Compiling with Polymorphic and Polyvariant Flow Types*

Franklyn Turbak
Wellesley College
fturbak@wellesley.edu

Allyn Dimock
Harvard University
dimock@das.harvard.edu

Robert Muller
Boston College
muller@cs.bc.edu

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

August, 1997

Abstract

Optimizing compilers for function-oriented and object-oriented languages exploit type and flow information for efficient implementation. Although type and flow information (both control and data flow) are inseparably intertwined, compilers usually compute and represent them separately. Partially, this has been a result of the usual polymorphic type systems using $\forall$ and $\exists$ quantifiers, which are difficult to use in combination with flow annotations.

In the Church Project, we are experimenting with intermediate languages that integrate type and flow information into a single *flow type* framework. This integration facilitates the preservation of flow and type information through program transformations. In this paper we describe λ^{CIL}, an intermediate language supporting polymorphic types and polyvariant flow information and describe its application in program optimiziation. We are experimenting with this intermediate language in a flow and type-directed compiler for a functional language. The types of λ^{CIL} encode flow information (1) via labels that approximate sources and sinks of values and (2) via intersection and union types, finitary versions of universal and existential types that support type polymorphism and (in combination with the labels) polyvariant flow analysis. Accurate flow types expose opportunities for a wide range of optimizing program transformations.

This paper is organized as follows. In section 1, we explain the background of the use of types and flow information in compilation and the motivation for our approach to combining types and flow information. In section 2, we present the language λ^{CIL} as an example of a language with polymorphic and polyvariant flow types and give simple examples of how it can be used. Section 3 concludes with a discussion of research directions.

1 Background and Motivation

Modern function-oriented programming languages (e.g., ML, Haskell, Scheme) are perceived to be more expressive but less efficient than traditional imperative languages (e.g., Fortran, C, Pascal, Ada). The perceived inefficiencies of function-oriented languages are due to a combination of direct and indirect costs incurred by straightforward implementations of features that make these languages expressive.

The *direct cost* of a feature is the overhead of the particular method of implementing it. Here are three key features of modern function-oriented languages and their associated direct costs:[1]

- *First-class lexically-scoped functions*: Lexically scoped functions can be created in any naming scope and "remember" the name bindings in the scoped where they were created, even if they escape that scope. Like other first-class values, such functions can be named, passed as arguments to functions, returned as results from functions, and stored in data structures. The combination of lexical scoping and first-classness imply that functions cannot be implemented as mere code pointers; in general, they

*This is a revised version of a paper that appears in [TIC97].

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

[1]In addition to the listed features, many function-oriented languages support additional features that are challenging to implement efficiently: lazy evaluation (Haskell), dynamic type checking (Scheme), generic arithmetic (Scheme), exception handling (ML and Scheme), and first-class continuations (ML and Scheme).

must be implemented as *closures* that pair a code pointer to a function with the values of that function's free variables. The time and space overheads of packaging, unpackaging, and managing the storage of these closures are direct costs of functional programming.

- *Parametric polymorphism*: Many functional languages support parametric polymporphism, in which a single generic function definition can be instantiated at many different argument and result types. Parametric polymorphism is traditionally modeled with universal types [Rey74, Gir72]. A universally polymorphic function must work when used at any of an infinite set of substitution instances of the universal type. It is challenging to implement such polymorphic functions efficiently. In the absence of information about the types at which it is instantiated, a universal type is almost a *promise* of a general implementation. In a straightforward implementation, a single copy of the polymorphic function is compiled under the assumption that all arguments and results are *boxed*, i.e., are uniformly represented as pointers to values. In this approach, the overhead of allocating, dereferencing, and deallocating boxed values can make polymorphic functions much less efficient than monomorphic functions.

- *Abstract data types*: Some functional languages support abstract data types, in which a single data type interface specification to be implemented in multiple ways without exposing the details of any implementation to the clients of an abstract data type. Abstract data types are dual to parametrically polymorphic functions in the sense that a single client works for many different implementation types. They are usually modeled with existential types [Mit96], which suffer problems similar to universal types. In particular, values of an abstract data type are typically boxed, and client code cannot take advantage of any representation-specific optimizations since the data type representations are generally not apparent.

The direct costs described above are particularly expensive in the presence of a *uniform representation assumption* that requires the most general representation to be used for each value, regardless of how it is used. Under this assumption, the direct costs are incurred in all programs, whether or not they make use of a feature. For instance, although a function without free variables is conceptually just a code pointer, it may be represented as a code/environment closure pair to be consistent with a uniform calling convention. Similarly, since it is not always apparent when a value is destined to be an argument of a polymorphic function, many implementations uniformly box *all* values. This strategy has the undesirable property that even purely monomorphic code must pay the cost of the *possibility* for polymorphism in the language.

The *indirect cost* of a feature is the inability to perform traditional optimizations that are obscured by the presence of the feature. For example, classical code motion and loop optimizations [ASU85] depend on local control and data flow information that is not readily apparent when a program is expressed as a collection of recursive functions or methods, many of which may be higher-order. Similarly, boxing integer and floating point values may interfere with storing them directly in machine registers.

It is worth noting that the efficiency problems enumerated for functional languages are shared by many object-oriented languages (e.g., Java, Smalltalk, Eiffel). Message-passing objects have a structure similar to that of closures. Small method size and dynamic method dispatch thwart traditional optimizations. The exact class of a method argument is often unknown (polymorphism). Even if the exact class of an argument is known, its representation may be hidden by data encapsulation (abstract data types).

Optimizing compilers for modern languages exploit type and/or flow information to reduce the direct and indirect costs of expressive language features. Types can be used by compilers to guide data-representation decisions and program transformations for improved program performance. For example, in *type-directed specialization*, different copies of a polymorphic function or method are compiled for each type at which the polymorphic routine is used [CU89, Ble93, Jon94, PC95]. The cost of polymorphism can be isolated to uses of polymorphic functions by wrapping them in *boxing coercions* that are determined by the instantiated types [PJL91, Ler92, HJ94]. But boxing coercions introduce new problems — they imply copying and coercions that are expensive for compound data and recursive functions and that are semantically incorrect for mutable data structures[Mor95]. An alternative is to use *dynamic type dispatch* to coerce a function to a specialized version based on an explicit type argument[Mor95, TMC+96].

Compilers also use flow information to ameliorate the costs of expressive language features. A flow analysis provides a conservative approximation of which program points (*sources*) can produce the values that are consumed at other program points (*sinks*). Traditionally, flow analyses merely pair program points

(a, b) where the value of a might become the value of b. In languages making heavy used of higher-order functions or object methods, flow information can show global patterns that are not apparent from local structure. For example, flow analysis is used to guide function and method inlining [PC95, JW96], efficient function representations [WS94, Ash96], the detection of loops hidden in function calls [Ash96], and type recovery and type check elimination in dynamically typed languages [Shi91b, JW95].

As an example of type and flow based optimizations, consider the following ML functions:

```
fun pfoldr p f i [] = (i,i)
  | pfoldr p f i (x::xs) =
      let val (l,r) = pfoldr p f i xs
       in if (p x)
          then (f(x,l),r)
          else (l,f(x,r))
      end

fun test (b, L1, L2, L3) =
  let val even_odd =
       pfoldr (fn x => x mod 2 = 0)
              (fn (n,len) => 1 + len)
              0
      val sums =
       pfoldr (if b
               then (fn y => y > 0.0)
               else (fn z => z < 0.0))
              op+
              0.0
   in (even_odd L1,
       let val (_,r) = even_odd L2 in r end,
       sums L3)
  end
```

Using binary operator `f` and initial value `i`, `pfoldr` accumulates two results from a list — one for elements satisfying predicate `p` and one for the other elements. The `test` function uses `pfoldr` to count the number of evens and odds in the integer list `L1`, to count the number of odds in the integer list `L2`, and to collect the sums of floating point numbers in `L3` according to a predicate determined by a boolean `b`.

Ideally[2], a compiler would be able to use type and flow information to choose from all of the following optimizations:

- Since `pfoldr` is instantiated at two types, it is possible to specialize `pfoldr` into two copies, one for lists of integers and one for for lists of floats. This would permit unboxed representations for the elements of the lists `L1`, `L2`, and `L3`, as well as for the elements of the tuples manipulated by `pfoldr`.

- It is possible for `even_odd` to be a specialized copy of `pfoldr` that results from in-lining the arguments to `pfoldr`. This sort of in-lining can be performed without any fancy flow analysis [App92, TMC+96, Tar96]. However, in the presence of non-trivial flow patterns, such as the conditional argument to `pfoldr` in the definition of the function `sums` that chooses between two predicates, in-lining can require more sophisticated flow analyses and transformations. In this case, two options are (1) to in-line the two predicates at `(p x)` within a single copy of `pfoldr` and discriminate between them via `b`, and (2) to in-line each of the two predicates in a separate copy of `pfoldr` and discriminate between the copies via `b`.

- Flow information can be used to to determine that only the second component of the tuple returned by `even_odd L2` is referenced. It should be possible to replace this call by a call to a specialized function that computes and returns only the second tuple component, thus avoiding the overheads of computing the first component and constructing and unpacking tuples.

[2]We are not aware of a current compiler that actually performs all of listed optimizations.

- Flow information can show that the tuple returned by the recursive calls to `pfoldr` is used linearly. This should allow the compiler to use in-place updates to construct the new return value rather than building a new tuple.

Because of the benefits of type and flow information, increasingly sophisticated type and flow analyses are being employed in compilers. Much recent work has focused on transmitting type information through the stages of a compiler via *typed intermediate languages* and *well-typedness-preserving transformations* [PJHH+93, TMC+96]. Not only is the preserved type information important for guiding representation decisions and enabling optimizations, but it also serves as a helpful tool for proving the correctness of transformations and debugging compiler implementations [TMC+96, PJM97].

While type information has been tightly integrated into modern intermediate languages, flow information has not. The only implemented or partially implemented languages of which we are aware which merge type and flow information are those based on constrained types [Cur90, AW93, AWL94, EST95]. It is not clear whether any work with constrained types has used a typed intermediate language. Also, support in constrained type systems for type polymorphism has been via **let**-style polymorphism, which is difficult to use in a typed intermediate language without losing the ability to express a polyvariant flow analysis. Formal connections have been established between monovariant flow analyses and monomorphic type system [PO95, Hei95], but typed intermediate languages need type polymorphism. In those languages which have used flow analysis with a typed intermediate language, the results of flow analysis have been maintained separately from the typed intermediate representation [JWW97]. Any formalism connecting the flow information to the typed program is outside the type system. Because the flow analysis is separate from the type system, the type system itself does not help in proving that program transformations preserve the correctness of the flow analysis.

Historically, most flow analysis work for higher-order languages has been done in the context of dynamically typed languages rather than statically typed ones. We hypothesize that the absence of static types in dynamically typed languages has forced researchers working with these language to explore flow-based techniques more aggressively as a means of determining the sort of type information naturally available in statically typed languages. But since flow analysis provides more than just type information, it has benefits for statically typed languages as well. Moreover, recent work on flow analysis in higher-order typed languages suggests that there are some important synergies between flows and types in typed languages. For example, types can provide a basis for polyvariant flow analysis [Ban97, JWW97] that is more natural than the call string contours of the n-CFA approach (see [Shi91a] for n-CFA).

To address these issues, we propose that the type and flow information in a compiler should be integrated into a single *flow type* system, and that flow types should be the basis for compiler intermediate languages. Flow types offer the following advantages for an intermediate language:

- Since flow types provide more information than types alone, they can support a wider range of transformations and optimizations.
- Because transformations must preserve well-typedness, they will automatically preserve the correctness of the flow analysis embedded in the types, i.e., the flow analysis must continue to be a conservative approximation of the actual flow. It is 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. The value of typed intermediate languages for verifying and debugging compiler implementation has been emphasized by other researchers [TMC+96, PJ96].

2 A Calculus of Polymorphic and Polyvariant Flow Types

We have designed a calculus, λ^{CIL}, for experimenting with polymorphic and polyvariant flow types.[3] In λ^{CIL}, flow information is encoded in two ways: (1) via flow label annotations that approximate the flow of values

[3]CIL stands for Church Intermediate Language. The goal of the Church Project is to explore the use of advanced type systems in the design and implementation of modern programming languages. For more information about the Church Project, see `http://www.cs.bu.edu/groups/church`.

from source expressions to sink expressions; and (2) via terms of intersection and union type. Intuitively, intersection and union types are finitary versions of infinitary universal and existential types. Intersection and union types are capable of expressing polymorphism and abstract data types. In conjunction with flow labels, they can express forms of polyvariant flow analyses.

We are using λ^{CIL} as the intermediate language for a framework that supports multiple closure representations for functions [DMTW97]. Some practical difficulties with λ^{CIL} include controlling the size of representations in λ^{CIL} and the dependence of λ^{CIL} on the closed program assumption (which is at odds with modular programming and separate compilation). Nevertheless, we feel that λ^{CIL} serves as a proof-of-concept for the notion of flow types. We expect that further research will continue to improve flow type systems.

In this section, we give an informal overview of λ^{CIL} by discussing its syntax and semantics in the context of some simple examples. Along the way, we also explain some of the design decisions underlying the language. For a more formal presentation, see [WDMT97]; a summary of the formal presentation appears in the appendix of [DMTW97].

We present three distinct versions of λ^{CIL}, which differ in terms of whether they include types and/or flow label annotations.

1. the *untyped* language $\lambda^{\text{CIL}}_{\text{ut}}$ has neither types nor flow labels.
2. the *unlabelled* language $\lambda^{\text{CIL}}_{\text{ul}}$ 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.

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

The untyped language $\lambda^{\text{CIL}}_{\text{ut}}$ is a call-by-value lambda calculus extended with constants, recursion, and positional products and sums. Here is a sample $\lambda^{\text{CIL}}_{\text{ut}}$ term that returns a tuple containing the results of three applications of the identity function: $\hat{M_a} \equiv \textbf{let } f = \lambda x.x \textbf{ in } \times(f \mathbin{@} 17, f \mathbin{@} 23, f \mathbin{@} \textbf{true})$

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

Application is indicated by an explicit @ symbol, which serves as a place to hang flow labels in the typed versions of the language. $\times(M_1, \ldots, M_n)$ creates a tuple with n components; The ith component of the tuple denoted by M can be selected via $\pi^{\times}_i M$. In the untyped language, **let** is syntactic sugar for a λ application. For presentational purpose, we use standard infix primitives in our examples even though they do not appear in the formal calculus. The call-by-value reduction rules for λ^{CIL} (not shown here) are straightforward. Via these rules, $\hat{M_a}$ reduces to the normal form $\times(17, 23, \textbf{true})$.

Here is second sample expression, which illustrates variants:

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

Both g and h are functions that take a variant value as their argument. Variant values are constructed via $\big(\textbf{in}^+_i\ M\big)$, which injects the value of M into a variant with tag i. Variants are deconstructed via a $\textbf{case}^+$ expression. 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 be of the form $\big(\textbf{in}^+_i\ V\big)$, where $1 \le i \le n$, and V is a value (i.e., constant, abstraction, tuple of values, or injection 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). 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]. $\hat{M_b}$ reduces to the normal form $\times(4, 12, 1)$.

2.2 The Unlabelled Language $\lambda_{\mathrm{ul}}^{\mathrm{CIL}}$

The typed but unlabelled language $\lambda_{\mathrm{ul}}^{\mathrm{CIL}}$ is an extension to $\lambda_{\mathrm{ut}}^{\mathrm{CIL}}$ in which all variables (except those introduced by **case** expressions) and injections are annotated with a type. In addition to the more familiar types — base types, function types, tuple types, variant types, and recursive types — $\lambda_{\mathrm{ul}}^{\mathrm{CIL}}$ provides intersection and union types.

Intersection types model the types of polymorphic functions in terms of the types at which they are used. For example, the identity function $\lambda x.x$ that appears in the untyped term $\hat{M}_a$ can be assigned the type $\wedge[\mathrm{int} \to \mathrm{int}, \mathrm{bool} \to \mathrm{bool}]$ 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 $\forall\tau.\tau \to \tau$. Universal types do not provide any information about the types at which polymorphic functions are instantiated. In effect, they strongly suggest a uniform representation, which implies overheads like boxing. In contrast, intersection types are finitary versions of universal types that 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 different instantiations of a polymorphic function).

Union types model the types of data abstractions in terms of the types that implement the abstractions. Union types allow otherwise incompatible types to be merged 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 abstraction 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 $\times[\mathrm{int} \to \mathrm{int}, \mathrm{int}]$, while the thunk $\times(\lambda z.\mathbf{if}\ z\ \mathbf{then}\ 1\ \mathbf{else}\ 0, w)$ can be assigned the type $\times[\mathrm{bool} \to \mathrm{int}, \mathrm{bool}]$. Even though these types are distinct, 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 modelled with existential types [Mit96], which hide the unobservable types. For example, both of the above types are instances of the existential type $\exists\tau.\times[\tau \to \mathrm{int}, \tau]$. As with universal types, existential types hide usage information and imply unnecessary overheads. Union types are finitary versions of existentials that expose the implementation types of a data abstraction. For example, the union type for the thunks would be $\vee[\times[\mathrm{int} \to \mathrm{int}, \mathrm{int}], \times[\mathrm{bool} \to \mathrm{int}, \mathrm{bool}]]$.

Here is a $\lambda_{\mathrm{ul}}^{\mathrm{CIL}}$ term corresponding to the untyped term $\hat{M}_a$:[4]

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

The notation $\bigwedge\left(\lambda x^{\mathrm{int}}.x, \lambda x^{\mathrm{bool}}.x\right)$ 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 $\pi_i^{\wedge}\ M$ selects one of the type-annotated copies from the virtual tuple. Virtual tuples and projections are entirely compile-time constructions whose purpose is to facilitate type-checking by tracking the different types at which a polymorphic value is used. All components of a virtual tuple denote the same run-time value; no code will be generated to construct or access the slots of a virtual tuple at run-time. Because $\lambda_{\mathrm{ul}}^{\mathrm{CIL}}$ uses virtual copies of terms as a kind of type annotation, we refer to it as a *duplicating calculus*.

Although the virtual tuple and virtual projection notations are somewhat cumbersome, they have two key benefits:

- They solve 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 [Rey96, 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.

- The notational similarity between products and intersections is specifically designed to suggest *splitting* transformations in which a virtual tuple and its corresponding projections are transformed into real

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

tuples and projections — 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

$$\wedge[\times[\mathrm{int},\mathrm{bool}] \to \times[\mathrm{bool},\mathrm{int}], \times[\mathrm{real},\mathrm{real}] \to \times[\mathrm{real},\mathrm{real}]]$$

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 booleans than for swapping reals (assuming that reals have a different size from integers and booleans). This specialization is expressed by converting the $\wedge$ in the type and corresponding term to a $\times$ 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].

A positional 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 flow path for each usage type of a polymorphic value, but the flow paths 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. Intersections can be used in this way to track different flows of *any* value, even a monomorphic one. The ability to distinguish the flows of different values generated by a single source term is a hallmark of *polyvariant* flow analysis [Ban97, JWW97]. The above example illustrates how intersections can encode polyvariance in $\lambda_{\mathrm{ul}}^{\mathrm{CIL}}$.

Although our intersection examples happens to exhibit Hindley-Milner ("**let**-style") polymorphism, 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:

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

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

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

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 $\lambda_{\text{ul}}^{\text{CIL}}$:[5]

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

The two incompatible types returned by the body of g are merged into the union type $\vee[\rho_1, \rho_2]$. Terms of union type are constructed by injecting a term into a *virtual variant*. Virtual variants are analyzed by *virtual cases* (i.e., $\textbf{case}^{\vee}$ terms), which are the duals of virtual tuples.[6] 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 specialize 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 specialization can be expressed by a *taggging* 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 the tagging 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, more fine 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 specializations on the corresponding $\textbf{case}^{\vee}$ clauses to be made based on flow information more fine-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 flow paths connecting the sources of values with the sinks of values. For instance, [DMTW97] uses a simple network connecting two functions with two application sites to illustrate various approaches to closure conversion.

2.3 The Flow Typed Language λ^{CIL}

The flow typed language λ^{CIL} extends $\lambda_{\text{ul}}^{\text{CIL}}$ with flow label annotations on terms and types. Terms are characterized as sources (value producers) or sinks (value consumers); some may be both (e.g., arithmetic operators), while others (virtual tuples, projections, injections, cases, and coercions) are neither. 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.[7]

[5] In $\lambda_{\text{ul}}^{\text{CIL}}$ 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.

[6] 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.

[7] 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 value (e.g., tuples, variants, numbers), but for simplicity of presentation we label only function values. The fact that we don't label tuples and other values does not

Here is a sample flow-annotated term:

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

For readabilty, 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 transitive closure 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 [WDMT97].

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].

The flow information in λ^{CIL} flow types can guide the sorts of flow-directed optimizations mentioned in Section 1. For example, if the flow annotations indicate that only one function flows to a call site, the function may be in-lined at that site (though special care needs to be taken to handle open functions, i.e, functions with free variables) [JW96]. When several functions flow to a call site, it is possible to dispatch to one of several in-lined functions. Flow information may also be helpful for detecting values that are used in a linear fashion. We have used flow information in conjunction with the splitting and tagging 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 intersections and unions to refines flow types and express finer-grained flow distinctions. Flow separation highlights the correspondence between flows and intersection and union types.

The notion of integrating flow and type information into a single flow type system is not new. Tang and Jouvelot track function flows via control flow effects annotating arrow types [Tan94, TJ94]. Heintze uses labelled types to show the equivalences between type systems and flow systems [Hei95]. Banerjee uses flow labels and intersection types in his combined approach to type inference and flow analysis [Ban97].

What is new about the flow type system of λ^{CIL} is that the fine-grained flow distinctions afforded by intersection and union types make it possible to express a wide range of polyvariant flow analyses. A *polyvariant* flow analysis is one in which a single occurrence of a term can be analyzed in multiple contexts. The flow type systems proposed thus far are either monovariant or only support polyvariance in limited ways. In our flow type framework, intersections and unions can encode the results of a wide range of polyvariant flow analyses. Intersection components model distinct value contexts and union components model distinct continuation contexts. Our framework naturally encodes the type-based polyvariance of [JWW97], but can also encode other polyvariant contexts, such as the call-string contours of [Shi91a]. We are investigating a formal characterization of the flow analyses that can be expressed in λ^{CIL}.

.

limit the range of function flow analyses of functions that can be expressed. It just limits the ease with which transformations can be performed on unlabelled values.

3 Research Directions

Our investigation of compiling with flow types is still in a preliminary stage and many important steps remain to be taken. Here we outline future research directions.

- *Prototype implementation*: We are implementing a prototype flow type compiler for a purely functional subset of ML without modules. We hypothesize that the combination of flow and type information supports more transformations that either kind of information alone. An important aspect of the implementation will be developing heuristics that make use of the type and flow information to guide representation decisions. We expect that a naive representation of terms and types in the duplicating calculus will incur high overheads, so we are investigating more efficient representation schemes. One of us (Wells) is exploring a non-duplicating version of the calculus.

- *Flow analysis experimentation*: λ^{CIL} does not prescribe a particular type inference or flow analysis strategy. In our current implementation, type inference is performed on ML-like terms using the type inference system for rank-2 intersection types developed by Jim [Jim96]. Although the result of type inference satisfies the rank-2 restriction, there is no restriction on the types of terms that are produced by subsequent program transformations. The type inference system attaches only trivial flow labels to terms and types. These are replaced by more precise labels in a subsequent flow analysis pass. Our current flow analysis algorithm is patterned after the flow analysis component of Banerjee's combined type and flow inference system [Ban97]. The analysis in the current prototype is polyvariant, in that it analyzes each element of a virtual tuple separately. The analysis in the prototype is constrained by a naive implementation of λ^{CIL}'s shallow subtyping restriction, which only allows coercions phrased in terms of the top-level labels of a type.

  We are interested in analyses that use additional elements in virtual tuples to circumvent the limitations of shallow subtyping, and in implementing (and extending) polymorphic splitting [JW95]. There is also evidence that even crude flow analysis can be useful for program optimization [Ash96]. One of us (Dimock) is developing a control flow analysis kit that will allow us to experiment with tradeoffs the between benefits and costs of analyses as the precision is varied.

- *Modularity*: Our current flow type system 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 uniform (and potentially expensive) representations could be used on values annoted 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.

- *Imperative features*: It is important to show that our techniques are still applicable in the context of imperative features like references and exceptions.

- *Connections with constrained types*: We are currently exploring connections between flow types and constrained types [Cur90, EST95, AW93, AWL94]. Our goal is to incorporate the polyvariant power of intersections and unions into the constrained type framework.

References

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

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

[ASU85] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 1985.

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

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

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

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

[CU89] Craig Chambers and David Ungar. Iterative type analysis and extended message splitting: Optimizing dynamically-typed object-oriented programs. In *Proc. ACM SIGPLAN '90 Conf. Prog. Lang. Design & Impl.*, 1989.

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

[DMTW97] Allyn Dimock, Robert Muller, Franklyn Turbak, and J. B. Wells. Strongly typed flow-directed representation transformations. In ICFP '97 [ICFP97], pages 11–24.

[EST95] Jonathan Eifrig, Scott Smith, and Valery Trifonov. Type inference for recursively constrained types and its application to OOP. In *Proc. 1995 Mathematical Foundations of Programming Semantics Conf.*, volume 1 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1995.

[FF97] Cormac Flanagan and Matthias Felleisen. Componential set-based analysis. In *Proc. ACM SIGPLAN '97 Conf. Prog. Lang. Design & Impl.*, 1997.

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

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

[HJ94] Fritz Henglein and Jesper Jorgensen. Formally optimal boxing. In POPL '94 [POPL94], pages 213–226.

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

[Jim96] Trevor Jim. What are principal typings and what are they good for? In POPL '96 [POPL96].

[Jon94] Mark P. Jones. Dictionary-free overloading by partial evaluation. In *PEPM '94 — ACM SIGPLAN Workshop Partial Eval. & Semantics-Based Prog. Manipulation*, 1994.

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

[JW96] Suresh Jagannathan and Andrew Wright. Flow-directed inlining. In PLDI '96 [PLDI96], pages 193–205.

[JWW97] Suresh Jagannathan, Stephen Weeks, and Andrew Wright. Type-directed flow analysis for typed intermediate languages. In *Proc. 4th Int'l Static Analysis Symp.*, volume 1302 of *LNCS*. Springer-Verlag, 1997.

[Ler92] Xavier Leroy. Unboxed objects and polymorphic typing. In *Conf. Rec. 19th Ann. ACM Symp. Princ. of Prog. Langs.*, pages 177–188. ACM, 1992.

[Mit96] John C. Mitchell. *Foundations for Programming Languages*. MIT Press, 1996.

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

[PC95] John Plevyak and Andrew A. Chien. Iterative flow analysis. Submitted, July 1995.

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

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

[PJHH+93] Simon L. Peyton Jones, Cordy Hall, Kevin Hammond, Will Partain, and Phil Wadler. The Glasgow Haskell compiler: A technical overview. In *Proc. UK Joint Framework for Information Technology (JFIT) Technical Conf.*, 1993.

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

[PJM97] Simon L. Peyton Jones and Erik Meijer. Henk: A typed intermediate language. In TIC '97 [TIC97].

[PLDI96] *Proc. ACM SIGPLAN '96 Conf. Prog. Lang. Design & Impl.*, 1996.

[PO95] Jens Palsberg and Patrick O'Keefe. A type system equivalent to flow analysis. *ACM Trans. on Prog. Langs. & Systs.*, 17(4):576–599, 1995.

[POPL94] ACM. *Conf. Rec. 21st Ann. ACM Symp. Princ. of Prog. Langs.*, 1994.

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

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

[Rey96] John C. Reynolds. Design of the programming language Forsythe. Technical Report CMU-CS-96-146, Carnegie Mellon Univ., Pittsburgh, PA 15213, USA, June 28 1996.

[SAS95] *Proc. 2nd Int'l Static Analysis Symp.*, volume 983 of *LNCS*, 1995.

[Shi91a] Olin Shivers. *Control Flow Analysis of Higher Order Languages.* PhD thesis, Carnegie Mellon University, 1991.

[Shi91b] Olin Shivers. Data-flow analysis and type recovery in Scheme. In Peter Lee, editor, *Topics in Advanced Language Implementation.* MIT Press, 1991.

[Tan94] Yan Mei Tang. *Systèmes d'Effet et Interprétation Abstraite pour l'Analyse de Flot de Contrôle.* PhD thesis, Ecole Nationale Superieure des Mines de Paris, 1994.

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

[TIC97] *Proc. First Int'l Workshop on Types in Compilation*, June 1997. The printed TIC '97 proceedings is Boston Coll. Comp. Sci. Dept. Tech. Rep. BCCS-97-03. The individual papers are available at http://www.cs.bc.edu/~muller/TIC97/ or http://oak.bc.edu/~muller/TIC97/.

[TJ94] Yan Mei Tang and Pierre Jouvelot. Separate abstract interpretation for control-flow analysis. *LNCS*, 789:224–243, 1994.

[TMC+96] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: A type-directed optimizing compiler for ML. In PLDI '96 [PLDI96].

[WDMT97] J. B. Wells, Allyn Dimock, Robert Muller, and Franklyn Turbak. A typed intermediate language for flow-directed compilation. In *Proc. 7th Int'l Joint Conf. Theory & Practice of Software Development*, pages 757–771, 1997.

[WS94] Mitchell Wand and Paul Steckler. Selective and lightweight closure conversion. In POPL '94 [POPL94], pages 435–445.