A Typed Intermediate Language for Flow-Directed Compilation

J. B. Wells*[1], Allyn Dimock[2], Robert Muller[3], and Franklyn Turbak[4]

[1] Boston University, Boston MA 02215, USA
[2] Harvard University, Cambridge MA 02138, USA
[3] Boston College, Chestnut Hill MA 02167, USA
[4] Wellesley College, Wellesley MA 02181, USA

Abstract. We present a typed intermediate language λ^{CIL} for optimizing compilers for function-oriented and polymorphically typed programming languages (e.g., ML). The language λ^{CIL} is a typed lambda calculus with product, sum, intersection, and union types as well as function types annotated with flow labels. A novel formulation of intersection and union types supports encoding flow information in the typed program representation. This flow information can direct optimization.

1 Introduction

Recently there has been much interest in the view of compilation as a composition of well typed program transformations. In this setting, the compiler maintains the invariant that at each step of the compilation process the intermediate representation of the source program is well typed. This invariant can be observed if the input program is well typed and each compiler transformation changes the intermediate representation and its typing in a consistent way. This approach requires using one or more typed intermediate languages.

Explicitly typed intermediate languages offer several benefits to the compiler writer [15, 18, 26, 19]. First, type information can guide program analyses and transformations. Second, some applications need accurate type information at run-time thereby requiring the compiler to preserve it. Finally, typed intermediate languages are useful as a debugging aid in the compiler development process.

This paper introduces a typed intermediate language for optimizing compilers for higher-order polymorphic programming languages. Our intermediate language[1] λ^{CIL} is an explicitly typed λ-calculus with product, sum, intersection, union as well as function types annotated with flow labels in the style of Heintze and Banerjee [12, 5].

The flow annotations on function types are sets of term labels that can encode control and data flow information as it would be computed by one of several typed flow analyses in the literature [12, 5]. If a flow analysis determines that

* Supported by NSF grant CCR–9417382.

[1] In λ^{CIL}, "C" is for the Church Project (http://www.cs.bu.edu/groups/church/) and "IL" is for "intermediate language." The Church Project is investigating the use of intersection and union types in compiling ML–like languages.

subterm occurrence M has type $\sigma \xrightarrow{\phi}_{\psi} \tau$, then the λ-abstractions flowing from M are those with labels in ϕ and they flow only to application sites with labels in ψ. The sets ϕ and ψ are sets of potential flow *sources* and *sinks*.

The formulation of λ^{CIL} allows flow information to be separated in a well typed manner to expose precise correspondences between sources and sinks of flow [8]. A λ-abstraction flowing to m application sites can be assigned an intersection type with m conjuncts. This is represented in λ^{CIL} by m virtual copies of the term. In a dual manner, an application to which n abstractions might flow can be assigned a union type with n disjuncts. This is represented in λ^{CIL} by a virtual case expression that dispatches to one of n clauses.

The program representation supported by λ^{CIL} can be exploited in generating efficient object code. One approach to compiling polymorphism is to generate *specialized* instances of a polymorphic definition based on its uses. Specialization not only avoids the overhead of boxing but more importantly enables subsequent optimizations such as inlining and common subexpression elimination. Empirical evidence suggests that the optimizations enabled by specialization can actually lead to smaller object programs than alternative approaches [14].

Typically the specialization approach is limited to non-escaping polymorphic functions where the required specializations of the definition are determined by its uses within the confines of a binding construct such as **let** [26, 7]. In λ^{CIL}, the required specializations can be determined by the flow analysis. Escaping polymorphic functions can be specialized for their uses in textually remote parts of the program. It is also easier in λ^{CIL} to provide multiple representations of a function for different types and for particular inputs. Inlining of functions can be performed even when multiple functions can flow to a call site. It can also be performed on open functions. This is further discussed in [8].

2 Flow-Directed Program Transformation

We informally illustrate the features of λ^{CIL} in the context of *closure conversion*, a key program transformation in optimizing compilers for function-oriented and object-oriented languages [30, 17, 10]. Closure conversion transforms programs that may contain open functions into equivalent programs that contain only closed functions. An important technical challenge in closure conversion is to generate efficient function representations without violating the invariant that all function representations flowing to a particular application site are consistent with that site's application protocol.

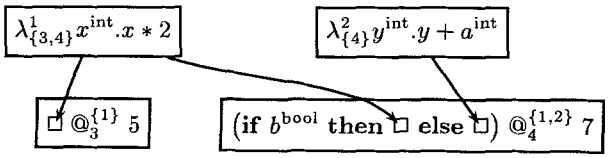
The simplest way to maintain this invariant is to give every function the same representation and use the same application protocol at every call site. In a naive strategy, closure conversion maps every source function to a *closure*, a pair of (1) the values of the function's free variables (the *environment*) and (2) a closed form of the function (the *code*) that takes the environment as an additional argument. However, the overhead of creating and applying a closure can often be avoided by choosing more efficient function representations.

We illustrate closure conversion with the following example: [2]

$$\begin{array}{l}\textbf{let } f^{\text{int}\to\text{int}} = \lambda x^{\text{int}}.x * 2 \\ \quad\; g^{\text{int}\to\text{int}} = \lambda y^{\text{int}}.y + a^{\text{int}} \\ \textbf{in } \times (f \mathbin{@} 5, (\textbf{if } b^{\text{bool}} \textbf{ then } f \textbf{ else } g) \mathbin{@} 7)\end{array} \tag{1}$$

The closed function $(\lambda x^{\text{int}}.x*2)$ flows to two call sites, the second of which is also a sink for the open function $(\lambda y^{\text{int}}.y+a^{\text{int}})$. The flow of this simple program is merely an example of more complex flow patterns arising in real programs.

A typed flow analysis of the example in λ^{CIL} might yield the flow graph: [3]



Each abstraction occurrence $(\lambda^l_\psi x^\tau.M)$ is identified by a label l and a set of labels ψ approximating the set of application occurrences that can consume it. Each application occurrence $(M \mathbin{@}^\phi_k N)$ is identified by a label k and a set of abstraction occurrence labels ϕ approximating the set of abstraction occurrences that it can consume. Function types "int $\xrightarrow[\psi]{\phi}$ int" are also annotated with sets of source and sink labels. [4]

Consider closure converting our example. The function $\lambda^1_{\{3,4\}}$ is already closed, so it is desirable to represent it as a function (not a closure) and to keep $@^{\{1\}}_3$ as a regular function application (not a closure application). This optimization is called *selective* closure conversion [30]. However, since $\lambda^1_{\{3,4\}}$ also flows to $@^{\{1,2\}}_4$ along with the open function $\lambda^2_{\{4\}}$, something must be done to ensure that the protocols at the call sites are consistent with the function representations that flow to them.

The flow-based features of λ^{CIL} are helpful in dealing with multiple representations that can be desirable in closure conversion. In λ^{CIL}, a term can be transformed to expose correspondences between sources and sinks via intersection ($\wedge$) and union ($\vee$) types:[5]

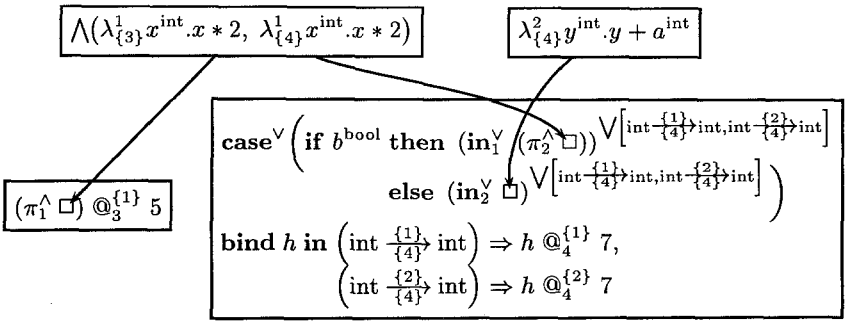
---

[2] Remarks on notation: Variables are annotated with types, applications are marked by "@", and tuples are marked by "$\times$". For readability, types on bound variable occurrences are omitted when the binding is present. We use base types (like int and bool), constants of these types, and familiar operators on these types, even though these are not formally defined.

[3] To emphasize that our approach addresses complex flow patterns, we present the example's flow graph diagramatically, detaching the abstractions and applications from their surrounding context.

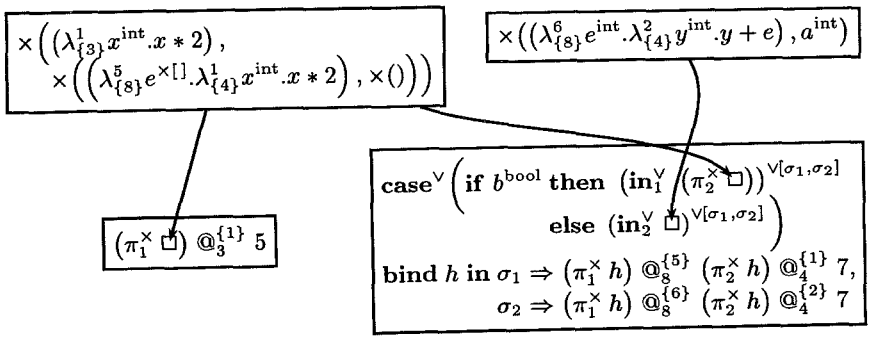
[4] For well-typedness, flow-label subtyping coercions may be needed. For readability, we omit these from examples.

[5] Notation: $\wedge(M_1, \ldots, M_n)$ constructs a term of intersection type $\wedge[\tau_1, \ldots, \tau_n]$ whose components are extracted via $\pi^\wedge_i$. $(\textbf{in}^\vee_i\ M)^{\tau_i}$ constructs a term of union type $\vee[\tau_1, \ldots, \tau_n]$ which is analyzed by $\textbf{case}^\vee$.



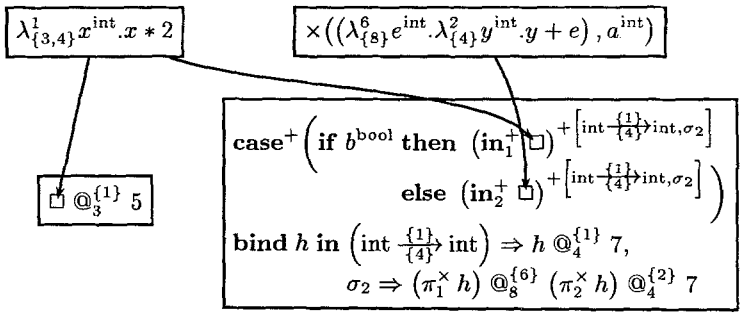
The abstraction occurrence $\lambda^1_{\{3,4\}}$ has been transformed into a *virtual tuple* (term of intersection type) containing two abstraction occurrences $\lambda^1_{\{3\}}$ and $\lambda^1_{\{4\}}$. Intuitively, a virtual tuple is a compile-time tuple containing copies of a term that differ only in their types. Since all of the components of a virtual tuple behave identically, no code will be generated to build or access its slots at run-time. Similarly, the application occurrence $@_4^{\{1,2\}}$ has been transformed into a *virtual case* expression that dispatches on the tag of a virtual variant to one of two application occurrences $@_4^{\{1\}}$ or $@_4^{\{2\}}$. All of the clauses of a virtual case expression will share the same code at run-time. The purpose of virtual tuples and variants is to make the term well typed and to provide a place to put type and flow annotations. However, a compiler can transform some virtual tuples ($\wedge$) to real tuples ($\times$) and some virtual variants ($\vee$) to real variants ($+$).

For example, one approach to closure converting our example is to *split* the virtual tuple for the closed function into two distinct functions representations, one which flows to $@_3^{\{1\}}$ and one which flows $@_4^{\{1\}}$. In this case, the virtual product becomes a real product, but the virtual variant stays virtual:

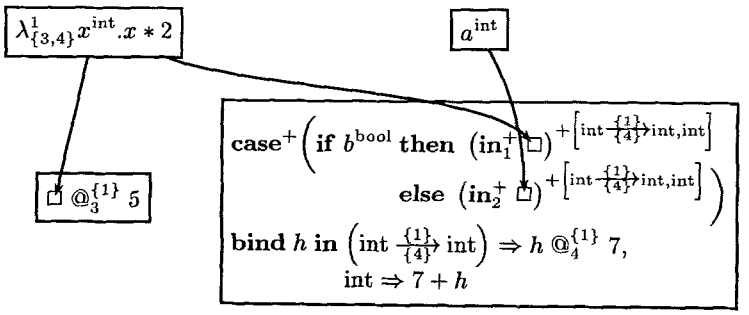


where $\sigma_1 = \times\left[\left(\times[]\xrightarrow{\{5\}}_{\{8\}}\text{int}\xrightarrow{\{1\}}_{\{4\}}\text{int}\right),\times[]\right]$ and $\sigma_2 = \times\left[\left(\text{int}\xrightarrow{\{6\}}_{\{8\}}\text{int}\xrightarrow{\{2\}}_{\{4\}}\text{int}\right),\text{int}\right]$

Another option is to use only one representation for the closed function, but to *tag* it to distinguish it from the open function representation. In this case, the virtual product stays virtual, but a case analysis will distinguish between the call protocols of the real variants at run-time:



λ^{CIL} can also handle *inlining*, a vital compiler optimization, as another function representation choice. In our example, the code of the open function can be inlined at its single call site, and the open function can be represented by the value of its sole free variable a rather than as a closure.



Not only does λ^{CIL} support the inlining of open functions, but the flow annotations in λ^{CIL} expose flow-based inlining opportunities that may not be apparent from the program text.

Every change from $\wedge$ to $\times$ or $\vee$ to $+$ may lead to a cascade of changes necesary to preserve well typedness and meaning. Our calculus aids in automatically handling these changes. Space does not permit us to specify the closure conversion transformations here; for details, see [8].

3 Design Issues

This section discusses some of the goals that guided the design of our language λ^{CIL} and some of the technical challenges that had to be overcome.

Finitary Types and Typings: A central theme of our work is the desire for types and typings containing detailed information on the uses of functions and data representations. Some type system designs conflict with these goals. For example, although universal and existential quantifiers are capable of representing

strong behavioral guarantees, they tend to inhibit access by the compiler to information on implementation decisions. As a result, a standard implementation method for languages with universal quantification is *boxing*, i.e., accessing every value that can not fit in a register through a pointer. Boxing is expensive due to run-time overhead and compile-time inhibition of optimization. The dynamic dispatch problem of object-oriented languages is essentially the same as boxing.

Implicit or deep subtyping can cause similar problems. Implicit subtyping fails to record decisions on the placement of coercions. A use of deep subtyping represents a potential coercion which modifies a value *at some other location in the program which may not even exist yet.* This interferes with optimization.

As an alternative to the approaches mentioned above, we have deliberately formulated our language to increase the *concrete* type information available to the compiler and to make typing decisions explicit instead of implicit. Thus, for handling code *polymorphism* and *abstractness*, we use the *finitary* intersection and union types instead of the *infinitary* universal and existential types. Finitary types allow typing as many or more terms as infinitary types.

Encoding Type Annotations: Intersection types are ordinarily implicitly typed using the following typing rule for introducing an intersection type:

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

As a result, for any subterm M in a typing, there may be multiple typing derivations. Thus, formulating explicit intersection types requires deciding (1) how to annotate the types of bound variables, (2) how to combine different typing annotations for the same term, and (3) how to determine if two different type annotations are for the same term. The new $\wedge$-introduction rule will look something like this:

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

The approach used by Reynolds in the language Forsythe [25] annotates the binding of an abstraction $(\lambda x.M)$ with a list of types as in $(\lambda x{:}\,\sigma_1 | \cdots | \sigma_n.M)$, requires the body M of the abstraction to be typable with the same type τ for each possible type σ_i of the bound variable x, and then assigns the abstraction 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. Pierce gives a more general approach using a special term-level construct to bind a type variable to some set of types [20]. For example, using this method the term $(\lambda x.\lambda y.x)$ could be annotated as $(\textbf{for } \alpha \in \{\sigma, \tau\}.\lambda x{:}\,\alpha.\lambda y{:}\,\alpha.x)$ to have 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 $(((\alpha \to \alpha) \wedge (\beta \to \beta)) \to \alpha \to \beta \to (\alpha \times \beta)) \wedge ((\gamma \to \gamma) \to \gamma \to \gamma \to (\gamma \times \gamma))$.

To provide a place for multiple conflicting type annotations, we altered the standard typing rule to "combine" the multiple type-annotated versions of a term by simply keeping both versions:

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

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* term. Given this choice, we can then use ordinary type annotations on variable bindings. For example, to give the term $\lambda x.x$ the type $(\sigma \to \sigma) \wedge (\tau \to \tau)$, we annotate it as $\wedge(\lambda x^{\sigma}.x^{\sigma}, \lambda x^{\tau}.x^{\tau})$.

One implication of our choice is that the tree structure of an explicitly typed term follows the tree structure of its typing proof instead of the tree structure of the untyped term which it represents. A difficulty this introduces is that reduction must essentially work on typing derivations, which is non-trivial to formulate. Wells [31] has developed an alternative formulation where typed and untyped terms have essentially the same tree structure, but the reduction rules are quite complex.

Difficulties with Union Types: It is difficult to formulate an implicitly typed calculus with union types which has the subject-reduction property. For an explicitly typed calculus, this problem manifests itself as a difficulty in guaranteeing that any computation that can be performed on an untyped program can be duplicated on a typed version of the same program. In an implicitly typed calculus, the $\vee$-elimination rule is usually formulated as:

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

With this formulation, the subject-reduction property is lost. Barbanera and Dezani-Ciancaglini give as an example the term $(\lambda x.\lambda y.\lambda z.x((\lambda t.t)yz)((\lambda t.t)yz))$, which 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 can not.

Since the $\vee$-elimination rule given above also causes other difficulties in formulating explicitly typed terms, it seems a solution to this might be to change the elimination rule to:

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

The same example above would still have a problem with this because one could just perform an extra β-reduction step. To solve the problem, it is sufficient to additionally require call-by-value reduction, if a variable is not considered a value. The base values are constants and abstractions and the set of values is closed under tuple and variant formation. This ensures that every reduction at the untyped level will have a corresponding reduction at the typed level.

4 Formal Language Definition

4.1 General Notation and Terminology

A *context* is a term containing holes. 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 $M_1, \ldots, M_n$ 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. The expression $\mathrm{FV}\,(M)$ denotes the set of free variables of M.

Our presentation generalizes *notions of reduction* (n.o.r.). 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 $\mathbf{C}_R$. 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 $i \in \{1, \ldots, k\}$ 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. Writing "$\longrightarrow\!\!\!\!\rightarrow_R$" denotes the transitive and reflexive closure of "$\longrightarrow_R$". A term M is in *normal form* with respect to R, written $R\text{-nf}\,(M)$, 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 $R\text{-nf}\,(N)$.

4.2 Untyped Language λ_u^{CIL}

Figure 1 shows the syntax and semantics of the untyped language λ_u^{CIL}.

Theorem 1 Confluence of Untyped Reduction. *If $\hat{M} \longrightarrow\!\!\!\!\rightarrow_{\hat{c}} \hat{N}_1$ and $\hat{M} \longrightarrow\!\!\!\!\rightarrow_{\hat{c}} \hat{N}_2$, then there exists $\hat{M}'$ such that $\hat{N}_1 \longrightarrow\!\!\!\!\rightarrow_{\hat{c}} \hat{M}'$ and $\hat{N}_2 \longrightarrow\!\!\!\!\rightarrow_{\hat{c}} \hat{M}'$.*

4.3 Explicitly Typed Language λ^{CIL}

Figure 2 shows the syntax of our explicitly typed language λ^{CIL}.

Although this presentation omits recursive types, they can be added by extending the types to regular trees. This causes no difficulties with the theorems given in this paper. Of course, a finite representation must be chosen, e.g., the usual $\mu\alpha.\tau$ syntax.

The type erasure $|C|$ of a type-annotated context C (defined in figure 2) is the corresponding untyped and unlabelled context. Some contexts do not have

Untyped Syntax

$$\begin{aligned}
\hat{C} \in \mathbf{UntContext} ::= {} & [\ ] \mid c \mid x \mid \mu x.\hat{C} \mid \lambda x.\hat{C} \mid \hat{C}_1 \mathbin{@} \hat{C}_2 \\
& \mid \times\left(\hat{C}_1, \ldots, \hat{C}_n\right) \mid \pi_i^{\times}\, \hat{C} \\
& \mid \mathbf{in}_i^{+} \hat{C} \mid \mathbf{case}^{+}\, \hat{C}\ \mathbf{bind}\ x\ \mathbf{in}\ \hat{C}_1, \ldots, \hat{C}_n \\
\hat{M}, \hat{N} \in \mathbf{UntTerm} = {} & \{\hat{C} \mid [\ ] \not\trianglelefteq \hat{C}\} \\
\hat{V} \in \mathbf{UntValue} ::= {} & c \mid \lambda x.\hat{M} \mid \times\left(\hat{V}_1, \ldots, \hat{V}_n\right) \mid \mathbf{in}_i^{+} \hat{V}
\end{aligned}$$

Untyped Reduction

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

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

Fig. 1. Untyped language λ_u^{CIL}.

a type erasure, i.e., those containing virtual tuples like $\wedge(C_1, \ldots, C_n)$ or virtual case expressions like

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

where the type erasures of $C_1, \ldots, C_n$ are not identical.

Figure 3 gives the typing rules of λ^{CIL}. 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 DomDef (A). A triple $A \vdash M : \tau$ is a *judgement*. A *derivation* $\mathcal{D}$ in language X is a sequence of judgements, each obtained from the previous ones by the typing rules of X. We write "$A \vdash_X M : \tau$ via $\mathcal{D}$" to mean 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 $A \vdash_{\lambda^{\mathrm{CIL}}} M : \tau$ means there exists some $\mathcal{D}$ such that $A \vdash_{\lambda^{\mathrm{CIL}}} M : \tau$ 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.

Theorem 2 Uniqueness of Typings in λ^{CIL}. *For $M \in$ **Term**, there is at most one type environment A and type τ such that* DomDef (A) = FV (M) *and* $A \vdash_{\lambda^{\mathrm{CIL}}} M : \tau$.

The call-by-value reduction rules for our typed language λ^{CIL} are in figure 4. The main notion of reduction, r-reduction, is divided into three steps: simplifying

Syntax Shared between Types and Terms

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

Types

$$\rho, \sigma, \tau \ ::= \ o \mid \sigma \xrightarrow[\psi]{\phi} \tau \mid Q[\tau_1, \ldots, \tau_n]$$

Type-Annotated Contexts

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

Type Erasure (a partial function)

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

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

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

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

Type-Annotated Terms, Values, Parallel Contexts

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

Syntactic Sugar for Examples

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

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

$(\mathbf{let}\ x^\tau = N\ \mathbf{in}\ M) \equiv \left(\left(\lambda^l_{\{k\}} x^\tau.M\right) \mathbin{@^{\{l\}}_k} N\right) \quad (\text{fresh } l, k)$

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

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

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

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

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

$$\text{($\times$,$\wedge$ elim)}\ \frac{A \vdash M : P[\tau_1, \ldots, \tau_n];\ 1 \leq i \leq n}{A \vdash \pi_i^P\, M : \tau_i} \qquad \text{(arrow-$\leq$)}\ \frac{\phi \subseteq \phi';\ \psi' \subseteq \psi}{\sigma \xrightarrow[\psi]{\phi} \tau \leq \sigma \xrightarrow[\psi']{\phi'} \tau}$$

$$\text{(+,$\vee$ intro)}\ \frac{A \vdash M : \tau_i;\ 1 \leq i \leq n}{A \vdash \left(\mathbf{in}_i^S\, M\right)^{S[\tau_1, \ldots, \tau_n]} : S[\tau_1, \ldots, \tau_n]}$$

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

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

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

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, it is convenient to go to t-normal form before and after computation steps. 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 **ParContext**), which require parallel c-redexes to fill holes that map to the same hole in the type-erased program.

Theorem 3 Subject Reduction for λ^{CIL}. *If $M \longrightarrow_r N$ and $A \vdash_{\lambda^{\text{CIL}}} M : \tau$, then $A \vdash_{\lambda^{\text{CIL}}} N : \tau$.*

Theorem 4 Typed/Untyped Reduction Correspondence.
If $A \vdash_{\lambda^{\text{CIL}}} M : \tau$, then

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

Main Notion of Reduction for Type-Annotated Terms

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

Computation Reduction

$$\begin{array}{lll}
(\lambda^l_\psi x^\tau.M)\ @^\phi_k\ V & \rightsquigarrow_c M\,[x{:=}V] & \\
\pi^\times_i \times(V_1,\ldots,V_n) & \rightsquigarrow_c V_i & \text{if } 1 \le i \le n \\
\mathbf{case}^+\ (\mathbf{in}^+_i\ V)^\tau\ \mathbf{bind}\ x\ \mathbf{in}\ \tau_1 \Rightarrow M_1,\ldots,\tau_n \Rightarrow M_n & \rightsquigarrow_c M_i\,[x{:=}V] & \text{if } 1 \le i \le n \\
\mu x^\tau.V & \rightsquigarrow_c V\,[x{:=}(\mu x^\tau.V)] &
\end{array}$$

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

Type-Annotation-Simplification Reduction

$$\begin{array}{lll}
\pi^\wedge_i\ \wedge(M_1,\ldots,M_n) & \rightsquigarrow_t M_i & \text{if } 1 \le i \le n \\
(\mathbf{case}^\vee\ (\mathbf{in}^\vee_i\ N)^\tau\ \mathbf{bind}\ x\ \mathbf{in} & \rightsquigarrow_t \left(\lambda^1_{\{1\}} x^{\tau_i}.M_i\right)\ @^{\{1\}}_1\ N & \text{if } 1 \le i \le n \\
\quad \tau_1 \Rightarrow M_1,\ldots,\tau_n \Rightarrow M_n) & & \\
(\mathbf{coerce}\,(\sigma,\tau)\,(\lambda^l_\psi x^\rho.M))\ @^\phi_k\ N & \rightsquigarrow_t \left(\lambda^l_{\{k\}} x^\rho.M\right)\ @^{\{l\}}_k\ N & \\
\mathbf{coerce}\,(\sigma_1,\tau)\ \mathbf{coerce}\,(\rho,\sigma_2)\,M & \rightsquigarrow_t \mathbf{coerce}\,(\rho,\tau)\,M &
\end{array}$$

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

Fig. 4. Reduction rules of explicitly typed language λ^{CIL}.

Theorem 5 Confluence of Typed Reduction. *If* $M \twoheadrightarrow_r N_1$ *and* $M \twoheadrightarrow_r N_2$, *then there exist* M_1' *and* M_2' *such that* $|M_1'| \equiv |M_2'|$ *and* $N_1 \twoheadrightarrow_r M_1'$ *and* $N_2 \twoheadrightarrow_r M_2'$.

4.4 Implicitly Typed Language λ_i^{CIL}

The implicitly typed language λ_i^{CIL} is automatically obtained from λ^{CIL} and λ_u^{CIL}. The syntax and semantics of λ_i^{CIL} are the same as λ_u^{CIL} as given in figure 1. The typing rules of λ_i^{CIL} are the rules of figure 3 modified by replacing every judgement $A \vdash M : \tau$ mentioned in a rule by $A \vdash |M| : \tau$, using the type erasure rules from figure 2.

Theorem 6 Subject Reduction for λ_i^{CIL}. *If* $\hat{M} \longrightarrow_{\hat{c}} \hat{N}$ *and* $A \vdash_{\lambda_i^{\text{CIL}}} \hat{M} : \tau$, *then* $A \vdash_{\lambda^{\text{CIL}}} \hat{N} : \tau$.

5 Related Work

Typed intermediate languages are used in several experimental compilers. Most typed intermediate languages for polymorphic programming languages can be seen as variants of the Girard/Reynolds λ-calculus, System F [9, 24].

Recent versions of the Standard ML of New Jersey (SML/NJ) compiler [3, 27] use a variant of system F as the representation in the front-end of the compiler. In

SML/NJ, type inference annotates polymorphic functions with universally quantified types and annotates function applications with the simple types to which the polymorphic types are instantiated. The compiler uses the type information to select efficient data representations and to minimize boxing coercions [16]. The SML/NJ compiler also uses *minimal typing derivations* [7] to reduce boxing coercions for let-polymorphic definitions. The compiler uses a simply typed representation in later stages of the compiler.

The Glasgow Haskell Compiler (GHC) [15] also uses a variant of System F. In GHC, type inference annotates polymorphic functions with type abstractions and uses of polymorphic functions with type arguments. This allows the compiler to preserve the well-typedness of the intermediate representation across program transformations. The type information is used in the later stages of the compiler to improve code generation.

System F can also be seen as the basis of the typed intermediate language λ_i^{ML} of the TIL compiler for Standard ML [18, 17]. The calculus λ_i^{ML} is a predicative variant of System F extended with intensional polymorphism [11]. The key feature is the support for dynamic type dispatch at run-time. This aids in efficient compilation of polymorphism without sacrificing separate compilation. A use of a polymorphic function can dispatch on a type argument to yield a monomorphic routine suitable for the type. This approach to compiling polymorphism yields excellent results [28] since many type dispatch redexes can be eliminated at compile-time and the compiler can then gain the resulting benefits of type specialization including in-lining and common subexpression elimination.

Our intermediate language λ^{CIL} was inspired by the earlier work on rank-2 intersection types of Jim [13]. As we have shown in this paper, intersection types naturally lead to a flow-directed approach to compilation. Our flow labels encode information about the operational behavior of the program that cannot be obtained from types without flow labels. At the same time, intersection and union types support a natural encoding of polyvariant flow information [5]. While it is clearly possible to compute, record, and use the flow and type information separately, we believe that a single representation is more natural for compilation.

General research into intersection types that has influenced our thinking includes the work of Van Bakel [4] and Jim [13]. Research on both intersection and union types that we have consulted includes the work by Pierce [20], Aiken, Wimmers, and Lakshman [1, 2], Barbanera and Dezani-Ciancaglini [6], and Trifonov and Smith [29]. Of the above, only Pierce considers intersection and union types in an explicitly typed language. Even that is somewhat distant from our work because Pierce includes a general subtyping relation on intersection and union types which we have deliberately avoided.

6 Conclusions and Future Work

We have presented λ^{CIL}, a typed intermediate language suitable for optimizing compilers for higher-order polymorphic programming languages such as ML. The

intermediate language is designed to facilitate verifiable flow-directed compiling. Based on λ^{CIL}, we have developed a framework for typed-directed flow-based representation transformations, and have illustrated this framework in a closure conversion application that supports multiple function representations, including the inlining of open functions [8]. This application (informally sketched in section 2) is an example of how λ^{CIL} supplies the compiler writer both important information and great flexibility in making optimization decisions.

Below, we outline some of the work ahead.

Labelling All Terms: In this presentation, only abstractions, applications, and function types were given flow labels. In order to track the flows of non-function values, it will be necessary to to annotate *all* terms and types in the language.

Compiling Polymorphism by Specialization: The λ^{CIL}-calculus suggests an approach to compiling polymorphism of flow-directed specialization. The number of specializations required for a given definition can be minimized if they are determined by representation types rather than source types. We are currently studying the issue of representation types.

Separate Compilation: If a program is compiled as a single unit, it is possible to express all instances of polymorphism and data abstraction in terms of intersection and union types. However, if a program is decomposed into separately compiled modules, universal and existential types may be necessary to model the module interfaces. λ^{CIL} will need to be extended in order to support separate compilation. Additionally, flow-directed specialization is difficult to extend to separately compiled modules. We are currently studying link-time specialization in which the linker determines whether new specializations of a definition are required.

Flow Analysis: The typed control flow analyses alluded to in this paper are limited by our shallow subtyping relation. We would like to weaken this restriction to permit more powerful control flow analysis algorithms.

Term Duplication: 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 using a typed calculus with intersection and union types in the style of [31].

References

1. 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.
2. A. S. Aiken, E. L. Wimmers, and T. K. Lakshman. Soft typing with conditional types. In POPL '94 [22], pp. 163–173.
3. A. W. Appel. *Compiling with Continuations.* Cambridge University Press, 1992.
4. S. van Bakel. *Intersection Type Disciplines in Lambda Calculus and Applicative Term Rewriting Systems.* PhD thesis, University of Nijmegen, 1993.

5. A. Banerjee. A modular, polyvariant, and type-based closure analysis. Manuscript, Nov. 1996.
6. F. Barbanera and M. Dezani-Ciancaglini. Intersection and union types: Syntax and semantics. *Information and Computation*, 119:202–230, 1995.
7. N. S. Bjørner. Minimal typing derivations. In *ACM SIGPLAN Workshop on ML and its Applications*, pp. 120–126, 1994.
8. A. Dimock, R. Muller, F. Turbak, and J. B. Wells. Strongly typed flow-directed representation transformations (extended abstract). Submitted. See http://www.cs.bu.edu/groups/church, Nov. 1996.
9. J.-Y. Girard. *Interprétation Fonctionnelle et Elimination des Coupures de l'Arithmétique d'Ordre Supérieur*. Thèse d'Etat, Université de Paris VII, 1972.
10. J. Hannan. Type systems for closure conversion. In *Workshop on Types for Program Analysis*, pp. 48–62, 1995. DAIMI PB-493.
11. R. Harper and G. Morrisett. Compiling polymorphism using intensional type analysis. In *Conf. Rec. 22nd Ann. ACM Symp. Principles of Programming Languages*, 1995.
12. N. Heintze. Control-flow analysis and type systems. In *Proc. 2nd Int'l Static Analysis Symp.*, pp. 189–206, 1995.
13. T. Jim. What are principal typings and what are they good for? In POPL '96 [23].
14. M. P. Jones. Dictionary-free overloading by partial evaluation. In *ACM SIGPLAN Workshop on Partial Eval. & Semantics-Based Prog. Manipulation*, 1994.
15. S. L. P. Jones. Compiling Haskell by program transformation: a report from the trenches. In *Proc. European Symp. on Programming*, 1996.
16. X. Leroy. Unboxed objects and polymorphic typing. In *Conf. Rec. 19th Ann. ACM Symp. Principles of Programming Languages*, pp. 177–188, 1992.
17. Y. Minamide, G. Morrisett, and R. Harper. Typed closure conversion. In POPL '96 [23].
18. G. Morrisett. *Compiling with Types*. PhD thesis, Carnegie Mellon University, 1995.
19. S. Peyton Jones and E. Meijer. Henk: A typed intermediate language. Submitted, Jan. 1997.
20. B. C. Pierce. Programming with intersection types, union types, and polymorphism. Technical Report CMU-CS-91-106, Carnegie Mellon University, Feb. 1991.
21. *Proc. ACM SIGPLAN '95 Conf. Prog. Language Design & Implementation*, 1995.
22. *Conf. Rec. 21st Ann. ACM Symp. Principles of Programming Languages*, 1994.
23. *Conf. Rec. POPL '96: 23rd ACM Symp. Principles of Prog. Languages*, 1996.
24. 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.
25. J. C. Reynolds. Design of the programming language Forsythe. In P. O'Hearn and R. D. Tennent, eds., *Algol-like Languages*. Birkhauser, 1996.
26. Z. Shao. *Compiling Standard ML for Efficient Execution on Modern Machines*. PhD thesis, Princeton University, 1994.
27. Z. Shao and A. Appel. A type-based compiler for Standard ML. In PLDI '95 [21].
28. D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: A type-directed optimizing compiler for ML. In PLDI '95 [21].
29. V. Trifonov and S. Smith. Subtyping constrained types. Revised Draft, May 1996.
30. M. Wand and P. Steckler. Selective and lightweight closure conversion. In POPL '94 [22], pp. 435–445.
31. J. B. Wells. Intersection types revisited in the Church style. Manuscript, June 1996.