5.17, $|M'| \equiv |M|$, implying that $|M'| \longrightarrow_{\hat{c}} \hat{N}$. By lemma 5.25, $M' \longrightarrow_c N'$ where $|N'| \equiv \hat{N}$. By lemmas 5.13 and 5.17, $N' \xrightarrow{\text{nf}}_t N$ where $|N'| \equiv |N| \equiv \hat{N}$. By definition of r-reduction, $M \longrightarrow_r N$, showing the desired result.

□

Theorem 5.27 (Confluence Modulo Type Erasure of Typed Reduction). *If* M_1 *and* M_2 *are well typed,* $|M_1| \equiv |M_2|$, $M_1 \longrightarrow\!\!\!\!\rightarrow_r N_1$, *and* $M_2 \longrightarrow\!\!\!\!\rightarrow_r N_2$, *then there exist* M_1' *and* M_2' *such that* $|M_1'| \equiv |M_2'|$, $N_1 \longrightarrow\!\!\!\!\rightarrow_r M_1'$ *and* $N_2 \longrightarrow\!\!\!\!\rightarrow_r M_2'$. □

Proof. By theorem 5.26, $|M_1| \longrightarrow\!\!\!\!\rightarrow_{\hat{c}} |N_1|$ and $|M_2| \longrightarrow\!\!\!\!\rightarrow_{\hat{c}} |N_2|$. By theorem 5.6, there exists $\hat{N}$ such that $|N_1| \longrightarrow\!\!\!\!\rightarrow_{\hat{c}} \hat{N}$ and $|N_2| \longrightarrow\!\!\!\!\rightarrow_{\hat{c}} \hat{N}$. By theorem 5.26, there exist terms M_1 and M_2 such that $|M_1| \equiv |M_2| \equiv \hat{N}$ and $N_1 \longrightarrow\!\!\!\!\rightarrow_r M_1$ and $N_2 \longrightarrow\!\!\!\!\rightarrow_r M_2$. □

REMARK 5.28. 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. □

5.4 Unlabelled Language $\lambda^{\text{CIL}}_{\text{ul}}$

The syntax of the unlabelled language $\lambda^{\text{CIL}}_{\text{ul}}$ is defined in figure 6. The unlabelled language is defined as the subset of λ^{CIL} (1) that has no instances of **coerce** and (2) in which all source and sink labels have been collapsed to the distinguished label $\bullet$. This subset is formally defined via a *label erasure* function on types and terms that erases **coerce**s and maps all labels to $\bullet$. Since no confusion can arise, $\bullet$ is typically omitted when writing $\lambda^{\text{CIL}}_{\text{ul}}$ terms and types, as in section 3.2.

The semantics of $\lambda^{\text{CIL}}_{\text{ul}}$ is inherited from λ^{CIL}. It is easy to show that $\lambda^{\text{CIL}}_{\text{ul}}$ is closed under $\longrightarrow_X$, where $X \in \{t, c, r\}$, so that the restrictions of these relations to $\lambda^{\text{CIL}}_{\text{ul}}$ (written $\longrightarrow_{\tilde{t}}$, $\longrightarrow_{\tilde{c}}$, and $\longrightarrow_{\tilde{r}}$) are well defined.

The typing rules of $\lambda^{\text{CIL}}_{\text{ul}}$ are also inherited from λ^{CIL}. Label erasure extends to type environments: if A is a type environment $\{x_1 : \tau_1, \ldots, x_n : \tau_n\}$, then $\langle A \rangle = \{x_1 : \langle \tau_1 \rangle, \ldots, x_n : \langle \tau_n \rangle\}$, The typing rules of $\lambda^{\text{CIL}}_{\text{ul}}$ are the rules of figure 3 (except for the **coerce** rule) modified by replacing every judgement $A \vdash_{\lambda^{\text{CIL}}} M : \tau$ mentioned in a rule by $\langle A \rangle \vdash_{\lambda^{\text{CIL}}_{\text{ul}}} \langle M \rangle : \langle \tau \rangle$.

Since $\lambda^{\text{CIL}}_{\text{ul}}$ is a subset of λ^{CIL}, all definitions on λ^{CIL} (such as type erasure) carry over to $\lambda^{\text{CIL}}_{\text{ul}}$. Moreover, since $\lambda^{\text{CIL}}_{\text{ul}}$ is closed under the reduction relations of λ^{CIL} and it respects the typing rules

Label Erasure on Types $\langle\cdot\rangle : \mathbf{Type} \to \mathbf{Type}$

$$
\begin{array}{lll}
\langle o \rangle & \equiv o & \langle \alpha \rangle \equiv \alpha \\
\left\langle \upsilon_1 \xrightarrow[\psi]{\phi} \upsilon_2 \right\rangle & \equiv \langle \upsilon_1 \rangle \xrightarrow[\{\bullet\}]{\{\bullet\}} \langle \upsilon_2 \rangle & \langle Q[\upsilon_1, \ldots, \upsilon_n] \rangle \equiv Q[\langle \upsilon_1 \rangle, \ldots, \langle \upsilon_n \rangle] \\
\langle \mu\alpha.\xi \rangle & \equiv \mu\alpha.\, \langle \xi \rangle &
\end{array}
$$

Label Erasure on Contexts $\langle\cdot\rangle : \mathbf{Context} \to \mathbf{Context}$

$$
\begin{array}{lll}
\langle \Box \rangle & \equiv \Box & \langle c \rangle \equiv c \\
\langle x^\tau \rangle & \equiv x^{\langle \tau \rangle} & \langle \mu x^\tau.C \rangle \equiv \mu x^{\langle \tau \rangle}.\langle C \rangle \\
\left\langle \lambda^l_\psi x^\tau.C \right\rangle & \equiv \lambda^\bullet_{\{\bullet\}} x^{\langle \tau \rangle}.\langle C \rangle & \left\langle C_1 \ @^\phi_k \ C_2 \right\rangle \equiv \langle C_1 \rangle \ @^{\{\bullet\}}_\bullet \ \langle C_2 \rangle \\
\langle \mathbf{coerce}\ (\sigma, \tau)\ C \rangle & \equiv \langle C \rangle & \langle \mathbf{let}\ x^\tau = C_1\ \mathbf{in}\ C_2 \rangle \equiv \mathbf{let}\ x^{\langle \tau \rangle} = \langle C_1 \rangle\ \mathbf{in}\ \langle C_2 \rangle \\
\langle P(C_1, \ldots, C_n) \rangle & \equiv P(\langle C_1 \rangle, \ldots, \langle C_n \rangle) & \langle \pi^P_i\ C \rangle \equiv \pi^P_i\ \langle C \rangle \\
\left\langle \left(\mathbf{in}^S_i\ C\right)^\tau \right\rangle & \equiv \left(\mathbf{in}^S_i\ \langle C \rangle\right)^{\langle \tau \rangle} &
\end{array}
$$

$$
\left\langle \mathbf{case}^S\ C\ \mathbf{bind}\ x\ \mathbf{in}\ \tau_1 \Rightarrow C_1, \ldots, \tau_n \Rightarrow C_n \right\rangle \equiv \mathbf{case}^S\ \langle C \rangle\ \mathbf{bind}\ x\ \mathbf{in}\ \langle \tau_1 \rangle \Rightarrow \langle C_1 \rangle, \ldots, \langle \tau_n \rangle \Rightarrow \langle C_n \rangle
$$

Syntax of $\lambda^{\mathrm{CIL}}_{\mathrm{ul}}$

$$
\begin{array}{rl}
\tilde{\rho}, \tilde{\sigma}, \tilde{\tau} \in \mathbf{UnlType} & = \{\, \langle \tau \rangle \mid \tau \in \mathbf{Type} \,\} \\
\tilde{C} \in \mathbf{UnlContext} & = \{\, \langle C \rangle \mid C \in \mathbf{Context} \,\} \\
\tilde{M}, \tilde{N} \in \mathbf{UnlTerm} & = \{\, \langle M \rangle \mid M \in \mathbf{Term} \,\} \\
\tilde{V} \in \mathbf{UnlValue} & = \{\, \langle V \rangle \mid V \in \mathbf{Value} \,\} \\
\tilde{Cp} \in \mathbf{UnlParallelContext} & = \{\, \langle Cp \rangle \mid Cp \in \mathbf{ParallelContext} \,\}
\end{array}
$$

Figure 6: Definition of the unlabelled language $\lambda^{\mathrm{CIL}}_{\mathrm{ul}}$.

of λ^{CIL}, all the results shown for λ^{CIL} (e.g., subject reduction, confluence modulo type erasure) hold for $\lambda^{\text{CIL}}_{\text{ul}}$.

5.5 Implicitly Typed Language $\lambda^{\text{CIL}}_{\text{i}}$

The implicitly typed language $\lambda^{\text{CIL}}_{\text{i}}$ is automatically obtained from $\lambda^{\text{CIL}}_{\text{ul}}$ and $\lambda^{\text{CIL}}_{\text{ut}}$. The syntax and semantics of $\lambda^{\text{CIL}}_{\text{i}}$ are the same as $\lambda^{\text{CIL}}_{\text{ut}}$ as given in figure 1. The types of $\lambda^{\text{CIL}}_{\text{i}}$ are those of $\lambda^{\text{CIL}}_{\text{ul}}$. The typing rules of $\lambda^{\text{CIL}}_{\text{i}}$ are the rules of $\lambda^{\text{CIL}}_{\text{ul}}$ modified by replacing every judgement $\tilde{A} \vdash_{\lambda^{\text{CIL}}_{\text{ul}}} \tilde{M} : \tilde{\tau}$ mentioned in a rule by $\tilde{A} \vdash_{\lambda^{\text{CIL}}_{\text{i}}} \left|\tilde{M}\right| : \tilde{\tau}$.

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. Furthermore, $\lambda^{\text{CIL}}_{\text{i}}$ appears to be the first implicitly typed lambda calculus with intersection and union types that has the subject reduction property:

Theorem 5.29 (Subject $\hat{c}$-Reduction for $\lambda^{\text{CIL}}_{\text{i}}$). *If $\hat{M} \longrightarrow_{\hat{c}} \hat{N}$ and $\tilde{A} \vdash_{\lambda^{\text{CIL}}_{\text{i}}} \hat{M} : \tilde{\tau}$, then $\tilde{A} \vdash_{\lambda^{\text{CIL}}_{\text{i}}} \hat{N} : \tilde{\tau}$.* □

Proof. Because $\tilde{A} \vdash_{\lambda^{\text{CIL}}_{\text{i}}} \hat{M} : \tilde{\tau}$, by definition of $\lambda^{\text{CIL}}_{\text{i}}$ we know there exists $\tilde{M}$ such that $\tilde{A} \vdash_{\lambda^{\text{CIL}}_{\text{ul}}} \tilde{M} : \tilde{\tau}$. Because $\hat{M} \longrightarrow_{\hat{c}} \hat{N}$ and by the unlabelled version of theorem 5.26, we know there exists $\tilde{N}$ such that $\tilde{M} \longrightarrow_{\tilde{r}} \tilde{N}$ and $\left|\tilde{N}\right| \equiv \hat{\tilde{N}}$. By the unlabelled version of theorem 5.21, we know that $\tilde{A} \vdash_{\lambda^{\text{CIL}}_{\text{ul}}} \tilde{N} : \tilde{\tau}$. By definition of $\lambda^{\text{CIL}}_{\text{i}}$, this implies that $\tilde{A} \vdash_{\lambda^{\text{CIL}}_{\text{i}}} \hat{N} : \tilde{\tau}$. □

6 Conclusion

We have presented λ^{CIL}, a typed λ-calculus that can be used as an intermediate language for optimizing compilers for higher-order polymorphic programming languages such as ML. We have shown that the calculus satisfies confluence and subject reduction properties.

As an intermediate language, λ^{CIL} is designed to facilitate verifiable flow-directed compiling with an emphasis on generating customized data representations. The intermediate language encourages the compiler writer to view values of intersection type as potential tuples and values of union type as potential variants. If a compiler analysis indicates that it would be advantageous to reify a potential tuple (variant) then it can be transformed into a real tuple (variant) and its type can be transformed into a product (sum) type. In a related paper, [DMTW97], we illustrate the utility of this approach by generating customized function representations. We are currently in the process of implementing this transformation framework.

The extent to which this approach can reduce the performance gap between first-order monomorphic languages and higher-order polymorphic languages remains unclear. What does seem clear is that some customization decisions are best made on a relatively low-level representation of the program in which "machine language polymorphism" is exposed. The extension of λ^{CIL} with low-level representation types along the lines of [MWCG98, MCGW97] remains an important open problem.

The flow type system presented here assumes that flow analysis and transformations are performed on entire programs, i.e., closed terms. In practice, it is necessary to support the analysis of modular program fragments. A simple approach is to extend flow labels with a distinguished "unknown" label; only conservative (and potentially expensive) representations could be used on

values annotated with this label. A more aggressive approach is to perform additional analysis and transformations when modules are linked together. Recent techniques for performing flow analysis across module boundaries [TJ94, Ban97, FF97] indicate that flow types are not inherently incompatible with modular program organization. However, link time optimizations remain a rich area for exploration.

Finally, an important practical consideration in compiling with types is controlling the size of the intermediate representations. Our current language duplicates terms when it duplicates types. While this language is conceptually convenient for specification, for implementation purposes a considerable size savings can be obtained by careful sharing of data structures. It could also be useful to pursue the use of a typed calculus with intersection and union types in the style of [Wel96].

References

[Age95] O. Agesen. The cartesian product algorithm. In *Proceedings of ECOOP'95, Seventh European Conference on Object-Oriented Programming*, vol. 952, pp. 2–26. Springer-Verlag, 1995.

[App92] A. W. Appel. *Compiling with Continuations.* Cambridge University Press, 1992.

[Ash96] J. M. Ashley. A practical and flexible flow analysis for higher-order languages. In POPL '96 [POPL96], pp. 195–207.

[ASU85] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison Wesley, 1985.

[AW93] A. S. Aiken and E. L. Wimmers. Type inclusion constraints and type inference. In *FPCA '93, Conf. Funct. Program. Lang. Comput. Arch.*, pp. 31–41. ACM, 1993.

[AWL94] A. S. Aiken, E. L. Wimmers, and T. K. Lakshman. Soft typing with conditional types. In POPL '94 [POPL94], pp. 163–173.

[vB93] S. van Bakel. *Intersection Type Disciplines in Lambda Calculus and Applicative Term Rewriting Systems.* PhD thesis, Catholic University of Nijmegen, 1993.

[Ban97] A. Banerjee. A modular, polyvariant, and type-based closure analysis. In ICFP '97 [ICFP97].

[Bar84] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics.* North-Holland, revised edition, 1984.

[Bar92] H. P. Barendregt. Lambda calculi with types. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, eds., *Handbook of Logic in Computer Science*, vol. 2, chapter 2, pp. 117–309. Oxford University Press, 1992.

[BDCd95] F. Barbanera, M. Dezani-Ciancaglini, and U. de'Liguoro. Intersection and union types: Syntax and semantics. *Information and Computation*, 119:202–230, 1995.

[BGS82] R. Brooks, R. Gabriel, and G. Steele. An optimizing compiler for lexically scoped LISP. In *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction*, pp. 261–275, June 1982.

[Ble93] G. E. Blelloch. NESL: A nested data-parallel language. Technical Report CMU-CS-93-129, School of Computer Science, Carnegie Mellon University, Apr. 1993.

[BR96] R. Bloo and K. H. Rose. Combinatory reduction systems with explicit substitution that preserve strong normalization. In *Proc. 7th Int'l Conf. Rewriting Techniques and Applications*, 1996.

[CU89a] C. Chambers and D. Ungar. Customization: Optimizing compiler technology for Self, a dynamically-typed object-oriented programming language. In PLDI '89 [PLDI89], pp. 146–160.

[CU89b] C. Chambers and D. Ungar. Iterative type analysis and extended message splitting: Optimizing dynamically-typed object-oriented programs. In *Proc. ACM SIGPLAN '90 Conf. Prog. Language Design & Implementation*, 1989.

[Cur90] P. Curtis. Constrained quantification in polymorphic type analysis. Technical Report CSL-90-1, XEROX PARC, CSLPubs.parc@xerox.com, 1990.

[DCG95] J. Dean, C. Chambers, and D. Grove. Selective specialization for object-oriented languages. In PLDI '95 [PLDI95].

[DMTW97] A. Dimock, R. Muller, F. Turbak, and J. B. Wells. Strongly typed flow-directed representation transformations. In ICFP '97 [ICFP97].

[EST95] J. Eifrig, S. Smith, and V. Trifonov. Type inference for recursively constrained types and its application to OOP. In *Proc. 1995 Mathematical Foundations of Programming Semantics Conf.* Elsevier, 1995.

[FF97] C. Flanagan and M. Felleisen. Componential set-based analysis. In *Proc. ACM SIGPLAN '97 Conf. Prog. Language Design & Implementation*, 1997.

[Gir72] J.-Y. Girard. *Interprétation Fonctionnelle et Elimination des Coupures de l'Arithmétique d'Ordre Supérieur.* Thèse d'Etat, Université de Paris VII, 1972.

[Gre77] B. S. Greenberg. The Multics MACLISP compiler, the basic hackery — a tutorial. http://www.lilli.com/lcp.html, 1977.

[Hei95] N. Heintze. Control-flow analysis and type systems. In SAS '95 [SAS95], pp. 189–206.

[HJ94] F. Henglein and J. Jorgensen. Formally optimal boxing. In POPL '94 [POPL94], pp. 213–226.

[HM95] R. Harper and G. Morrisett. Compiling polymorphism using intensional type analysis. In *Conf. Rec. 22nd Ann. ACM Symp. Principles of Programming Languages*, 1995.

[ICFP97] *Proc. 1997 Int'l Conf. Functional Programming*, 1997.

[Jim95] T. Jim. Rank-2 type systems and recursive definitions. Technical Report MIT/LCS/TM-531, Massachusetts Institute of Technology, Nov. 1995.

[Jim96] T. Jim. What are principal typings and what are they good for? In POPL '96 [POPL96].

[Jon94] M. P. Jones. Dictionary-free overloading by partial evaluation. In *ACM SIGPLAN Workshop on Partial Eval. & Semantics-Based Prog. Manipulation*, 1994.

[JW95] S. Jagannathan and A. K. Wright. Effective flow analysis for avoiding run-time checks. In SAS '95 [SAS95], pp. 207–224.

[JW96] S. Jagannathan and A. Wright. Flow-directed inlining. In *Proc. ACM SIGPLAN '96 Conf. Prog. Language Design & Implementation*, pp. 193–205, 1996.

[JWW97] S. Jagannathan, S. Weeks, and A. Wright. Type-directed flow analysis for typed intermediate languages. In *Proc. 4th Int'l Static Analysis Symp.*, 1997.

[Klo80] J. W. Klop. *Combinatory Reduction Systems.* Mathematisch Centrum, Amsterdam, 1980. Ph.D. Thesis.

[KvOvR93] J. W. Klop, V. van Oostrom, and F. van Raamsdonk. Combinatory reduction systems: Introduction and survey. *Theor. Comp. Sc.*, 121(1–2):279–308, 1993.

[Ler92] X. Leroy. Unboxed objects and polymorphic typing. In *Conf. Rec. 19th Ann. ACM Symp. Principles of Programming Languages*, pp. 177–188, 1992.

[Ler97] X. Leroy. The effectiveness of type-based unboxing. In *Workshop Types in Compilation '97.* Technical report BCCS-97-03, Boston College, Computer Science Department, June 1997.

[MCGW97] G. Morrisett, K. Crary, N. Glew, and D. Walker. Stack-based typed assembly language. Submitted., Dec. 1997.

[MMH96] Y. Minamide, G. Morrisett, and R. Harper. Typed closure conversion. In POPL '96 [POPL96].

[Mor95] G. Morrisett. *Compiling with Types.* PhD thesis, Carnegie Mellon University, 1995.

[Mos97] C. Mossin. *Flow Analysis of Typed Higher-Order Programs.* PhD thesis, DIKU, University of Copenhagen, January 1997.

[MP88] J. C. Mitchell and G. D. Plotkin. Abstract types have existential type. *ACM Transactions on Programming Languages and Systems*, 10(3):470–502, July 1988.

[MWCG98] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to typed assembly language. In POPL '98 [POPL98].

[PC95] J. Plevyak and A. Chien. Type directed cloning for object-oriented programs. In *Workshop for Languages and Compilers for Parallel Computers*, Aug. 1995.

[Pie91] B. C. Pierce. Programming with intersection types, union types, and polymorphism. Technical Report CMU-CS-91-106, Carnegie Mellon University, Feb. 1991.

[PJ96] S. L. Peyton Jones. Compiling Haskell by program transformation: A report from the trenches. In *Proc. European Symp. on Programming*, 1996.

[PJL91] S. L. Peyton Jones and J. Launchbury. Unboxed values as first class citizens in a non-strict functional language. In *FPCA '91, Conf. Funct. Program. Lang. Comput. Arch.*, vol. 523 of *LNCS*, Cambridge, MA. U.S.A., 1991. Springer-Verlag.

[PJM97] S. L. Peyton Jones and E. Meijer. Henk: A typed intermediate language. Submitted, Jan. 1997.

[PLDI89] *Proc. ACM SIGPLAN '89 Conf. Prog. Language Design & Implementation*, 1989.

[PLDI95] *Proc. ACM SIGPLAN '95 Conf. Prog. Language Design & Implementation*, 1995.

[Ple96] J. Plevyak. *Optimization of Object-Oriented and Concurrent Programs.* PhD thesis, University of Illinois at Urbana-Champaign, 1996.

[Plo75] G. D. Plotkin. Call-by-name, call-by-value and the lambda calculus. *Theor. Comp. Sc.*, 1:125–159, 1975.

[POPL94] *Conf. Rec. 21st Ann. ACM Symp. Principles of Programming Languages*, 1994.

[POPL96] *Conf. Rec. POPL '96: 23rd ACM Symp. Principles of Prog. Languages*, 1996.

[POPL98] *Conf. Rec. POPL '98: 25th ACM Symp. Principles of Prog. Languages*, 1998.

[PP98] J. Palsberg and C. Pavlopoulou. From polyvariant flow information to intersection and union types. In POPL '98 [POPL98].

[Rey74] J. C. Reynolds. Towards a theory of type structure. In *Colloque sur la Programmation*, vol. 19 of *LNCS*, pp. 408–425, Paris, France, 1974. Springer-Verlag.

[Rey96a] J. C. Reynolds. Design of the programming language Forsythe. Technical Report CMU-CS-96-146, Carnegie Mellon Univ., Pittsburgh, PA 15213, USA, June 28 1996.

[Rey96b] J. C. Reynolds. Design of the programming language Forsythe. In P. O'Hearn and R. D. Tennent, eds., *Algol-like Languages.* Birkhauser, 1996.

[SA95] Z. Shao and A. Appel. A type-based compiler for Standard ML. In PLDI '95 [PLDI95].

[SAS95] *Proc. 2nd Int'l Static Analysis Symp.*, 1995.

[SGL96] V. C. Sreedhar, G. R. Gao, and Y. Lee. Identifying loops using DJ graphs. *ACM Transactions on Programming Languages and Systems*, 18(6):649–658, 1996.

[Sha94] Z. Shao. *Compiling Standard ML for Efficient Execution on Modern Machines.* PhD thesis, Princeton University, 1994.

[Shi91] O. Shivers. Data-flow analysis and type recovery in Scheme. In P. Lee, ed., *Topics in Advanced Language Implementation*. MIT Press, 1991.

[Ste78] G. Steele. Rabbit: A compiler for Scheme. Technical Report MIT/AI-TR-474, Massachusetts Institute of Technology, 1978.

[Tar96] D. Tarditi. *Design and Implementation of Code Optimizations for a Typed-Directed Compiler for Standard ML*. PhD thesis, Carnegie Mellon University, 1996. Dec.

[TJ94] Y. Tang and P. Jouvelot. Separate abstract interpretation for control-flow analysis. *LNCS*, 789:224–??, 1994.

[WDMT97] J. B. Wells, A. Dimock, R. Muller, and F. Turbak. A typed intermediate language for flow-directed compilation. In *Proc. 7th Int'l Joint Conf. Theory & Practice of Software Development*, 1997. Superseded by [**?**].

[Wel96] J. B. Wells. Intersection types revisited in the Church style. Manuscript, June 1996.

[WS94] M. Wand and P. Steckler. Selective and lightweight closure conversion. In POPL '94 [POPL94], pp. 435–445.

A Combinatory Reduction Systems

We use the *functional* presentation of CRS's [KvOvR93]. An alternative *applicative* presentation can be found in [Klo80]. 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* $\mathrm{Fun}(\Sigma)$ (in the applicative presentation, a set of constants) and a set of *reduction rules* $\mathrm{Red}(\Sigma)$ (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 $[x]s \in \mathrm{MTer}\mathcal{F}$.
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 *context* is defined for metaterms and terms as usual.

A *valuation* ν : MVar $\rightarrow$ 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 MTer$\mathcal{F}$ to Ter($\mathcal{F}$) as follows[17]:

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 $s \rightarrow t$ (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 $r = s \rightarrow t$ 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 $s \rightarrow t$ 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 C it is the case that (1) $\nu(s) = C[\nu'(t)]$ and (2) the position[18] 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 $s \rightarrow t$ and $s' \rightarrow t'$ 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 Red(Σ) 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 $u \longrightarrow\!\!\!\!\rightarrow_\Sigma v_1$, and $u \longrightarrow\!\!\!\!\rightarrow_\Sigma v_2$, then there exists u' such that $v_1 \longrightarrow\!\!\!\!\rightarrow_\Sigma u'$ and $v_2 \longrightarrow\!\!\!\!\rightarrow_\Sigma u'$.* □

Proof. See [Klo80] or [KvOvR93] for proofs that any regular (orthogonal) CRS is confluent. □

[17] This definition of valuation differs from that of [KvOvR93] and [Klo80] (which differ from each other anyway), but produces equivalent results.

[18] We leave this notion of *position* unspecified. See [KvOvR93] or [Klo80] for a more precise definition.