Type-Directed Flow Analysis for Typed Intermediate Languages

Suresh Jagannathan, Stephen Weeks, and Andrew Wright

NEC Research Institute, 4 Independence Way, Princeton, NJ 08540

Abstract. Flow analysis is especially valuable for optimizing functional languages because control-flow information is not syntactically apparent in higher-order programs. Flow analyses typically operate on untyped languages. However, recent compilers for typed functional languages such as ML and Haskell use a *typed* higher-order intermediate language to expose data representations for optimization. This paper presents a polyvariant flow analysis framework for a typed intermediate language. Analyses in this framework can take advantage of types to analyze programs more precisely. We study a specific analysis called $S_{\mathcal{RT}}$ that uses types to control polyvariance. We prove that $S_{\mathcal{RT}}$ *respects types*: whenever it assigns abstract value $\hat{v}$ to a variable and the type system assigns type σ to the same variable, then $[\![\hat{v}]\!] \subseteq [\![\sigma]\!]$, where $[\![\cdot]\!]$ denotes a set of values.

1 Introduction

Recent compilers for functional languages such as ML [24] and Haskell [18] express most optimizations in a *typed* higher-order intermediate language. Typed intermediate languages serve two useful roles. First, they expose representation information valuable for code generation. For example, typed intermediate languages enable intensional polymorphism [9], a technology that eliminates the need to use a universal representation for polymorphic objects. Second, typed intermediate languages aid in verifying compiler optimizations, and help to identify compiler bugs by detecting code transformations that are not type-safe.

Flow analysis is especially valuable for optimizing functional languages because control-flow information is not syntactically apparent in higher-order programs. Flow analyses, however, typically operate on *untyped* intermediate languages. In this paper, we consider the design of flow analyses for typed intermediate languages. Incorporating flow analyses into a typed intermediate language framework improves optimizations in two respects. First, a flow analysis can exploit types to obtain more precise control-flow and data-flow information without undue computational cost. This added precision can improve the quality of optimizations that rely on global control-flow information. Second, flow analyses can discover information about types to facilitate type-dependent optimizations. Through intensional polymorphism, a program can build and operate on types at runtime. By using flow analysis to identify the paths along which types flow, a compiler can specialize or eliminate many of these type operations.

Monovariant analyses such as 0CFA [23] and set-based analysis [10] compute a map that associates an abstract value with each subexpression of the program.

An abstract value conservatively approximates the set of possible values the subexpression may take on during execution. *Polyvariant* analyses yield more precise results by associating several abstract values with each subexpression, each of which describes the expression's value during some subset of execution states. Existing flow analysis frameworks for untyped languages use syntactic notions such as call-strings [22] or other *ad hoc* heuristics [4, 10, 15] to distinguish different execution states. Surprisingly, even analyses used in compilers for typed languages ignore type information [3, 10].

Using types to control polyvariance[1] offers the pragmatic benefit that a flow analysis will not discard type information supplied by the programmer (whether explicit or inferred). Without using type information, the precision of the information obtained by flow analysis will not scale with program size. As a software system grows, polymorphic procedures such as map or fold will be called from many unrelated parts of the system with arguments of different type. A flow analysis that does not use type information to guide polyvariance will merge information across these unrelated calls, reducing the effectiveness of optimizations which depend upon its results. An analysis that uses type information is less likely to suffer degradation in the precision of flow information it computes as new, independent components are added to the program.

To illustrate how a flow analysis can exploit types to obtain more precise flow information, consider the following polymorphic procedure which uses a typecase construct [9] to perform a dispatch on its type argument:

```
letrec print = Λα.λx^α.
                   typecase α of
                     int ⇒ Integer.print x
                     real ⇒ Real.print x
                     β list ⇒ map (print β) x
in print int list [1,2,3];
   print int list [4,5];
   print real list [7.0,8.0,9.0]
end
```

Excluding the recursive call, print is applied at three different call-sites with two different types. A monovariant analysis computes a single abstract value describing all possible bindings to x, and hence merges values of types *int list*, *real list*, *int*, and *real*. Because the polyvariance in analyses such as 1CFA [23] and polymorphic splitting [15] depends on a program's syntactic structure, procedures which are supplied values of different type through deep or recursive call chains cause these analyses to introduce unnecessary merging. Hence, these analyses would merge values of different type at x. By using the types at which print is called to create distinct contexts in which to analyze its body, a polyvariant flow analysis can keep arguments and results of different types distinct.

[1] Our notion differs from systems [2, 6, 13, 25] which use let-polymorphism to formalize polyvariance. For a detailed discussion, see Section 4.3.

Using types to control polyvariance can also lead to more efficient analyses. Sufficiently complex polyvariant analyses based on call-strings (e.g., 2CFA [23]), let-polymorphism [2], or other syntactic heuristics might handle the above example without merging values of different types. But, such analyses would create separate contexts in which to analyze the first two calls to print. For many optimizations (*e.g.*, unboxing), analyzing these two calls in one shared context leads to equally useful flow information at lesser computational cost. An analysis that uses types to control polyvariance will recognize that the argument types are the same at each call and establish a single context to analyze both calls.

Type-directed flow analyses can also facilitate type-dependent optimizations. Without optimization, both the typecase operation within print and the type applications that bind *int list* to α in the calls to print would be performed at runtime. These runtime operations can severely impact the performance of programs that make significant use of polymorphism. Flow analysis can determine the contexts in which type arguments are constant, and guide specialization and inlining decisions to eliminate type operations. In this example, an implementation could specialize print for arguments of type *int list*, eliminate the typecase expression in the specialized function body, and elide the type arguments. While local transformations might address this simple example, optimizing more sophisticated higher-order uses of routines like print requires the power of flow analysis.

1.1 Outline

In the following section, we describe a parameterized flow analysis framework for a typed intermediate language Λ_i. Section 3 defines a specific analysis called $S_{\mathcal{RT}}$ that uses types to control polyvariance. Essentially, $S_{\mathcal{RT}}$ constructs a distinct polyvariance context for each type at which a polymorphic function is used. We prove that $S_{\mathcal{RT}}$ *respects types*: whenever it assigns abstract value $\hat{v}$ to a variable and the type system assigns type σ to the same variable, then $[\![\hat{v}]\!] \subseteq [\![\sigma]\!]$, where $[\![\cdot]\!]$ denotes a set of values. In Section 4, we define several variants of $S_{\mathcal{RT}}$ that are better suited to practical use. We conclude with comparisons to related work.

2 Flow Analysis for Typed Languages

We study a simple core language called Λ_i suitable for compiling languages such as ML. Λ_i extends the predicative subset of system $\mathcal{F}$ [7, 20] with recursive procedures. In Λ_i, as in the predicative subset of $\mathcal{F}$, polymorphic functions cannot be applied to quantified types.

2.1 Language

The expressions and types of Λ_i are defined as follows:

$$\begin{array}{ll}
e ::= f \mid x \mid e\ e \mid e\ \tau \mid \mu x^{\sigma}.f & \text{(expressions)} \\
f ::= \lambda x^{\sigma}.e \mid \Lambda\alpha.e & \text{(functions)} \\
\\
\kappa ::= b \mid \kappa \to \kappa & \text{(constructors)} \\
\tau ::= b \mid \tau \to \tau \mid \alpha & \text{(types)} \\
\sigma ::= b \mid \sigma \to \sigma \mid \alpha \mid \forall\alpha.\sigma & \text{(type schemes)}
\end{array}$$

where x and α range over variables and type variables, respectively, and b ranges over base types. Constructors (closed monotypes) are a subset of types, which are in turn a subset of type schemes. The expression $\lambda x^{\sigma}.e$ binds x of type σ in e and provides lexically-scoped, call-by-value procedures. The expression $\Lambda\alpha.e$ binds α in e and provides type abstraction. The expression $\mu x^{\sigma}.f$ binds x in f and provides recursive procedures. The type scheme $\forall\alpha.\sigma$ binds α in σ and provides polymorphic types. These constructs induce the usual notions of free variables (FV) and free type variables (FTV) of expressions and type schemes. We use *ExpOcc*, *TypeOcc*, *FunExpOcc*, and *AppExpOcc* to refer to occurrences of expressions, types, functions, and applications in a given program P. All occurrences are distinct, except that we conflate the bound occurrences of a variable with its binding occurrence. Programs are closed, well-typed expressions. Section 3 presents the usual typing rules for Λ_i and a call-by-value semantics.

2.2 Flow Analysis Framework

The parameters that specify a particular analysis within our framework are:

1. a set of *instances* I,
2. an initial instance $i_0 \in I$, and
3. a *call map* $M : \mathit{FunExpOcc} \times I \times \mathit{AppExpOcc} \times I \to I$.

We refer to a specific analysis as a *strategy* $S = \langle I, i_0, M\rangle$. An instance corresponds to a set of evaluation contexts. A call map describes which instance is selected for the analysis of a procedure body, depending on the function being applied, the instance in effect when the function was constructed, the text of the call expression, and the instance in effect at the call. For the remainder of this section, we assume S is fixed.

For a given strategy, the results of an analysis are expressed as a *flow function* F that maps expression-instance pairs to *abstract values.*

$$\begin{array}{rl}
F \in \mathit{Flow} & = (\mathit{ExpOcc} \times I \to \mathit{Avalue})\ + (\mathit{TypeOcc} \times I \to \mathit{Atype}) \\
\hat{v} \in \mathit{Avalue} & = \mathcal{P}(\mathit{Aclosure}) \\
\langle f, i\rangle \in \mathit{Aclosure} & = \mathit{FunExpOcc} \times I \\
\hat{\tau} \in \mathit{Atype} & = \mathcal{P}(\mathit{Aconstructor}) \\
\langle \tau, i\rangle \in \mathit{Aconstructor} & = \mathit{TypeOcc} \times I
\end{array}$$

An abstract value $\hat{v}$ is a set of abstract closures. An abstract closure $\langle f, i \rangle$ consists of a function occurrence paired with an instance. The instance is used to find abstract values and abstract types for free variables and free type variables of the function body. For example, if abstract closure $\langle f, i \rangle$ has x free in f, the abstract value for x is $F(x, i)$. Abstract types are sets of abstract constructors, which consist of a type occurrence paired with an instance. As with closures, the instance is used to find abstract types for free type variables of the type expression. Section 3 formally defines the meaning of abstract values and types.

To compute a flow function F for a program, we define an algorithmic notion of *safety* that imposes constraints on F. First, we say that F is *initialized* for occurrence o in instance i if:

1. $o = f$ then $\langle f, i \rangle \in F(f, i)$;
2. $o = \tau$ then $\langle \tau, i \rangle \in F(\tau, i)$;
3. $o = \mu x^{\sigma}.f$ then $\langle f, i \rangle \in F(\mu x^{\sigma}.f, i)$ and $\langle f, i \rangle \in F(x, i)$;
4. $o = x$;
5. $o = (o_1\ o_2)$ then F is initialized for o_1 in i.

Initialization ensures that the flow function includes an appropriate abstract closure or abstract constructor for a function or type expression that could be reached in instance i. Safety then specifies certain set containment constraints that "pass" and "return" these initial abstract values and abstract types to any closures that arise at the function position of an application.

Definition 1. Flow F is *safe* for program P if:

1. F is initialized for P in i_0.
2. For all applications $(e\ o)$ in P and instances i,
   (a) if $F(e, i) \neq \emptyset$ then F is initialized for o in i;
   (b) if $\langle f, i' \rangle \in F(e, i)$, $f = \lambda z^{\sigma}.e'$ or $f = \Lambda z.e'$, and $i'' = M(f, i', (e\ o), i)$ then
      i. F is initialized for e' in i'';
      ii. $F(o, i) \subseteq F(z, i'')$;
      iii. $F(e', i'') \subseteq F((e\ o), i)$;
      iv. $F(w, i') \subseteq F(w, i'')$ for every $w \in FV(f) \cup FTV(f)$.

These constraints ensure that any flow function which is safe for program P provides a conservative approximation to the behavior of P (see Appendix A for details).

For a specific program, there are many safe flow functions. The pointwise ordering on flow functions allows us to compare different flows for the same program. The following lemma establishes that there exists a *minimum* safe flow under such an ordering.

Lemma 2. *For any program P, there exists a minimum flow F_{minsafe} that is safe for P.*

```
let id = Λα.λx^α. x
    g = Λβ.λy^β. id β y
in  g int→int f₁; g bool→bool f₂;
```

Fig. 1. Example program.

$$
\begin{aligned}
I_0 &= \{\bullet\} \\
i_0 &= \bullet \\
M_0(\lambda x^{\sigma}.e, i', (e_1\, e_2), i) &= \bullet \\
M_0(\Lambda\alpha.e, i', (e\,\tau), i) &= \bullet
\end{aligned}
$$

$$
\begin{aligned}
F_0(\alpha,\bullet) = F_0(\beta,\bullet) &= \{\langle \mathit{int}{\to}\mathit{int},\bullet\rangle,\ \langle \mathit{bool}{\to}\mathit{bool},\bullet\rangle\} \\
F_0(\mathsf{x},\bullet) = F_0(\mathsf{y},\bullet) &= \{\langle f_1,\bullet\rangle,\ \langle f_2,\bullet\rangle\} \\
F_0(\mathsf{g}\ \mathit{int}{\to}\mathit{int}\ f_1,\bullet) &= \{\langle f_1,\bullet\rangle,\ \langle f_2,\bullet\rangle\} \\
F_0(\mathsf{g}\ \mathit{bool}{\to}\mathit{bool}\ f_2,\bullet) &= \{\langle f_1,\bullet\rangle,\ \langle f_2,\bullet\rangle\}
\end{aligned}
$$

Fig. 2. Strategy $S_{0\mathrm{CFA}}$ and fragment of minimum safe flow for Figure 1.

Proof Sketch. We first show that there is a trivial safe flow which maps every element of its domain to a maximal abstract value. This maximal abstract value includes every possible abstract closure for P. We then show that safety is preserved by (possibly infinite) intersection of flows.

We can compute the minimum safe flow using a variant of standard constraint solving algorithms. The algorithms terminate provided that the minimum safe flow is finite.

2.3 An Example: 0CFA

To illustrate our framework, consider the program in Figure 1.[2] Here, id is a polymorphic identity function, f_1 is a function of type $\mathit{int}{\to}\mathit{int}$, and f_2 is a function of type $\mathit{bool}{\to}\mathit{bool}$. Figure 2 presents a strategy $S_{0\mathrm{CFA}} = \langle I_0, i_0, M_0\rangle$ and a fragment of the minimum safe flow under $S_{0\mathrm{CFA}}$ for the program in Figure 1. $S_{0\mathrm{CFA}}$ is monovariant because there is only one instance of any expression. Since it merges values of different types at several program points, we say $S_{0\mathrm{CFA}}$ does not respect types. For example, f_1 which is of type $\mathit{int}{\to}\mathit{int}$ arrives at expression "g $\mathit{bool}{\to}\mathit{bool}$ f_2" which is of type $\mathit{bool}{\to}\mathit{bool}$. In the next section, we develop a strategy that is better suited to analyzing Λ_i.

[2] We use let $x = e_1$ in e_2 to abbreviate $((\lambda x.e_2)\, e_1)$, and $(e_1;\, e_2)$ to abbreviate $((\lambda x.e_2)\, e_1)$ where $x \notin FV(e_2)$.

3 Polyvariance Using Types

We say that a flow analysis *respects types* if $[\![\hat{v}]\!] \subseteq [\![\sigma]\!]$ whenever the analysis associates abstract value $\hat{v}$ with a variable and the type system assigns type scheme σ to the same variable. To formalize this intuition, we must define the set of values $[\![\hat{v}]\!]$ associated with abstract value $\hat{v}$ and the set of values $[\![\sigma]\!]$ associated with type scheme σ. Although we could introduce denotational models for both abstract values and types and compare denotations, we prefer a simpler operational approach. We define a natural semantics for Λ_i in which values are closures $\langle f^\sigma, E, CE, i\rangle$, where E is an environment mapping free variables of f to values, CE is a constructor environment mapping free type variables of f to constructors, and i is the strategy's approximation of these environments.

We define $[\![\hat{v}]\!]$ as follows:

$$[\![\hat{v}]\!] = \{v \mid v \sqsubseteq \hat{v}\}$$

where $\sqsubseteq$ is inductively defined as follows:

1. $\langle f, E, CE, i\rangle \sqsubseteq \hat{v}$ if $\langle f, i\rangle \in \hat{v}$ and $E \sqsubseteq i$ and $CE \sqsubseteq i$;
2. $\langle \mu x^\sigma\!.f, E, CE, i\rangle \sqsubseteq \hat{v}$ if $\langle f, i\rangle \in F(x, i)$ and $\langle f, i\rangle \in \hat{v}$ and $E \sqsubseteq i$ and $CE \sqsubseteq i$;
3. $\kappa \sqsubseteq \hat{\tau}$ if there exists $\langle \tau, i\rangle \in \hat{\tau}$ such that $CE \sqsubseteq i$ and $CE(\tau) = \kappa$;
4. $E \sqsubseteq i$ if $E(x) \sqsubseteq F(x, i)$ for all $x \in dom(E)$;
5. $CE \sqsubseteq i$ if $CE(\alpha) \sqsubseteq F(\alpha, i)$ for all $\alpha \in dom(CE)$.

When F is not clear from context, we write $\sqsubseteq_F$. For $\langle f, E, CE, i\rangle$ to belong to the set corresponding to $\hat{v}$, the first clause of this definition requires $\hat{v}$ to include abstract closure $\langle f, i\rangle$ and $\sqsubseteq$ to hold for the free variables and free type variables of the closure. The next two clauses impose similar constraints for recursive procedures and type expressions. The last two clauses extend $\sqsubseteq$ pointwise to value environments and constructor environments.

We now turn our attention to describing the set of values $[\![\sigma]\!]$ associated with type scheme σ. Figure 3 defines the usual typing rules for Λ_i. A judgment $TE \triangleright e : \sigma$ indicates that expression e possesses type scheme σ in type environment TE. Type environments map variables to type schemes. The notation $\sigma[\alpha/\tau]$ denotes the substitution of τ for free occurrences of α in σ. When e is a subexpression of a program P, we write e^σ to indicate that there exists a type judgment $TE \triangleright e : \sigma$ in the type derivation for P.

We define $[\![\sigma]\!]$ to be the set of values that expand to closed expressions of closed type σ:

$$[\![\sigma]\!] = \{\langle f, E, CE, i\rangle \mid \phi \triangleright \langle\!\langle f, E, CE\rangle\!\rangle : \sigma\}$$

Expansion is defined as follows:

$$\begin{aligned}
\langle\!\langle \lambda x^\sigma.e, E, CE\rangle\!\rangle &= \lambda x^{CE(\sigma)}.\langle\!\langle e, E, CE\rangle\!\rangle \\
\langle\!\langle \Lambda\alpha.e, E, CE\rangle\!\rangle &= \Lambda\alpha.\langle\!\langle e, E, CE\rangle\!\rangle \\
\langle\!\langle \mu x^\sigma\!.f, E, CE\rangle\!\rangle &= \mu x^{CE(\sigma)}\!.\langle\!\langle f, E, CE\rangle\!\rangle \\
\langle\!\langle (e_1\, e_2), E, CE\rangle\!\rangle &= (\langle\!\langle e_1, E, CE\rangle\!\rangle\, \langle\!\langle e_2, E, CE\rangle\!\rangle) \\
\langle\!\langle (e\, \sigma), E, CE\rangle\!\rangle &= (\langle\!\langle e, E, CE\rangle\!\rangle\, CE(\sigma)) \\
\langle\!\langle x, E, CE\rangle\!\rangle &= \begin{cases} \langle\!\langle f', E', CE'\rangle\!\rangle & \text{if } x \in \mathrm{dom}(E) \text{ and } E(x) = \langle f', E', CE', i'\rangle \\ x & \text{otherwise.} \end{cases}
\end{aligned}$$

$$TE \triangleright x : TE(x) \qquad (\mathbf{var}_{\triangleright})$$

$$\frac{TE[x \mapsto \sigma_1] \triangleright e : \sigma_2}{TE \triangleright \lambda x^{\sigma_1}.e : \sigma_1 \to \sigma_2} \qquad (\mathbf{fun}_{\triangleright})$$

$$\frac{TE \triangleright e : \sigma \qquad \alpha \notin FTV(TE)}{TE \triangleright \Lambda\alpha.e : \forall\alpha.\sigma} \qquad (\mathbf{tfun}_{\triangleright})$$

$$\frac{TE \triangleright e_1 : \sigma_1 \to \sigma_2 \qquad TE \triangleright e_2 : \sigma_1}{TE \triangleright (e_1\ e_2) : \sigma_2} \qquad (\mathbf{app}_{\triangleright})$$

$$\frac{TE \triangleright e : \forall\alpha.\sigma}{TE \triangleright (e\ \tau) : \sigma[\alpha/\tau]} \qquad (\mathbf{tapp}_{\triangleright})$$

$$\frac{TE[x \mapsto \sigma] \triangleright f : \sigma}{TE \triangleright \mu x^{\sigma}.f : \sigma} \qquad (\mathbf{fix}_{\triangleright})$$

Fig. 3. Typing rules for Λ_i.

To establish a correspondence between instances and different stages of program execution, Figure 4 defines a semantics that manipulates instances explicitly as a component of the judgment.[3] A judgment $E, CE, i \vdash e \Rightarrow v$ indicates that expression e evaluates to v in environment E and constructor environment CE, under instance i. The notation $X|_{FV(\lambda x^{\sigma}.e)}$ means the restriction of environment X to the free variables of $\lambda x^{\sigma}.e$. We write $CE(\tau)$ to denote the substitution of constructors for free type variables in τ according to CE. The function $\rightsquigarrow$ unrolls recursive closure values prior to their application, and is the identity otherwise. Program P yields answer v if there exists a derivation $\mathcal{D}_P$ concluding $\phi, \phi, i_0 \vdash P \Rightarrow v$.

With these definitions, we can directly relate the sets of values described by abstract values to the sets of values described by types.

Definition 3 (Flow Respects Types). Flow function F for program P *respects types* if for every judgment $E, CE, i \vdash e^{\sigma} \Rightarrow v'$ in the derivation for P, we have $[\![F(e,i)]\!] \subseteq [\![CE(\sigma)]\!]$.

Definition 4 (Strategy Respects Types). Strategy S *respects types* if, for all programs P, the minimum safe flow F_{minsafe} for P under S respects types.

Figure 5 defines a strategy $S_{\mathcal{RT}}$ in which instances are partial functions that map type variables to constructors. The initial instance is the empty environment, ϕ. At a type application, the call map extends the instance i' of the type abstraction with a binding for the type variable α by treating i as a substitution applied to τ. Figure 5 also illustrates a fragment of the minimum safe flow

[3] While at first glance this semantics may appear unusual, erasing instances and uses of M from judgments and values yields an ordinary natural semantics. Instances play no role in determining how expressions are evaluated.

$$E, CE, i \vdash x \Rightarrow E(x) \qquad (\mathbf{var}_\vdash)$$

$$E, CE, i \vdash \tau \Rightarrow CE(\tau) \qquad (\mathbf{type}_\vdash)$$

$$E, CE, i \vdash \lambda x^\sigma\!.e \Rightarrow \langle \lambda x^\sigma\!.e, E|_{FV(\lambda x^\sigma.e)}, CE|_{FTV(\lambda x^\sigma.e)}, i\rangle \qquad (\mathbf{fun}_\vdash)$$

$$E, CE, i \vdash \Lambda\alpha.e \Rightarrow \langle \Lambda\alpha.e, E|_{FV(\Lambda\alpha.e)}, CE|_{FTV(\Lambda\alpha.e)}, i\rangle \qquad (\mathbf{tfun}_\vdash)$$

$$E, CE, i \vdash \mu x^\sigma\!.f \Rightarrow \langle \mu x^\sigma\!.f, E|_{FV(\mu x^\sigma.f)}, CE|_{FTV(\mu x^\sigma.f)}, i\rangle \qquad (\mathbf{fix}_\vdash)$$

$$\frac{\begin{array}{c} E, CE, i \vdash e_1 \Rightarrow v_1 \qquad E, CE, i \vdash e_2 \Rightarrow v_2 \\ v_1 \leadsto \langle \lambda x^\sigma\!.e', E', CE', i'\rangle \qquad E'[x \mapsto v_2], CE', i'' \vdash e' \Rightarrow v \\ i'' = M(\lambda x^\sigma\!.e', i', (e_1\ e_2), i) \end{array}}{E, CE, i \vdash (e_1\ e_2) \Rightarrow v} \qquad (\mathbf{app}_\vdash)$$

$$\frac{\begin{array}{c} E, CE, i \vdash e \Rightarrow v_1 \qquad E, CE, i \vdash \tau \Rightarrow \kappa \\ v_1 \leadsto \langle \Lambda\alpha.e', E', CE', i'\rangle \qquad E', CE'[\alpha \mapsto \kappa], i'' \vdash e' \Rightarrow v \\ i'' = M(\Lambda\alpha.e', i', (e\ \tau), i) \end{array}}{E, CE, i \vdash (e\ \tau) \Rightarrow v} \qquad (\mathbf{tapp}_\vdash)$$

$$\begin{array}{l} \langle f, E, CE, i\rangle \leadsto \langle f, E, CE, i\rangle \\ \langle \mu x^\sigma\!.f, E, CE, i\rangle \leadsto \langle f, E[x \mapsto \langle \mu x^\sigma\!.f, E, CE, i\rangle], CE, i\rangle \end{array}$$

Fig. 4. Semantics for Λ_i.

$F_{\mathcal{RT}}$ for Figure 1. Unlike $S_{0\text{CFA}}$, $S_{\mathcal{RT}}$ avoids merging the abstract closures for f_1 and f_2 which have different type. This strategy respects types because instances collect the actual types manipulated by the program at runtime: the values of free type variables in type expressions are substituted at type applications, and instances are extended with constructors when computing new instances. These characteristics ensure that the flow function preserves all bindings in a call chain of type applications.

Theorem 5. *$S_{\mathcal{RT}}$ respects types.*

For a proof of this theorem, see Appendix B.

Unfortunately, there are programs for which the minimum safe flow constructed under $S_{\mathcal{RT}}$ is not finite. Consider the following program.

$$\begin{array}{l} \mathsf{let\ loop} = \mu \mathsf{f}^{\forall\alpha.\alpha\to int}.\,\Lambda\alpha.\lambda \mathsf{x}^\alpha.\ \mathsf{f}\ int\to\alpha\ (\lambda \mathsf{y}^{int}.\mathsf{x}) \\ \mathsf{in\ \ loop}\ int\to int\ (\lambda \mathsf{z}^{int}.\mathsf{z}) \end{array}$$

During execution, this program constructs an infinite number of types. Hence its minimum safe flow under $S_{\mathcal{RT}}$ contains the following infinite set of instances for the body of loop:

$$\begin{array}{l} [\alpha \mapsto int\to int] \\ [\alpha \mapsto int\to int\to int] \\ [\alpha \mapsto int\to int\to int\to int] \\ \ldots \end{array}$$

In the next section, we define several variants of $S_{\mathcal{RT}}$ to address this problem.

$$
\begin{aligned}
I_{\mathcal{RT}} &= \mathit{Tyvar} \rightarrow \mathit{Constructor} \\
i_{\mathcal{RT}} &= \phi \\
M_{\mathcal{RT}}(\lambda x^{\sigma}.e, i', (e_1\, e_2), i) &= i' \\
M_{\mathcal{RT}}(\Lambda\alpha.e, i', (e\ \tau), i) &= i'[\alpha \mapsto i(\tau)]
\end{aligned}
$$

$$
\begin{aligned}
F_{\mathcal{RT}}(\mathsf{id}\ \beta, [\beta \mapsto \mathit{int}\rightarrow\mathit{int}]) &= \{\langle \lambda \mathsf{x}^{\alpha}.\mathsf{x}, [\alpha \mapsto \mathit{int}\rightarrow\mathit{int}]\rangle\} \\
F_{\mathcal{RT}}(\mathsf{id}\ \beta, [\beta \mapsto \mathit{bool}\rightarrow\mathit{bool}]) &= \{\langle \lambda \mathsf{x}^{\alpha}.\mathsf{x}, [\alpha \mapsto \mathit{bool}\rightarrow\mathit{bool}]\rangle\} \\
F_{\mathcal{RT}}(\beta, [\beta \mapsto \mathit{int}\rightarrow\mathit{int}]) &= \{\langle \mathit{int}\rightarrow\mathit{int}, \phi\rangle\} \\
F_{\mathcal{RT}}(\beta, [\beta \mapsto \mathit{bool}\rightarrow\mathit{bool}]) &= \{\langle \mathit{bool}\rightarrow\mathit{bool}, \phi\rangle\}
\end{aligned}
$$

$$
\begin{array}{ll}
F_{\mathcal{RT}}(\mathsf{x}, [\alpha \mapsto \mathit{int}\rightarrow\mathit{int}]) = \{\langle f_1, \phi\rangle\} & F_{\mathcal{RT}}(\mathsf{x}, [\alpha \mapsto \mathit{bool}\rightarrow\mathit{bool}]) = \{\langle f_2, \phi\rangle\} \\
F_{\mathcal{RT}}(\mathsf{y}, [\alpha \mapsto \mathit{int}\rightarrow\mathit{int}]) = \{\langle f_1, \phi\rangle\} & F_{\mathcal{RT}}(\mathsf{y}, [\alpha \mapsto \mathit{bool}\rightarrow\mathit{bool}]) = \{\langle f_2, \phi\rangle\} \\
F_{\mathcal{RT}}(\mathsf{g}\ \mathit{int}\rightarrow\mathit{int}\ f_1, \phi) = \{\langle f_1, \phi\rangle\} & F_{\mathcal{RT}}(\mathsf{g}\ \mathit{bool}\rightarrow\mathit{bool}\ f_2, \phi) = \{\langle f_2, \phi\rangle\}
\end{array}
$$

Fig. 5. Strategy $S_{\mathcal{RT}} = \langle I_{\mathcal{RT}}, i_{\mathcal{RT}}, M_{\mathcal{RT}}\rangle$ and minimum safe flow for Figure 1.

4 Variants of $S_{\mathcal{RT}}$

In this section, we present several variants of $S_{\mathcal{RT}}$. First, we define a family of strategies related to $S_{\mathcal{RT}}$ that bound the sizes of types they consider. Second, we identify a subset of Λ_i sufficient to express the core of ML and show that $S_{\mathcal{RT}}$ terminates on every program in this subset. Third, we define a strategy that simulates let-polymorphism, and compare it to $S_{\mathcal{RT}}$. Finally, we show how to combine a type-respecting analysis with any other analysis to produce an analysis that respects types.

In order to compare various strategies, we define a preorder on strategies which relates the instances of one strategy to the instances of another by a homomorphism.

Definition 6. Let strategy $S = \langle I, i_0, M\rangle$ and strategy $S' = \langle I', i'_0, M'\rangle$. Strategy S is more precise than strategy S' under homomorphism H, written $S \geq_H S'$, if there exists a function $H : I \rightarrow I'$ such that:

1. $H(i_0) = i'_0$, and
2. $H(M(f, i_1, (e\ o), i_2)) = M'(f, H(i_1), (e\ o), H(i_2))$.

We write $S \geq S'$ to mean that there exists H such that $S \geq_H S'$. $S_{0\text{CFA}}$ (the strategy in Figure 2) is the least element (up to isomorphism) of this preorder. If $S \geq S'$, the flow functions produced by S are more precise than the flow functions produced by S'. See Appendix **??** for further details.

4.1 Bounded Types

To construct an analysis that terminates for all Λ_i programs, we limit the sizes of constructors that the analysis uses in instances. We view constructors as binary trees and inductively define a family of sets indexed by integers. $\mathit{Constructor}_n$ is the set of constructors of depth less than or equal to n, where leaves are either base types b or $\bullet$, indicating that the type has been truncated:

$$\begin{aligned} \mathit{Constructor}_0 &= \{\bullet\} \\ \mathit{Constructor}_1 &= \mathit{Constructor}_0 \cup \{b\} \\ \mathit{Constructor}_{n+2} &= \mathit{Constructor}_{n+1} \cup \{\kappa \rightarrow \kappa' \mid \kappa, \kappa' \in \mathit{Constructor}_{n+1}\} \end{aligned}$$

Next, we define a family of functions Prune_n such that $\mathit{Prune}_n(\kappa)$ is the constructor formed by pruning branches of κ having depth greater than n:

$$\begin{aligned} \mathit{Prune}_0(\bullet) &= \bullet & \mathit{Prune}_{n+1}(\bullet) &= \bullet \\ \mathit{Prune}_0(b) &= \bullet & \mathit{Prune}_{n+1}(b) &= b \\ \mathit{Prune}_0(\kappa \rightarrow \kappa') &= \bullet & \mathit{Prune}_{n+1}(\kappa \rightarrow \kappa') &= \mathit{Prune}_n(\kappa) \rightarrow \mathit{Prune}_n(\kappa') \end{aligned}$$

Finally, we define a family of strategies $S^n_{\mathcal{RT}} = \langle I_n, i_n, M_n \rangle$ that only uses constructors from $\mathit{Constructor}_n$:

$$\begin{aligned} I_n &= \mathit{Tyvar} \rightarrow \mathit{Constructor}_n \\ i_n &= \phi \\ M_n(\lambda x^\sigma.e, i', (e_1\, e_2), i) &= i' \\ M_n(\Lambda\alpha.e, i', (e\ \tau), i) &= i'[\alpha \mapsto \mathit{Prune}_n(i(\tau))] \end{aligned}$$

For any depth n, the set of instances used by $S^n_{\mathcal{RT}}$ is finite, hence its minimum safe flow is finite. However, because $S^n_{\mathcal{RT}}$ uses function spaces as instances, its complexity is exponential, even though the size of constructors is limited to a constant. $S^n_{\mathcal{RT}}$ respects types for programs that use types of depth less than n. Since programs rarely build types of great depth, $S^n_{\mathcal{RT}}$ will behave the same as $S_{\mathcal{RT}}$ for all but the smallest values of n.

We can show that strategy $S_{\mathcal{RT}}$ is more precise than $S^n_{\mathcal{RT}}$ (*i.e.*, $S_{\mathcal{RT}} \geq S^n_{\mathcal{RT}}$, as the required homomorphism simply applies Prune_n to the type constructors of $S_{\mathcal{RT}}$). Likewise, we can show that $S^n_{\mathcal{RT}}$ is more precise than $S^{n-1}_{\mathcal{RT}}$.

4.2 Restricting Λ_i to ML

The following subset of Λ_i, which we call ML_i, is general enough to express the core of ML:

$$e ::= (x\, \boldsymbol{\tau}) \mid \lambda x^\tau.e \mid (e_1\, e_2) \mid (\lambda x^\sigma.e\ \ \Lambda\boldsymbol{\alpha}.e') \mid \mu x_1^{\tau_1}.\lambda x_2^{\tau_2}.e$$

where $(x\, \boldsymbol{\tau})$ abbreviates $(\ldots(x\, \tau_1)\ldots \tau_n)$ and $\Lambda\boldsymbol{\alpha}.e$ abbreviates $\Lambda\alpha_1.\ldots.\Lambda\alpha_n.e$. Applications of the form $(\lambda x^\sigma.e\ \ \Lambda\boldsymbol{\alpha}.e')$ serve as polymorphic let-expressions. The grammar ensures that polymorphic functions only appear in let-expressions, and are fully instantiated when used. For a translation of ML into ML_i, see [8].

Because of the restrictions on polymorphic functions, programs in ML_i can construct only a finite number of types during evaluation. Hence we can show that the minimum safe flow constructed under $S_{\mathcal{RT}}$ is finite for programs in ML_i. As with $S^n_{\mathcal{RT}}$ however, the complexity of $S_{\mathcal{RT}}$ on ML_i is exponential because of its use of a function space for instances.

Theorem 7. *The minimum safe flow for $P \in ML_i$ under $S_{\mathcal{RT}}$ is finite.*

For a proof of this theorem, see Appendix B. This theorem ensures we can compute a flow function that respects types for any program expressible in the ML core.

4.3 Polyvariance and Polymorphism

A common way to incorporate polyvariance into analyses of functional languages is to use let-polymorphism [2, 6, 13, 25]. Our approach differs in several respects.

Type systems that use let-polymorphism to express polyvariance encode a fixed strategy that is equivalent to expanding all let-expressions in the program and running a monovariant analysis. In contrast, our framework allows both coarser-grained (less precise) and finer-grained (more precise) forms of polyvariance. In particular, $S_{\mathcal{RT}}$ yields a coarser-grained analysis than let-expansion, and $S^n_{\mathcal{RT}}$ is coarser still. Finer-grained analyses may involve significantly greater computational effort—effort that is wasted if the increased precision is unnecessary for the intended optimizations.

Instead of using a type system to specify the analysis, we use an explicitly defined predicate (Definition 1, Safety). The solution algorithm for this predicate smoothly accommodates conditional constraints (*e.g.*, 2a in Definition 1). In contrast, conditional constraints complicate type systems and pose significant difficulties for type inference algorithms [1].

Finally, for programs outside the ML_i subset, analyses that express polyvariance using let-polymorphism analyze functions that are not bound by let-expressions in a monovariant manner. That is, polyvariance is introduced only at let-expressions. In constrast, our framework allows polymorphic functions to be passed as arguments, and permits any call to be treated polyvariantly.

We can express the let-polymorphism approach in our framework as the following strategy, called S_{LET}.

$$\begin{aligned}
I_{\text{LET}} &= \mathit{Tyvar} \rightarrow \mathit{AppExpOcc} \\
i_{\text{LET}} &= \phi \\
M_{\text{LET}}(\lambda x^{\sigma}.e, i, (e_1\, e_2), i') &= i \\
M_{\text{LET}}(\lambda x^{\sigma}.e, i, (\lambda x^{\sigma}.e \;\; \Lambda\boldsymbol{\alpha}.e'), i') &= i \\
M_{\text{LET}}(\Lambda\alpha.e, i, (x\, \boldsymbol{\tau}), i') &= i[\alpha \mapsto (x\, \boldsymbol{\tau})]
\end{aligned}$$

in the 2nd last rule, i' subsumes i?

Instances in S_{LET} are partial functions that map type variables to *occurrences* of type applications. Thus different uses of a polymorphic function, even if applied to the same type, will be evaluated in different instances.

Because $S_{\mathcal{RT}}$ analyzes uses of a polymorphic function at the same type in the same instance, $S_{\mathcal{RT}}$ can lead to a more efficient analysis than S_{LET}. To illustrate, consider the following program.

$$\begin{aligned}
\mathsf{let}\ \mathsf{f}_0 &= \Lambda\alpha.\lambda \mathsf{x}^\alpha.\,\mathsf{x}\\
\mathsf{f}_1 &= \Lambda\alpha.\lambda \mathsf{x}^\alpha.\,\mathsf{f}_0\alpha(\mathsf{f}_0\alpha\mathsf{x})\\
&\vdots\\
\mathsf{f}_n &= \Lambda\alpha.\lambda \mathsf{x}^\alpha.\,\mathsf{f}_{n-1}\alpha(\mathsf{f}_{n-1}\alpha\mathsf{x})\\
\mathsf{in}\ \ \mathsf{f}\ \mathit{int}\ 13
\end{aligned}$$

S_{LET} will analyze the two calls to f_i from f_{i+1} in two different instances. Consequently, S_{LET} will construct 2^n instances of f_0. On the other hand, $S_{\mathcal{RT}}$ evaluates a type application in a new instance only when it is supplied a new argument type. Thus, $S_{\mathcal{RT}}$ will construct only one instance of each f_i. This is because both calls to f_i from f_{i+1} are evaluated in an instance that maps α to *int*.

5 Related Work

Recent work on establishing equivalences between control-flow analyses and type systems [11, 17] is close in spirit to the idea of a type-respecting analysis. Palsberg and O'Keefe [17] show that safety analysis accepts the same programs as a simple type system extended with recursive types and subtyping. Their safety analysis can be regarded as a predicate on programs that uses a monovariant control-flow analysis to detect possible type errors. Heintze [11] extends their work to establish equivalences for a set of weaker type and control-flow systems. Taking the type system as primary, these results are similar to ours in that they prove that a flow analysis is type respecting. The results differ from ours in that the analysis is monovariant rather than polyvariant and the type system is monomorphic rather than polymorphic.

Many polyvariant flow analysis frameworks have been described by other authors [14, 16, 21, 23]. Beyond technical differences due to different underlying semantics, our framework differs in two important ways. First, it operates on a typed language—prior research has concentrated primarily on untyped or simply-typed languages, and has not directly considered languages with a polymorphic type system. Second, because our framework permits strategies that explicitly use type information to control polyvariance, we can specify analyses whose behavior is less closely tied to the syntactic structure of the program.

The idea of using type information to control polyvariance has been addressed informally in optimizing compilers for object-oriented languages such as Self [4] and Concert [19]. These systems use approximations to the types of arguments to select a context in which to analyze a method call. The approximations used are quite coarse—typically, only the outermost constructor of an argument's type plays a role in selecting the analysis context. Consequently, these systems do not respect types.

Dimock *et. al.* [5] present a framework that uses both type and flow information to optimize closure representations. Their system inserts explicit operations into programs in a typed intermediate language to convert between different closure representations. Their framework places relatively few constraints on the form of flow information used to select representations, and includes intersection types with which to express polyvariance. We expect that $S_{\mathcal{RT}}$ would be an ideal candidate for the analysis portion of their system.

Like our work, Heintze and McAllester [12] exploit types to control a flow analysis algorithm. But their system uses types for an entirely different purpose—to control termination of an algorithm for the monovariant core. We believe these uses of types are complementary, and expect we can adapt their algorithm to our framework.

References

1. Aiken, A., and Wimmers, E. L. Type inclusion constraints and type inference. *Proceedings of the International Conference on Functional Programming Languages and Computer Architecture* (1993), 31–41.
2. Aiken, A., Wimmers, E. L., and Lakshman, T. K. Soft typing with conditional types. In *Proc. ACM Symp. Principles of Programming Languages* (Jan. 1993), pp. 163–173.
3. Bacon, D. F., and Sweeney, P. F. Fast static analyses of C++ virtual function calls. In *OOPSLA* (1996).
4. Chambers, C., and Ungar, D. Iterative type analysis and extended message splitting: Optimizing dynamically-typed object-oriented programs. *Lisp and Symbolic Computation 4*, 3 (1991).
5. Dimock, A., Muller, R., Turbak, F., and Wells, J. Strongly typed Flow-directed Representation Transformations. In *Proceedings of the International Conference on Functional Programming Languages* (1997), pp. 11–24.
6. Flanagan, C., and Felleisen, M. Componential set-based analysis. In *Proc. ACM Conf. Programming Language Design and Implementation* (June 1997).
7. Girard, J.-Y. Une extension de l'interprétation de Gödel à l'analyse, et son application à l'élimination des coupures dans l'analyse et la théorie des types. In *Proceedings of the 2nd Scandinavian Logic Symposium* (1971), J. E. Fenstad, Ed., North–Holland, pp. 63–92.
8. Harper, R., and Mitchell, J. C. On the type structure of Standard ML. *ACM Transactions on Programming Languages and Systems 15*, 2 (Apr. 1993), 211–252.
9. Harper, R., and Morrisett, J. G. Compiling polymorphism using intensional type analysis. In *Proc. ACM Symp. Principles of Programming Languages* (Jan. 1995), ACM, pp. 130–141.
10. Heintze, N. Set-based analysis of ML programs. In *Proc. ACM Symp. Lisp and Functional Programming* (1994), pp. 306–317.
11. Heintze, N. Control-flow analysis and type systems. In *Proc. Intl. Static Analysis Symposium* (Sept. 1995), pp. 189–206. Also appears as CMU-CS-94-227.
12. Heintze, N., and McAllester, D. Linear-time subtransitive control flow analysis. In *Proc. ACM Conf. Programming Language Design and Implementation* (June 1997).
13. Henglein, F., and Mossin, C. Polymorphic binding-time analysis. In *Proc. European Symp. Programming* (Apr. 1994), D. Sannella, Ed., vol. 788 of *Lecture Notes in Computer Science*, pp. 287–301.
14. Jagannathan, S., and Weeks, S. T. A unified treatment of flow analysis in higher-order languages. In *Proc. ACM Symp. Principles of Programming Languages* (Jan. 1995), pp. 393–407.

15. JAGANNATHAN, S., AND WRIGHT, A. K. Effective flow analysis for avoiding run-time checks. In *Proc. Intl. Static Analysis Symposium* (Sept. 1995), mycroft, Ed., no. 983 in Lecture Notes in Computer Science, Springer–Verlag, pp. 207–224.
16. NIELSON, F., AND NIELSON, H. R. Infinitary control flow analysis: a collecting semantics for closure analysis. In *Proc. ACM Symp. Principles of Programming Languages* (Jan. 1997), ACM, pp. 332–345.
17. PALSBERG, J., AND O'KEEFE, P. A type system equivalent to flow analysis. In *Proc. ACM Symp. Principles of Programming Languages* (Jan. 1995), ACM, pp. 367–378.
18. PEYTON JONES, S. L. Compiling Haskell by program transformation: a report from the trenches. In *Proc. European Symp. Programming* (Apr. 1996).
19. PLEVYAK, J., AND CHIEN, A. A. Precise concrete type inference of object-oriented programs. In *OOPSLA* (1994).
20. REYNOLDS, J. C. Towards a theory of type structure. In *Paris Colloquium on Programming* (1974), no. 19 in Lecture Notes in Computer Science, Springer–Verlag, pp. 408–425.
21. SCHMIDT, D. Natural-semantics-based abstract interpretation (preliminary version). In *Proc. Intl. Static Analysis Symposium* (Sept. 1995), A. Mycroft, Ed., no. 983 in Lecture Notes in Computer Science, Springer–Verlag, pp. 1–18.
22. SHARIR, M., AND PNUELI, A. Two approaches to interprocedural dataflow analysis. In *Program Flow Analysis: Theory and Applications*, S. S. Muchnick and N. D. Jones, Eds. Prentice-Hall, 1981, pp. 189–235.
23. SHIVERS, O. *Control-Flow Analysis of Higher-Order Languages or Taming Lambda.* PhD thesis, Carnegie Mellon University, Computer Science Department, 1991.
24. TARDITI, D., MORRISETT, J. G., CHENG, P., STONE, C., HARPER, R., AND LEE, P. TIL: A type-directed optimizing compiler for ML. In *Proc. ACM Conf. Programming Language Design and Implementation* (May 1996).
25. TURNER, D. N., WADLER, P., AND MOSSIN, C. Once upon a type. In *Proc. Conf. Functional Programming and Computer Architecture* (June 1995).

Appendix

A Soundness and Safety

A flow analysis is *sound* if it provides a conservative approximation to program behavior. That is, the abstract value of a subexpression instance must include every possible exact value that the subexpression may yield. We use relation $\sqsubseteq_F$ (defined in Section 3) to define soundness for the flow functions yielded by a polyvariant analysis.

Definition 8 (Flow Soundness). Flow F is *sound* for program P if for every judgment $E, CE, i \vdash e \Rightarrow v$ in $\mathcal{D}_\mathcal{P}$, we have $E \sqsubseteq_F i$, $CE \sqsubseteq_F i$, and $v \sqsubseteq_F F(e, i)$.

Safety is a sufficient condition for soundness.

Theorem 9 (Safety $\Rightarrow$ Soundness). *Let $\phi, \phi, i_0 \vdash P \Rightarrow v$, and let F be a safe flow for P. Then F is sound for $\phi, \phi, i_0 \vdash P \Rightarrow v$.*

Proof Sketch. We prove the stronger statement that if $E, CE, i \vdash e \Rightarrow v$, and $E \sqsubseteq_F i$, $CE \sqsubseteq_F i$, and F is initialized for e in i, then F is sound for $E, CE, i \vdash e \Rightarrow v$. The proof proceeds by induction on the derivation.

B Proofs of Theorems for $S_{\mathcal{RT}}$

Theorem 5. *$S_{\mathcal{RT}}$ respects types.*

Proof. Fix a program P. We first exhibit an infinite flow F_{type} that is both safe and respects types. Since the "respects types" property is preserved by containment and since $F_{\text{minsafe}} \subseteq F_{\text{type}}$, the result follows. To construct F_{type}, we map each program point to *all* abstract values of appropriate type:

$$\begin{aligned}
F_{\text{type}}(e^\sigma, i) &= \{\langle f^{\sigma'}, i'\rangle \mid i'(\sigma') = i(\sigma)\} && \text{provided } \mathrm{dom}(i) \supseteq FTV(\sigma)\\
F_{\text{type}}(x^\sigma, i) &= \{\langle f^{\sigma'}, i'\rangle \mid i'(\sigma') = i(\sigma)\} && \text{provided } \mathrm{dom}(i) \supseteq FTV(\sigma)\\
F_{\text{type}}(\tau, i) &= \{\langle \tau', i'\rangle \mid i'(\tau') = i(\tau)\} && \text{provided } \mathrm{dom}(i) \supseteq FTV(\tau)\\
F_{\text{type}}(\alpha, i) &= \{\langle \tau', i'\rangle \mid i'(\tau') = i(\alpha)\} && \text{provided } \mathrm{dom}(i) \supseteq \{\alpha\}
\end{aligned}$$

Showing that F_{type} is safe requires checking the various safety constraints, and is straightforward. To show that F_{type} respects types, let $E, CE, i \vdash e^\sigma \Rightarrow v'$ be a judgment appearing in $\mathcal{D}_\mathcal{P}$ and let $\langle f^{\sigma'}, E', CE', i'\rangle \sqsubseteq_{F_{\text{type}}} F_{\text{type}}(e^\sigma, i)$. According to Definition 3, we must show that $\langle f^{\sigma'}, E', CE', i'\rangle \in [\![CE(\sigma)]\!]$, *i.e.*, $\phi \triangleright \langle\!\langle f^{\sigma'}, E', CE'\rangle\!\rangle : CE(\sigma)$. To do this, we need three lemmas.

Lemma 10. $\phi \triangleright \langle\!\langle f^{\sigma}, E, CE\rangle\!\rangle : CE(\sigma)$.

The proof of this lemma follows by induction on the structure of $\langle\!\langle \cdot, \cdot, \cdot\rangle\!\rangle$, employing a substitution property of the type system.

Lemma 11. *If $CE \sqsubseteq_{F_{\text{type}}} i$ then $CE \subseteq i$.*

The proof proceeds by induction on the definition of $\sqsubseteq_{F_{\text{type}}}$. When $CE = \phi$, the result trivially follows. For the inductive case, assume that $CE \sqsubseteq_{F_{\text{type}}} i$ and let an arbitrary $\alpha \in dom(CE)$ be given. By the definition of F_{type}, $\alpha \in dom(i)$. By the definition of $\sqsubseteq_{F_{\text{type}}}$, $CE(\alpha) \sqsubseteq_{F_{\text{type}}} F_{\text{type}}(\alpha, i)$. Hence, there exists $\langle \tau, i'\rangle \in F_{\text{type}}(\alpha, i)$ and $CE' \sqsubseteq_{F_{\text{type}}} i'$ such that $CE'(\tau) = CE(\alpha)$. By the induction hypothesis, $CE' \subseteq i'$. By the definition of F_{type}, $i'(\tau) = i(\alpha)$. Hence, $CE(\alpha) = CE'(\tau) = i'(\tau) = i(\alpha)$.

Lemma 12. *Let $E, CE, i \vdash e^\sigma \Rightarrow v$ be a judgment appearing in $\mathcal{D}_\mathcal{P}$ that uses strategy $S_{\mathcal{RT}}$. Then $CE \subseteq i$.*

The proof of this lemma follows by induction over the structure of a derivation constructed using strategy $S_{\mathcal{RT}}$.

Given these lemmas, we establish $\phi \triangleright \langle\!\langle f^{\sigma'}, E', CE'\rangle\!\rangle : CE(\sigma)$ by showing that $CE(\sigma) = CE'(\sigma')$. First, $\phi \triangleright \langle\!\langle f^{\sigma'}, E', CE'\rangle\!\rangle : CE'(\sigma')$ by Lemma 10. Since $\langle f^{\sigma'}, E', CE', i'\rangle \sqsubseteq_{F_{\text{type}}} F_{\text{type}}(e^\sigma, i)$, we have $CE' \sqsubseteq_{F_{\text{type}}} i'$ by the definition of

$\sqsubseteq$. Then $CE' \subseteq i'$ by Lemma 11, hence $CE'(\sigma') = i'(\sigma')$. By the definition of F_{type}, $i'(\sigma') = i(\sigma)$. By Lemma 12, $CE \subseteq i$, hence $CE(\sigma) = i(\sigma)$. Then $CE'(\sigma') = i'(\sigma') = i(\sigma) = CE(\sigma)$.

The theorem now follows from the following lemma which establishes that type-respecting flow functions are closed under containment.

Lemma 13. *Let F and F' be flow functions for program P under strategy $S_{\mathcal{RT}}$ such that $F \subseteq F'$. If F' respects types then F respects types.*

Since F_{type} respects types and $F_{\text{minsafe}} \subseteq F_{\text{type}}$, Lemma 13 implies that F_{minsafe} respects types. This completes the proof of Theorem 5.

Theorem 7. *The minimum safe flow for $P \in ML_i$ under $S_{\mathcal{RT}}$ is finite.*

Proof. The idea is similar to the proof of Theorem 5. Let F_{minsafe} be the minimum safe flow for $P \in ML_i$ under $S_{\mathcal{RT}}$. We first exhibit a flow F_{expand} that is both safe and finite. Since $F_{\text{minsafe}} \subseteq F_{\text{expand}}$, F_{minsafe} is finite. To construct F_{expand}, we essentially "expand away" all polymorphism in the program. To formalize this, we use the auxiliary function $\mathcal{X}$, which constructs the set on which F_{expand} will be defined.

$$
\begin{gathered}
\mathit{ExpInst} = \mathcal{P}(\mathit{Occ} \times I + \mathit{TypeOcc} \times I) \\
\mathcal{X} : \mathit{Exp} \times I \times (\mathit{Var} \to \mathit{Constructor}^* \to \mathit{ExpInst}) \to \mathit{ExpInst}
\end{gathered}
$$

$$
\begin{aligned}
\mathcal{X}((x\,\boldsymbol{\tau}), i, \rho) &= \{\langle (x\,\boldsymbol{\tau}), i\rangle\} \cup \rho(x)(i(\tau_1), \dots, i(\tau_n)) \\
\mathcal{X}(\lambda x^{\tau}.e, i, \rho) &= \{\langle \lambda x^{\tau}.e, i\rangle\} \cup \mathcal{X}(e, i, \rho) \\
\mathcal{X}((\lambda x^{\sigma}.e \;\; \Lambda\boldsymbol{\alpha}.e'), i, \rho) &= \{\langle (\lambda x^{\sigma}.e \;\; \Lambda\boldsymbol{\alpha}.e'), i\rangle\} \cup \mathcal{X}(e, i, \rho[x \mapsto f]), \text{ where} \\
& \qquad f(\tau_1, \dots, \tau_n) = \{\langle \alpha_1, i_1\rangle, \dots \langle \alpha_n, i_n\rangle\} \cup \mathcal{X}(e', i', \rho) \\
& \qquad\qquad i' = i[\alpha_1 \mapsto \tau_1] \dots [\alpha_n \mapsto \tau_n] \\
\mathcal{X}((e_1\, e_2), i, \rho) &= \{\langle (e_1\, e_2), i\rangle\} \cup \mathcal{X}(e_1, i, \rho) \cup \mathcal{X}(e_2, i, \rho) \\
\mathcal{X}(\mu x_1^{\tau_1}.\lambda x_2^{\tau_2}.e, i, \rho) &= \{\langle \mu x_1^{\tau_1}.\lambda x_2^{\tau_2}.e, i\rangle, \; \langle \lambda x_2^{\tau_2}.e, i\rangle\} \cup \mathcal{X}(e, i, \rho)
\end{aligned}
$$

Let $T = \mathcal{X}(P, \phi, \phi)$. By construction, T is finite. We define F_{expand} as we did F_{type} in the proof of Theorem 5, except that we only use instances that appear in T. Showing that F_{expand} is safe requires checking the various safety constraints, and is straightforward. F_{expand} is finite since T is finite.

This article was processed using the LaTeX macro package with LLNCS style