A Type- and Control-Flow Analysis for System F

Matthew Fluet

Computer Science Department
Rochester Institute of Technology, Rochester NY 14623
`mtf@cs.rit.edu`

Abstract. We present a monovariant flow analysis for System F (with recursion). The flow analysis yields both *control-flow* information, approximating the λ- and Λ-expressions that may be bound to variables, and *type-flow* information, approximating the type expressions that may instantiate type variables. Moreover, the two flows are mutually beneficial: the control flow determines which Λ-expressions may be applied to which type expressions (and, hence, which type expressions may instantiate which type variables), while the type flow filters the λ- and Λ-expressions that may be bound to variables (by keeping only those expressions with a static type that is compatible with the static type of the variable with respect to the type flow). As is typical for a monovariant control-flow analysis, control-flow information is expressed as an abstract environment mapping variables to sets of (syntactic) λ- and Λ-expressions that appear in the program under analysis. Similarly, type-flow information is expressed as an abstract environment mapping type variables to sets of (syntactic) types that appear in the program under analysis. Compatibility of static types (with free type variables) with respect to a type flow is decided by interpreting the abstract environment as productions for a *regular-tree grammar* and asking if the languages generated by taking the types in question as starting terms have a non-empty intersection.

1 Introduction

Control-flow analysis is an important enabling technology for the compilation and optimization of functional languages. Because functional languages have first-class functions, the control flow of a functional program is not syntactically apparent: in an application expression, the function can itself be the result of a computation and may not be available until run time. Indeed, during the execution of a program, many different functions may be applied at the same (source-program) application expression. A control-flow analysis [21,44,43,32,27] approximates, at compile time, the flow of first-class functions in a program: which first-class functions might be bound to a given variable or returned by a given expression at run time. This approximate control-flow information can be used to enable optimizations of a functional language.

Control-flow analyses are typically formulated for *dynamically-* or *simply-*typed functional languages.[1] However, most statically-typed functional languages have rich type systems that include polymorphic types. Indeed, System F [13,38], the polymorphic lambda calculus, and extensions thereof are commonly used as typed intermediate languages in compilers for functional languages [48,34,46]. Typed intermediate languages provide a number of benefits. First, explicit type information can support type-dependent optimizations, such as using a specialized representation for known types rather than a universal representation. Second, explicit type information can support validation of optimizations, by detecting when an optimization transforms a well-typed program to an ill-typed program. Since optimizations are performed on a typed intermediate language and optimizations are enabled by control-flow analyses, it is natural to seek a control-flow analysis that is formulated for System F.

While one could naïvely adopt an existing control-flow analysis that is formulated for a dynamically- or simply-typed functional language and ignore the System F features of type abstraction and type application, such an approach fails to take advantage of the static information provided by a well-typed program. Intuitively, a control-flow analysis for System F should exploit the well-typedness of the program under analysis in order to obtain more precise control-flow information. For instance, if a control-flow analysis asserts that a variable x might be bound to a function of type $\mathsf{int} \rightarrow \mathsf{int}$, a function of type $\mathsf{bool} \rightarrow \mathsf{bool}$, and a function of type $\mathsf{string} \rightarrow \mathsf{string}$ (and no other functions), but the static type of x is $\mathsf{int} \rightarrow \mathsf{int}$, then the type soundness of the language guarantees that x will only be bound to functions of type $\mathsf{int} \rightarrow \mathsf{int}$ at run time and the control-flow result may be soundly refined to assert that x might be bound to only the function of type $\mathsf{int} \rightarrow \mathsf{int}$. However, if the static type of x is $\alpha \rightarrow \alpha$ (where the type variable α is bound by a type abstraction in the program under analysis), then it is unclear whether or not the control-flow result may be soundly refined, because the type variable α may be soundly instantiated at any type.

Given additional information that asserts that the type variable α might be instantiated at the type int and the type bool (and no other types), then the control-flow result may be soundly refined to assert that x might be bound to only the function of type $\mathsf{int} \rightarrow \mathsf{int}$ and the function of type $\mathsf{bool} \rightarrow \mathsf{bool}$. This additional information may be obtained by a *type-flow analysis* that approximates, at compile time, the flow of types in a program: which types might instantiate a given type variable at run time. As demonstrated by the example above, this approximate type-flow information can be used to increase the precision of a control-flow analysis. Furthermore, this approximate type-flow information can be used to enable type-dependent optimization, such as guiding the specialization of a polymorphic function that is used at a small number of distinct types or

[1] Although there are many *type-based* [33] control-flow analyses, whereby the analyses themselves are expressed as a sophisticated type systems (e.g., type-and-effect systems [9,31,19], type systems with polymorphic types [36], type systems with union/intersection types [50,29]), the language under analysis is typically a simply-typed language.

eliminating type operations in a language with intensional polymorphism [14]. Just as a control-flow analysis yields useful information because, for a given program, it is unlikely that a given variable is bound to *every* function during execution, a type-flow analysis yields useful information because it is unlikely a given type variable is instantiated at that *every* type during execution.

Although type-flow information and control-flow information might be obtained by independent analyses, the two kinds of information can be mutually beneficial, particularly for the higher-rank impredicative polymorphism of System F. Control-flow analysis supports type-flow analysis by yielding information about the type abstractions which may be applied at type applications and, hence, about the types at which type variables may be instantiated. The type-flow information soundly refines the control-flow information by rejecting flows that are incompatible with the static typing of the program under analysis; because the static typing may be expressed in terms of syntactic types with free type variables, the type-flow information is used to determine the compatibility of types. When the type-flow information refines the control-flow information by rejecting the flow of a type abstraction, the type-flow information itself may be refined because the type abstraction may be applied at fewer type applications, and, hence, there may be fewer types at which the type variable may be instantiated.[2]

In a combined type- and control-flow analysis, the type-flow information soundly refines the control-flow information by determining when types are compatible. In the presence of recursion and higher-rank impredicative polymorphism, the type-flow information must approximate complex relationships between type variables and types and the compatibility or incompatibility of types with respect to the type-flow information may not be obvious. Indeed, during the execution of a program that is well-typed in System F (with recursion), a type variable may be instantiated at an infinite number of types. In order to obtain a computable analysis, the type-flow information must use a finite representation that approximates the (potentially infinite) set of ground types that may instantiate a type variable and the compatibility of types with respect to the type-flow information must be a decidable property.

Most control-flow analyses approximate the (potentially infinite) set of first-class functions that might be bound to a variable at run time by a (necessarily finite) set of function expressions (possibly with free variables) that appear in the program under analysis. Similarly, a type-flow analysis may approximate the (potentially infinite) set of ground types that may instantiate a type variable at run time by a (necessarily finite) set of type expressions (possibly with free type variables) that appear in the program under analysis. For instance, if a type-flow analysis asserts that a type variable α might be instantiated at the type expression $\mathsf{int} \rightarrow \mathsf{int}$ and the type expression $\mathsf{int} \rightarrow \alpha$ (and no other type expressions), then, by interpreting the type-flow information as productions for a

[2] In practice, though, a flow analysis is computed by adding information that is consistent with existing information (i.e., ascending a lattice) rather than removing information that is inconsistent with existing information (i.e., descending a lattice).

regular-tree grammar [12,2,7], the type-flow analysis may be seen to be asserting that the type variable α might be instantiated at the infinite set of ground types: $\{\mathsf{int} \to \mathsf{int}, \mathsf{int} \to \mathsf{int} \to \mathsf{int}, \mathsf{int} \to \mathsf{int} \to \mathsf{int} \to \mathsf{int}, \ldots\}$. Furthermore, if the type-flow analysis asserts that a type variable β might be instantiated at the type expression $\mathsf{int} \to \mathsf{bool}$ and the type expression $\mathsf{int} \to \beta$ (and no other type expressions) and a control-flow analysis asserts that a variable x might be bound to a function of type $\mathsf{int} \to \mathsf{int}$, a function of type $\mathsf{bool} \to \mathsf{int}$, a function of type $\mathsf{string} \to \mathsf{int}$, a function of type $\mathsf{int} \to \alpha$, and a function of type $\mathsf{int} \to \beta$ (and no other functions), but the static type of x is α, then the control-flow result may be soundly refined to assert that x might be bound to only the function of type $\mathsf{int} \to \mathsf{int}$ and the function of type $\mathsf{int} \to \alpha$, because the types of these two functions are compatible with the type α (with respect to the type-flow information), while the types of the other three functions are incompatible with the type α.

Two types are compatible (with respect to the type-flow information) if there exists a ground type that is a member of the sets of ground types at which the types might be instantiated; conversely, two types are incompatible if there does not exist a ground type that is a member of the sets of ground types at which the types might be instantiated. The type soundness of the language guarantees that a variable will only be bound to a well-typed closed function of a ground type at run time; hence, if there is no ground type at which both the static type of a variable and the static type of a function might be instantiated, then that variable will never be bound to that function at run time. For example, the types $\mathsf{int} \to \alpha$ and α are compatible because the type $\mathsf{int} \to \alpha$, interpreted as a starting term for the regular-tree grammar corresponding to the type-flow information, represents the infinite set of ground types $\{\mathsf{int} \to \mathsf{int} \to \mathsf{int}, \mathsf{int} \to \mathsf{int} \to \mathsf{int} \to \mathsf{int}, \ldots\}$, which has a non-empty intersection with the infinite set of ground types that might instantiate the type variable α (given above). Similarly, the types $\mathsf{int} \to \beta$ and α are incompatible because the type $\mathsf{int} \to \beta$ represents the infinite set of ground types $\{\mathsf{int} \to \mathsf{int} \to \mathsf{bool}, \mathsf{int} \to \mathsf{int} \to \mathsf{int} \to \mathsf{bool}, \ldots\}$, which has an empty intersection with the infinite set of ground types that might instantiate the type variable α. Since regular-tree grammars are closed under intersection and the emptiness of a regular-tree grammar is decidable, the compatibility of types with respect to the type-flow information is a decidable property.

Overview

We present a monovariant[3] type- and control-flow analysis for System F extended with recursive functions. Our flow analysis is a variation on 0CFA, the classic monovariant control-flow analysis [32]. For a given program, the flow analysis computes an abstract environment that maps variables to (finite) sets of λ- and Λ-expressions that appear in the program and maps type variables to (finite) sets of type expressions that appear in the program.

[3] i.e., context insensitive

Soundness of the analysis is proven with respect to an operational semantics for System F given in the style of the administrative-normal-form (ANF) environment- and continuation-based C_aEK abstract machine [10], where the (concrete) environment component of the abstract machine maps variables to closures (pairs of λ- or Λ-expressions and an environment, which captures the free variables and type variables of the λ- or Λ-expression) and maps type variables to *type closures* (pairs of type expressions and an environment, which captures the free type variables of the type expression). A sound flow analysis computes an abstract environment that approximates every concrete environment that arises during evaluation. To simplify the soundness proof, the machine transition rules are instrumented with explicit type-equality conditions; these conditions are necessarily satisfied by a well-typed program, but are the concrete analogue of the type-compatibility conditions used in the flow analysis. We present both the run-time type-equality and the analysis-time type-compatibility predicates as judgments; this yields a declarative specification of type compatibility, for which the regular-tree-grammar interpretation gives an algorithmic implementation.

Our formulation of the type- and control-flow analysis as a refinement of the syntax-directed constraint-based formulation of 0CFA establishes that the combined type- and control-flow analysis can be more precise than 0CFA. Although not as precise as a type-directed polyvariant[4] control-flow analysis [20], our monovariant type- and control-flow analysis nonetheless rejects some similar classes of spurious flows and furthermore has the benefits of handling full (i.e., impredicative) System F and terminating for all well-typed programs.

2 Language and Semantics

Our source language is a variant of System F, extended with recursive functions and presented in (a restriction of) administrative normal form (ANF). The static semantics of the language is entirely standard, but given for completeness. The operational semantics of the language is presented as an abstract machine, but deviates in some ways from a straightforward adaptation of the environment- and continuation-based C_aEK abstract machine to an ANF variant of System F.

2.1 Syntax

The syntax of our ANF variant of System F is given in Figure 1.

Types include function types, type variables, and universal types; in the universal type $\forall\alpha.\ \tau$, the type variable α is bound in the type τ. Type equality is syntactic identity (up to α-conversion of bound type variables).

Expressions include variables, `let`-bindings of values, `let`-bindings of non-tail function applications, and `let`-bindings of non-tail type applications; in the `let`-binding expressions `let` x`:`τ_x `=` $\cdots$ `in` e, the variable x is bound in e. Values include recursive functions and recursive type abstractions; in the recursive

[4] i.e., context sensitive

$$\begin{array}{llrl}
\textit{Types} & \textit{Type} \ni \tau & ::= & \tau \to \tau \mid \alpha \mid \forall\alpha.\,\tau \\
\textit{Type variables} & \textit{TyVar} \ni \alpha, \beta & & \\
 & & & \\
\textit{Expressions} & \textit{Exp} \ni e & ::= & x \mid \texttt{let}\ x{:}\tau\ \texttt{=}\ v\ \texttt{in}\ e \mid \\
 & & & \texttt{let}\ x{:}\tau\ \texttt{=}\ x\ x\ \texttt{in}\ e \mid \\
 & & & \texttt{let}\ x{:}\tau\ \texttt{=}\ x\ \texttt{[}\tau\texttt{]}\ \texttt{in}\ e \\
\textit{Values} & \textit{Value} \ni v & ::= & \mu x{:}\tau\,.\,\lambda x{:}\tau\,.\,e \mid \mu x{:}\tau\,.\,\Lambda\alpha\,.\,e \\
\textit{Variables} & \textit{Var} \ni x, y, z, f, g & & \\
 & & & \\
\textit{Programs} & \textit{Prog} \ni P & ::= & e
\end{array}$$

Fig. 1. Syntax of ANF System F

function $\mu f{:}\tau_f\,.\,\lambda x{:}\tau_x\,.\,e_b$, the variables f and x are bound in the expression e_b and in the recursive type abstraction $\mu f{:}\tau_f\,.\,\Lambda\alpha\,.\,e_b$, the variable f and the type variable α are bound in the expression e_b. Programs are (closed, well-typed) expressions.

The language is Church-style, in which every bound variable is annotated with its type. In contrast to some presentations of ANF-like languages [39,8,10] but in concert with some others [47,49,5], we restrict the constituents of function applications and type applications to variables, rather than allowing a larger class of "trivial" expressions, and we restrict function applications and type applications to non-tail calls, rather than allowing tail calls. Neither of these restrictions is essential for the forthcoming type- and control-flow analysis; we adopt them simply to reduce the number of inference rules and helper functions in the static semantics, operational semantics, and flow analysis.

We let $\textit{TyVar}_P$ be the (finite) set of Λ-bound type variables, $\textit{Var}_P$ be the (finite) set of `let`-, μ-, and λ-bound variables, $\textit{Type}_P$ be the (finite) set of types, and $\textit{Value}_P$ be the (finite) set of values that occur in a given program P; distinguishing syntactically identical sub-terms can be defined formally using paths or unique labellings.

Finally, we define a function $\mathsf{last}(\cdot)$ on expressions, which returns the variable that yields the expression's value:

$$\begin{aligned}
\mathsf{last}(\cdot) &:: \textit{Exp} \to \textit{Var} \\
\mathsf{last}(x) &= x \\
\mathsf{last}(\texttt{let}\ x{:}\tau_x\ \texttt{=}\ v\ \texttt{in}\ e) &= \mathsf{last}(e) \\
\mathsf{last}(\texttt{let}\ x{:}\tau_x\ \texttt{=}\ x_f\ x_a\ \texttt{in}\ e) &= \mathsf{last}(e) \\
\mathsf{last}(\texttt{let}\ x{:}\tau_x\ \texttt{=}\ x_f\ \texttt{[}\tau_a\texttt{]}\ \texttt{in}\ e) &= \mathsf{last}(e)
\end{aligned}$$

2.2 Static Semantics

The standard static semantics for System F, adapted to our ANF variant, are given in Figure 2. A type-variable context Δ records free type variables and the judgment $\Delta \vdash \tau$ asserts that all of the free type variables of type τ are

$$\begin{array}{lrcl} \textit{Type-variable contexts} & \mathit{TCtxt} \ni \Delta & ::= & \emptyset \mid \Delta, \alpha{:}\star \\ \textit{Variable contexts} & \mathit{Ctxt} \ni \Gamma & ::= & \emptyset \mid \Gamma, x{:}\tau \end{array}$$

$\boxed{\Delta \vdash \tau}$

$$\frac{\Delta \vdash \tau_a \qquad \Delta \vdash \tau_b}{\Delta \vdash \tau_a \to \tau_b} \qquad \frac{\Delta(\alpha) = \star}{\Delta \vdash \alpha} \qquad \frac{\Delta, \alpha{:}\star \vdash \tau_b}{\Delta \vdash \forall\alpha.\ \tau_b}$$

$\boxed{\Delta; \Gamma \vdash v : \tau}$

$$\frac{\begin{array}{c}\Delta \vdash \tau_f \qquad \Delta \vdash \tau_z \qquad \Delta; \Gamma, f{:}\tau_f, z{:}\tau_z \vdash e_b : \tau_b \\ \tau_f = \tau_a \to \tau_b \qquad \tau_z = \tau_a\end{array}}{\Delta; \Gamma \vdash \mu f{:}\tau_f.\lambda z{:}\tau_z.e_b : \tau_a \to \tau_b} \qquad \frac{\begin{array}{c}\Delta \vdash \tau_f \qquad \Delta, \beta{:}\star; \Gamma, f{:}\tau_f \vdash e_b : \tau_b \\ \tau_f = \forall\beta.\ \tau_b\end{array}}{\Delta; \Gamma \vdash \mu f{:}\tau_f.\Lambda\beta.e_b : \forall\beta.\ \tau_b}$$

$\boxed{\Delta; \Gamma \vdash e : \tau}$

$$\frac{\Gamma(x) = \tau_x}{\Delta; \Gamma \vdash x : \tau_x} \qquad \frac{\begin{array}{c}\Delta; \Gamma \vdash v : \tau_v \\ \Delta \vdash \tau_x \qquad \tau_x = \tau_v \qquad \Delta; \Gamma, x{:}\tau_x \vdash e : \tau\end{array}}{\Delta; \Gamma \vdash \texttt{let}\ x{:}\tau_x\ \texttt{=}\ v\ \texttt{in}\ e : \tau}$$

$$\frac{\begin{array}{c}\Gamma(x_f) = \tau_a \to \tau_b \qquad \Gamma(x_a) = \tau_a \\ \Delta \vdash \tau_x \qquad \tau_x = \tau_b \qquad \Delta; \Gamma, x{:}\tau_x \vdash e : \tau\end{array}}{\Delta; \Gamma \vdash \texttt{let}\ x{:}\tau_x\ \texttt{=}\ x_f\ x_a\ \texttt{in}\ e : \tau}$$

$$\frac{\begin{array}{c}\Gamma(x_f) = \forall\beta.\ \tau_b \qquad \Delta \vdash \tau_a \\ \Delta \vdash \tau_x \qquad \tau_x = [\beta \rightarrowtail \tau_a](\tau_b) \qquad \Delta; \Gamma, x{:}\tau_x \vdash e : \tau\end{array}}{\Gamma \vdash \texttt{let}\ x{:}\tau_x\ \texttt{=}\ x_f\ \texttt{[}\tau_a\texttt{]}\ \texttt{in}\ e : \tau}$$

$\boxed{\vdash P : \tau}$

$$\frac{\emptyset; \emptyset \vdash e : \tau}{\vdash e : \tau}$$

Fig. 2. Static Semantics of ANF System F

$$\begin{array}{llcl}
\textit{Run-time types} & \textit{RType} \ni \pi & ::= & \langle \tau; \phi \rangle \\
\textit{Run-time type environments} & \textit{RTEnv} \ni \phi & ::= & \emptyset \mid \phi, \alpha \mapsto \pi \\
\\
\textit{Run-time values} & \textit{RValue} \ni w & ::= & \langle v; \phi; \rho \rangle \\
\textit{Run-time value environments} & \textit{REnv} \ni \rho & ::= & \emptyset \mid \rho, x \mapsto w \\
\\
\textit{Continuations} & \textit{Kont} \ni \kappa & ::= & \bullet \mid \langle x; \tau; e; \phi; \rho \rangle {::} \kappa \\
\textit{States} & \textit{State} \ni \varsigma & ::= & \langle e; \phi; \rho; \kappa \rangle
\end{array}$$

$\boxed{\varsigma \longrightarrow \varsigma}$

$$\frac{\rho_r(x_r) = w_r \qquad \vdash w_r \coloneqq \langle \tau_z; \phi \rangle}{\begin{array}{l}\langle x_r; \phi_r; \rho_r; \langle z; \tau_z; e; \phi; \rho \rangle {::} \kappa \rangle \\ \quad \longrightarrow \langle e; \phi; \rho, z \mapsto w_r; \kappa \rangle\end{array}} \qquad \frac{w = \langle v; \phi; \rho \rangle}{\begin{array}{l}\langle \texttt{let } x{:}\tau_x \texttt{ = } v \texttt{ in } e; \phi; \rho; \kappa \rangle \\ \quad \longrightarrow \langle e; \phi; \rho, x \mapsto w; \kappa \rangle\end{array}}$$

$$\frac{\begin{array}{c}\rho(x_f) = w_f \qquad w_f = \langle \mu f{:}\tau_f \,.\, \lambda z{:}\tau_z \,.\, e_b; \phi_f; \rho_f \rangle \\ \rho(x_a) = w_a \qquad \vdash w_a \coloneqq \langle \tau_z; \phi_f \rangle\end{array}}{\begin{array}{l}\langle \texttt{let } x{:}\tau_x \texttt{ = } x_f\ x_a \texttt{ in } e; \phi; \rho; \kappa \rangle \\ \quad \longrightarrow \langle e_b; \phi_f; \rho_f, f \mapsto w_f, z \mapsto w_a; \langle x; \tau_x; e; \phi; \rho \rangle {::} \kappa \rangle\end{array}}$$

$$\frac{\begin{array}{c}\rho(x_f) = w_f \qquad w_f = \langle \mu f{:}\tau_f \,.\, \Lambda\beta \,.\, e_b; \phi_f; \rho_f \rangle \\ \pi_a = \langle \tau_a; \phi \rangle\end{array}}{\begin{array}{l}\langle \texttt{let } x{:}\tau_x \texttt{ = } x_f\ \texttt{[}\tau_a\texttt{]} \texttt{ in } e; \phi; \rho; \kappa \rangle \\ \quad \longrightarrow \langle e_b; \phi_f, \beta \mapsto \pi_a; \rho_f, f \mapsto w_f; \langle x; \tau_x; e; \phi; \rho \rangle {::} \kappa \rangle\end{array}}$$

Fig. 3. Operational Semantics of ANF System F

recorded in Δ. A variable context Γ records free variables and their types and the judgments $\Delta; \Gamma \vdash v : \tau$ and $\Delta; \Gamma \vdash e : \tau$ assert that value v and expression e have type τ in Δ and Γ; in the rule for type applications, we write $[\beta \rightarrowtail \tau_a]\tau_b$ for the capture-avoiding substitution of τ_a for free occurrences of β in τ_b. The judgment $\vdash P : \tau$ asserts that program P is closed and has type τ.

2.3 Operational Semantics

The operational semantics for our ANF-variant of System F is presented as an adaptation of the environment- and continuation-based C_aEK machine [10] and is given in Figure 3.

A machine state ς has four components: a control expression, a run-time type environment, a run-time value environment, and a continuation.

A run-time type environment ϕ is a map from type variables to run-time types and a run-time value environment ρ is a map from variables to run-time values. A run-time type π is a “type closure”: a pair of a (possibly open) type and a run-time type environment; the run-time type environment captures the free

$$\boxed{\Delta \vdash \pi \equiv \pi}$$

$$\frac{\phi_1(\alpha_1) = \pi_1' \qquad \emptyset \vdash \pi_1' \equiv \pi_2}{\Delta \vdash \langle \alpha_1; \phi_1 \rangle \equiv \pi_2} \qquad \frac{\phi_2(\alpha_2) = \pi_2' \qquad \emptyset \vdash \pi_1 \equiv \pi_2'}{\Delta \vdash \pi_1 \equiv \langle \alpha_2; \phi_2 \rangle}$$

$$\frac{\Delta \vdash \langle \tau_{z1}; \phi_1 \rangle \equiv \langle \tau_{z2}; \phi_2 \rangle \qquad \Delta \vdash \langle \tau_{b1}; \phi_1 \rangle \equiv \langle \tau_{b2}; \phi_2 \rangle}{\Delta \vdash \langle \tau_{z1} \to \tau_{b1}; \phi_1 \rangle \equiv \langle \tau_{z2} \to \tau_{b2}; \phi_2 \rangle}$$

$$\frac{\Delta(\alpha_1) = \star \qquad \Delta(\alpha_2) = \star \qquad \alpha_1 = \alpha_2}{\Delta \vdash \langle \alpha_1; \phi_1 \rangle \equiv \langle \alpha_2; \phi_2 \rangle} \qquad \frac{\Delta, \alpha{:}\star \vdash \langle \tau_{b1}; \phi_1 \rangle \equiv \langle \tau_{b2}; \phi_2 \rangle}{\Delta \vdash \langle \forall \alpha.\ \tau_{b1}; \phi_1 \rangle \equiv \langle \forall \alpha.\ \tau_{b2}; \phi_2 \rangle}$$

$$\boxed{\vdash w :\equiv \pi}$$

$$\frac{\emptyset \vdash \langle \tau_f; \phi \rangle \equiv \pi}{\vdash \langle \mu f{:}\tau_f\,.\,\lambda z{:}\tau_z\,.\,e_b; \phi; \rho \rangle :\equiv \pi} \qquad \frac{\emptyset \vdash \langle \tau_f; \phi \rangle \equiv \pi}{\vdash \langle \mu f{:}\tau_f\,.\,\Lambda \beta\,.\,e_b; \phi; \rho \rangle :\equiv \pi}$$

Fig. 4. Run-time Type Equality

type variables of the type. A run-time value w is a "function closure" or a "type-abstraction closure": a triple of a (possibly open) value (a recursive function or a recursive type abstraction), a run-time type environment, and a run-time value environment; the run-time type environment captures the free type variables of the value and the run-time value environment captures the free variables of the value.

A continuation κ is a stack of frames, each of the form $\langle x; \tau_x; \phi; \rho; e \rangle$, where x is the variable receiving the result w of a non-tail function application or non-tail type application, τ_x is the (static, syntactic) type of x, and e is the expression to be evaluated in the environments ϕ and ρ extended with x bound to w to yield the result of the frame.

Ignoring the shaded terms, the machine transition rules are straightforward. The first rule returns a result to the top-most frame of the continuation when the control expression has been reduced to a variable. The second rule creates function closures and type-abstraction closures. The third and fourth rules extract the expression body, run-time type environment, and run-time value environment from the applied function closure or type-abstraction closure, extend the closure's run-time value environment with f bound to the applied function closure or type-abstraction closure (making the recursive function or recursive type-abstraction available to the expression body), extend the closure's run-time type environment with the run-time value argument (in the case of a function application) or extend the closure's run-time type environment with the run-time type argument (in the case of a type application), and push a frame onto the continuation to receive the result of the function application or type application. Note that the machine transitions are syntax directed and deterministic.

We now consider the shaded terms in the first and third rules and the judgments and rules in Figure 4. In essence, the shaded terms perform a kind of run-time type checking at the point where a run-time value environment is extended with a non-local run-time value. In the first rule, the result w_r must have a run-time type equal to $\langle \tau_z; \phi \rangle$, the run-time type of the variable receiving the result. In the third rule, the argument w_a must have a run-time type equal to $\langle \tau_z; \phi_f \rangle$, the run-time type of the variable receiving the argument.

The rules for the judgment $\vdash w :\equiv \pi$ simply form the run-time type of the recursive function or recursive type abstraction from the (static, syntactic) type of the μ-bound variable and the run-time type environment of the closure; note that there is no type checking of the value using the typing judgments from the static semantics. The judgment $\Delta \vdash \pi_1 \equiv \pi_2$ asserts that the run-time types π_1 and π_2 are equal under Δ. The first and second rules expand a Λ-bound type variable according to the appropriate run-time type environment. The third rule asserts that two function types are equal when their argument types are equal and their result types are equal. The fifth rule asserts that two universal types (sharing the same $\forall$-bound type variable via α conversion) are equal when their range types are equal. The fourth rule asserts that two $\forall$-bound type variables are equal when they are the same type variable. Note that the first and second rules assert the equality of the expanded type under the empty type-variable context; this is because the expanded run-time type type is not in the scope of the $\forall$-bound type variables appearing in Δ.

We emphasize that the shaded terms and the judgments in Figure 4 are instrumentation that simplifies the proof of flow soundness in Section 3.1. The presence of these terms does not change the evaluation of well-typed programs, a fact that is established in Section 2.4.

2.4 Type Soundness

Theorem 1 (Type Soundness).
If $\vdash e : \tau$ *and* $\langle e; \emptyset; \emptyset; \bullet \rangle \longrightarrow^* \varsigma'$*, then either* $\varsigma' = \langle x'; \phi'; \rho'; \bullet \rangle$ *or* $\varsigma' \longrightarrow \varsigma''$.

A syntactic proof [51], using Progress and Preservation theorems, is given in a companion technical report [11]. In addition to the judgments of Figure 2, we introduce judgments that assert the "well-typedness" of run-time types ($\vdash \pi \Rrightarrow \tau$), run-time type environments ($\vdash \phi : \theta$), run-time values ($\vdash w : \tau$), run-time value environments ($\vdash \rho : \Gamma$), continuations ($\vdash \tau \rhd \kappa : \tau$), and states ($\vdash \varsigma : \tau$). Of note is the judgment $\vdash \phi : \theta$ that asserts that the run-time type environment ϕ corresponds to a substitution θ; the domains of ϕ and θ are equal, but whereas ϕ maps a type variable to a run-time type (a pair of a (possibly open) type and a run-time type environment), θ maps a type variable to a closed type obtained by (recursively) expanding the (possibly open) type by the run-time type environment.

Preservation is entirely straightforward, but Progress requires showing that the shaded terms of Figure 2 can always be satisfied with well-typed configurations. A key lemma is the following, which establishes that two run-time types may be judged equal if their induced expansions are syntactically equal:

Lemma 2 (**Syntactic Type Equality of Expansions implies Run-Time Type Equality**)
If $\vdash \phi_1 : \theta_1$, $\emptyset \vdash \theta_1(\tau_1)$, $\vdash \phi_2 : \theta_2$, $\emptyset \vdash \theta_2(\tau_2)$, *and* $\theta_1(\tau_1) = \theta_2(\tau_2)$,
then $\emptyset \vdash \langle \tau_1; \phi_1 \rangle \equiv \langle \tau_2; \phi_2 \rangle$.

An immediate corollary to Type Soundness is that, for well-typed programs, the operational semantics of Figure 3 with the shaded terms is equivalent to the operational semantics without the shaded terms.

3 Type- And Control-Flow Analysis

Our type- and control-flow analysis is presented as an adaptation of the syntax-directed 0CFA, the classic monovariant control-flow analysis [32, Section 3.3], and is given in Figure 5.

The result of our type- and control-flow analysis is a pair of abstract environments. An abstract type environment $\hat{\phi}$ is a map from type variables to sets of abstract types, where an abstract type is simply a (possibly open) type.[5] An abstract value environment $\hat{\rho}$ is a map from variables to sets of abstract values, where an abstract value is simply a (possibly open) recursive function or recursive type abstraction.[6] Abstract type and value environments form complete lattices.

The judgments $\hat{\phi}; \hat{\rho} \models P$, $\hat{\phi}; \hat{\rho} \models e$, and $\hat{\phi}; \hat{\rho} \models v$ assert that a pair of abstract environments is an acceptable type- and control-flow analysis of the program P, expression e, and value v, respectively. An acceptable type- and control-flow analysis is one that soundly approximates the run-time behavior of the program, in a sense made precise in Section 3.1; intuitively, acceptable abstract type and value environments must describe every run-time type and value environment that arises during evaluation of the program.

Ignoring the shaded terms, the constraints asserted by the rules are standard for a monovariant control-flow analysis. The rules for values assert that the value itself is included in the set of abstract values mapped from the μ-bound variable f (corresponding to the $f \mapsto w_f$ binding in the operational semantics at makes the recursive function or recursive type-application available to the expression body) and that the abstract environments are acceptable for the expression body. Each of the rules for `let`-binding expressions `let` x`:`τ_x `=` $\cdots$ `in` e assert that the abstract environments are acceptable for the expression e. The rule for `let`-bindings of values asserts that the abstract environments are acceptable for the value v and that the value v is included in the set of abstract values mapped

[5] We introduce abstract types in preparation for future extensions of the analysis; for example, we may wish to introduce a $\top$ abstract type to represent an unknown type coming from outside the scope of the analysis.

[6] Again, we introduce abstract values in preparation for future extensions of the analysis; for example, we may wish to introduce a $\top$ abstract value to represent an unknown value coming from outside the scope of the analysis or we may wish to introduce $[m, n]$ abstract values to incorporate an interval/range data-flow analysis [6,15].

$$\begin{array}{lrcl}
\textit{Abstract types} & \textit{AType} \ni \hat{\pi} & ::= & \tau \\
\textit{Sets of abstract types} & \mathcal{P}(\textit{AType}) \ni \hat{\Pi} & & \\
\textit{Abstract values} & \textit{AValue} \ni \hat{w} & ::= & v \\
\textit{Sets of abstract values} & \mathcal{P}(\textit{AValue}) \ni \hat{W} & & \\
\\
\textit{Abstract type environments} & \textit{ATEnv} \ni \hat{\phi} & \in & \textit{TyVar} \to \mathcal{P}(\textit{AType}) \\
\textit{Abstract value environments} & \textit{AEnv} \ni \hat{\rho} & \in & \textit{Var} \to \mathcal{P}(\textit{AValue})
\end{array}$$

$$\begin{array}{ll}
\hat{\phi}_1 \sqsubseteq \hat{\phi}_2 \stackrel{\text{def}}{=} \forall \alpha \in \textit{TyVar}.\hat{\phi}_1(\alpha) \subseteq \hat{\phi}_2(\alpha) & \hat{\rho}_1 \sqsubseteq \hat{\rho}_2 \stackrel{\text{def}}{=} \forall x \in \textit{Var}.\hat{\rho}_1(x) \subseteq \hat{\rho}_2(x) \\
\left(\bigsqcup_{i \in I} \hat{\phi}_i\right)(\alpha) \stackrel{\text{def}}{=} \bigcup_{i \in I} \hat{\phi}_i(\alpha) & \left(\bigsqcup_{i \in I} \hat{\rho}_i\right)(x) \stackrel{\text{def}}{=} \bigcup_{i \in I} \hat{\rho}_i(x) \\
\left(\bigsqcap_{i \in I} \hat{\phi}_i\right)(\alpha) \stackrel{\text{def}}{=} \bigcap_{i \in I} \hat{\phi}_i(\alpha) & \left(\bigsqcap_{i \in I} \hat{\rho}_i\right)(x) \stackrel{\text{def}}{=} \bigcap_{i \in I} \hat{\rho}_i(x) \\
\hat{\phi}_\bot(\alpha) \stackrel{\text{def}}{=} \{\} & \hat{\rho}_\bot(x) \stackrel{\text{def}}{=} \{\} \\
\hat{\phi}_\top(\alpha) \stackrel{\text{def}}{=} \textit{AType} & \hat{\rho}_\top(x) \stackrel{\text{def}}{=} \textit{AValue}
\end{array}$$

$$\boxed{\hat{\phi}; \hat{\rho} \vDash v}$$

$$\frac{\hat{\rho}(f) \supseteq \{\mu f{:}\tau_f \,.\, \lambda z{:}\tau_z \,.\, e_b\} \qquad \hat{\phi}; \hat{\rho} \vDash e_b}{\hat{\phi}; \hat{\rho} \vDash \mu f{:}\tau_f \,.\, \lambda z{:}\tau_z \,.\, e_b} \qquad \frac{\hat{\rho}(f) \supseteq \{\mu f{:}\tau_f \,.\, \Lambda\beta \,.\, e_b\} \qquad \hat{\phi}; \hat{\rho} \vDash e_b}{\hat{\phi}; \hat{\rho} \vDash \mu f{:}\tau_f \,.\, \Lambda\beta \,.\, e_b}$$

$$\boxed{\hat{\phi}; \hat{\rho} \vDash e}$$

$$\frac{}{\hat{\phi}; \hat{\rho} \vDash x} \qquad \frac{\hat{\phi}; \hat{\rho} \vDash v \qquad \hat{\rho}(x) \supseteq \{v\} \qquad \hat{\phi}; \hat{\rho} \vDash e}{\hat{\phi}; \hat{\rho} \vDash \texttt{let}\ x{:}\tau_x\ \texttt{=}\ v\ \texttt{in}\ e}$$

$$\frac{\bigwedge_{\mu f{:}\tau_f \,.\, \lambda z{:}\tau_z \,.\, e_b \in \hat{\rho}(x_f)} \begin{pmatrix} \hat{\rho}(z) \supseteq \{\hat{w}_a \in \hat{\rho}(x_a) \mid \ \vdash \hat{\phi} \Rrightarrow \hat{w}_a :\approx \tau_z\} \wedge \\ \hat{\rho}(x) \supseteq \{\hat{w}_b \in \hat{\rho}(\mathsf{last}(e_b)) \mid \ \vdash \hat{\phi} \Rrightarrow \hat{w}_b :\approx \tau_x\} \end{pmatrix} \qquad \hat{\phi}; \hat{\rho} \vDash e}{\hat{\phi}; \hat{\rho} \vDash \texttt{let}\ x{:}\tau_x\ \texttt{=}\ x_f\ x_a\ \texttt{in}\ e}$$

$$\frac{\bigwedge_{\mu f{:}\tau_f \,.\, \Lambda\beta \,.\, e_b \in \hat{\rho}(x_f)} \begin{pmatrix} \hat{\phi}(\beta) \supseteq \{\tau_a\} \wedge \\ \hat{\rho}(x) \supseteq \{\hat{w}_b \in \hat{\rho}(\mathsf{last}(e_b)) \mid \ \vdash \hat{\phi} \Rrightarrow \hat{w}_b :\approx \tau_x\} \end{pmatrix} \qquad \hat{\phi}; \hat{\rho} \vDash e}{\hat{\phi}; \hat{\rho} \vDash \texttt{let}\ x{:}\tau_x\ \texttt{=}\ x_f\ \texttt{[}\tau_a\texttt{]}\ \texttt{in}\ e}$$

$$\boxed{\hat{\phi}; \hat{\rho} \vDash P}$$

$$\frac{\hat{\phi}; \hat{\rho} \vDash e}{\hat{\phi}; \hat{\rho} \vDash e}$$

Fig. 5. Type- and Control-Flow Analysis of ANF System F

$\boxed{\Delta \vdash \hat{\phi} \Rrightarrow \hat{\pi} \approxeq \hat{\pi}}$

$$\frac{\hat{\pi}'_1 \in \hat{\phi}(\alpha_1) \qquad \emptyset \vdash \hat{\phi} \Rrightarrow \hat{\pi}'_1 \approxeq \hat{\pi}_2}{\Delta \vdash \hat{\phi} \Rrightarrow \alpha_1 \approxeq \hat{\pi}_2} \qquad \frac{\hat{\pi}'_2 \in \hat{\phi}(\alpha_2) \qquad \emptyset \vdash \hat{\phi} \Rrightarrow \hat{\pi}_1 \approxeq \hat{\pi}'_2}{\Delta \vdash \hat{\phi} \Rrightarrow \hat{\pi}_1 \approxeq \alpha_2}$$

$$\frac{\Delta \vdash \hat{\phi} \Rrightarrow \tau_{z1} \approxeq \tau_{z2} \qquad \Delta \vdash \hat{\phi} \Rrightarrow \tau_{b1} \approxeq \tau_{b2}}{\Delta \vdash \hat{\phi} \Rrightarrow \tau_{z1} \to \tau_{b1} \approxeq \tau_{z2} \to \tau_{b2}}$$

$$\frac{\Delta(\alpha_1) = \star \qquad \Delta(\alpha_2) = \star \qquad \alpha_1 = \alpha_2}{\Delta \vdash \hat{\phi} \Rrightarrow \alpha_1 \approxeq \alpha_2} \qquad \frac{\Delta, \alpha{:}\star \vdash \hat{\phi} \Rrightarrow \tau_{b1} \approxeq \tau_{b2}}{\Delta \vdash \hat{\phi} \Rrightarrow \forall\alpha.\ \tau_{b1} \approxeq \forall\alpha.\ \tau_{b2}}$$

$\boxed{\vdash \hat{\phi} \Rrightarrow \hat{w} :\approxeq \hat{\pi}}$

$$\frac{\emptyset \vdash \hat{\phi} \Rrightarrow \tau_f \approxeq \hat{\pi}}{\vdash \hat{\phi} \Rrightarrow \mu f{:}\tau_f\,.\,\lambda z{:}\tau_z\,.\,e_b :\approxeq \hat{\pi}} \qquad \frac{\emptyset \vdash \hat{\phi} \Rrightarrow \tau_f \approxeq \hat{\pi}}{\vdash \hat{\phi} \Rrightarrow \mu f{:}\tau_f\,.\,\Lambda\beta\,.\,e_b :\approxeq \hat{\pi}}$$

Fig. 6. Analysis-time Type Compatibility

from the variable x. The rule for `let`-bindings of non-tail function applications assert that, for all recursive functions in the set of abstract values mapped from the variable x_f, the abstract values mapped from the actual argument x_a flow to the formal argument z and the abstract values from the function result $\mathsf{last}(e_b)$ flow to the receiving variable x. Similarly, the rule for `let`-bindings of non-tail type applications assert that, for all recursive type abstractions in the set of abstract values mapped from the variable x_f, the actual type argument τ_a flows to the formal type argument β and the abstract values from the function result $\mathsf{last}(e_b)$ flow to the receiving variable x.

We now consider the shaded terms in the third and fourth rules of the $\hat{\phi}; \hat{\rho} \models e$ judgment and the judgments and rules in Figure 6. In essence, the shaded terms perform a kind of analysis-time type checking at the point where there is a non-local flow of abstract values. In the third rule, each abstract argument $\hat{w}_a$ that flows to the formal argument z must have an abstract type compatible with τ_z, the static type of formal argument; this is the abstract analogue of the run-time type equality condition in the function application rule of the operational semantics. Similarly, in the third and fourth rules, each abstract result $\hat{w}_b$ that flow to the receiving variable x must have an abstract type compatible with τ_x, the static type of the receiving variable; this is the abstract analogue of the run-time type equality condition in the return rule of the operational semantics.

The rules for the judgment $\vdash \hat{\phi} \Rrightarrow \hat{w} :\approxeq \pi$ simply form the abstract type of the recursive function or recursive type abstraction from the (static, syntactic) type of the μ-bound variable. The judgment $\Delta \vdash \hat{\phi} \Rrightarrow \hat{\pi}_1 \equiv \hat{\pi}_2$ asserts that the abstract types $\hat{\pi}_1$ and $\hat{\pi}_2$ are compatible under $\hat{\phi}$ and Δ. The first and second rules expand a Λ-bound type variable according to the (global) abstract type environment; note that these rules must "guess" a satisfying abstract type from among the set of abstract types mapped from the type variable, unlike the corresponding rules in the judgment $\Delta \vdash \pi_1 \equiv \pi_2$, where the expansion is uniquely determined by the (local) run-time type environment. The third rule asserts that two function types are compatible when their argument types are compatible and their result types are compatible. The fifth rule asserts that two universal types (sharing the same $\forall$-bound type variable via α conversion) are compatible when their range types are compatible. The fourth rule asserts that two $\forall$-bound type variables are compatible when they are the same type variable. Note that the first and second rules assert the compatibility of the expanded abstract type under the empty type-variable context; this is because the expanded abstract type is not in the scope of the $\forall$-bound type variables appearing in Δ.

3.1 Flow Soundness

We now show that every acceptable (with respect to a given program) pair of abstract environments soundly approximates the run-time behavior of the program. To formalize the approximation, we introduce "shallow" abstraction functions that take run-time types and values to abstract types and values and that take run-time type and value environments to abstract type and value environments:[7]

$$|\cdot| :: \mathit{RType} \to \mathit{AType} \qquad |\langle \tau; \phi \rangle| = \tau$$

$$|\cdot| :: \mathit{RValue} \to \mathit{AValue} \qquad |\langle v; \phi; \rho \rangle| = v$$

$$|\cdot| :: \mathit{RTEnv} \to \mathit{ATEnv} \qquad |\phi|(\alpha) = \begin{cases} \{\} & \text{if } \alpha \notin \mathsf{dom}(\phi) \\ \{|\pi|\} & \text{if } \phi(\alpha) = \pi \end{cases}$$

$$|\cdot| :: \mathit{REnv} \to \mathit{AEnv} \qquad |\rho|(x) = \begin{cases} \{\} & \text{if } x \notin \mathsf{dom}(\rho) \\ \{|w|\} & \text{if } \rho(x) = w \end{cases}$$

Theorem 3 (Flow Soundness).
If $\hat{\phi}; \hat{\rho} \vDash e$ and $\langle e; \emptyset; \emptyset; \bullet \rangle \longrightarrow^* \langle e'; \phi'; \rho'; \kappa' \rangle$, then $|\phi'| \sqsubseteq \hat{\phi}$ and $|\rho'| \sqsubseteq \hat{\rho}$.

A proof, using a Preservation (aka, subject reduction) theorem, is given in a companion technical report [11]. In addition to the judgments of Figure 5, we introduce judgments that assert the acceptability of abstract environments with respect to run-time types ($\hat{\phi} \vDash \pi$), run-time type environments ($\hat{\phi} \vDash \phi$), run-time values ($\hat{\phi}; \hat{\rho} \vDash w$), run-time value environments ($\hat{\phi}; \hat{\rho} \vDash \rho$), continuations ($\hat{\phi}; \hat{\rho} \vDash x \rhd \kappa$), and states ($\hat{\phi}; \hat{\rho} \vDash \varsigma$). The judgments $\hat{\phi} \vDash \phi$ and $\hat{\phi}; \hat{\rho} \vDash \rho$ assert that the abstract environments are "deep" abstractions of the run-time environments.

[7] These abstraction functions are "shallow" in the sense that they do not abstract and join the embedded run-time type and value environments of run-time types and values.

Preservation is straightforward, simplified by the explicit run-time type equality conditions in the operational semantics and the following lemma, which establishes that type compatibility soundly approximates type equality:

Lemma 4 (Run-Time Type Equality implies Analysis-Time Type Compatibility)
If $\hat{\phi} \vDash \phi_1$, $\hat{\phi} \vDash \phi_2$, and $\emptyset \vdash \langle \tau_1 ; \phi_1 \rangle \equiv \langle \tau_2 ; \phi_2 \rangle$, then $\emptyset \vdash \hat{\phi} \Rrightarrow \tau_1 \approxeq \tau_2$.

It is perhaps surprising to note that Flow Soundness of our type- and control-flow analysis does not require a well-typed source program. In essence, the explicit run-time type equality conditions in the instrumented operational semantics ensure that a machine state is “just well-typed enough” to preserve the acceptability of abstract environments across a taken machine transition. A well-typed source program and Type Soundness ensure that machine transitions may be repeatedly taken. This suggests an alternate presentation in which we adopt an uninstrumented operational semantics (i.e., Figure 3 without the shaded terms) and require a well-typed source program for Flow Soundness; Progress (for Type Soundness) would become straightforward, but Preservation (for Flow Soundness) would require a lemma (essentially, a chaining of Lemma 2 and Lemma 4) that establishes that the abstractions of two run-time types may be judged compatible if their induced expansions are syntactically equal, forgoing the run-time type equality judgment entirely. The necessary preconditions for this lemma would be obtained from the well-typedness of the machine state undergoing transition.

3.2 Existence of Minimum, Finite Flows

As is typical for a flow analysis, for a given program, there may be many acceptable pairs of abstract environments. One intuitively acceptable pair of abstract type- and value-environments is the one that maps every Λ-bound type variable that occurs in the program to the set of types that occur in the program and that maps every `let`-, μ-, and λ-bound variable that occurs in the program to the set of values that occur in the program:

$$\hat{\phi}^P_\top(\alpha) = \begin{cases} \{\} & \text{if } \alpha \notin \mathit{TyVar}_P \\ \mathit{Type}_P & \text{if } \alpha \in \mathit{TyVar}_P \end{cases} \qquad \hat{\rho}^P_\top(x) = \begin{cases} \{\} & \text{if } x \notin \mathit{Var}_P \\ \mathit{Value}_P & \text{if } x \in \mathit{Var}_P \end{cases}$$

Note that these abstract environments are “finite”, in the sense that they map a finite set of elements to finite sets (and the remaining elements to empty sets).

The following theorem establishes that minimum, finite acceptable abstract environments exist for every program:

Theorem 5 (Minimum, Finite Flows Exist).
For all programs P,
there exist minimum abstract environments $\hat{\phi}_{\min}$ and $\hat{\rho}_{\min}$
such that $\hat{\phi}_{\min} \sqsubseteq \hat{\phi}^P_\top$, $\hat{\rho}_{\min} \sqsubset \hat{\rho}^P_\top$, and $\hat{\phi}_{\min}; \hat{\rho}_{\min} \vDash P$.

A proof is given in a companion technical report [11], first showing that $\hat{\phi}^P_\top$ and $\hat{\rho}^P_\top$ are an acceptable pair of abstract environments and then showing that the greatest lower bound of a (possibly infinite) set of acceptable pairs of abstract environments is an acceptable pair of abstract environments. Furthermore, for a given program P, we may restrict ourselves to considering abstract type environments $\hat{\phi}^P \in \mathit{ATEnv}_P = \mathit{TyVar}_P \to \mathcal{P}(\mathit{Type}_P)$ and abstract value environments $\hat{\rho}^P \in \mathit{AEnv}_P = \mathit{Var}_P \to \mathcal{P}(\mathit{Value}_P)$, which form finite, complete lattices.

3.3 Computability of Minimum, Finite Flows

While Theorem 5 establishes that minimum, finite acceptable abstract environments exist for every program, we would like such abstract environments to be computable. The key concern is the decidability of the $\Delta \vdash \hat{\phi} \Rrightarrow \tau_1 \approx \tau_2$ judgment. Even simply verifying that a pair of abstract environments is acceptable for a given program requires showing that constraints of the form $\hat{\rho}(z) \supseteq \{\hat{w} \in \hat{\rho}(x) \mid \ \vdash \hat{\phi} \Rrightarrow \hat{w} :\approx \tau_z\}$ are satisfied; this, in turn, requires showing, for each an abstract value $\hat{w}$ that is an element of $\hat{\rho}(x)$ but not an element of $\hat{\rho}(z)$, that the judgment $\vdash \hat{\phi} \Rrightarrow \hat{w} :\approx \tau_z$ is not derivable.[8]

Due to “recursion” in the abstract type environment, whereby a type variable may be mapped to a set of abstract types in which the type variable itself occurs free, we cannot exhaustively apply the first and second rules of the $\Delta \vdash \hat{\phi} \Rrightarrow \tau_1 \approx \tau_2$ judgment in order to search for a derivation. Consider deciding whether or not $\emptyset \vdash \hat{\phi}^\ddagger \Rrightarrow \mathsf{int} \to \beta \approx \alpha$ is derivable where $\hat{\phi}^\ddagger(\alpha) = \{\mathsf{int} \to \mathsf{int}, \mathsf{int} \to \alpha\}$ and $\hat{\phi}^\ddagger(\beta) = \{\mathsf{int} \to \mathsf{bool}, \mathsf{int} \to \beta\}$.[9] We must avoid getting “stuck” exploring the infinite non-derivation:

$$\dfrac{\mathsf{int} \to \alpha \in \hat{\phi}^\ddagger(\alpha) \qquad \dfrac{\dfrac{}{\emptyset \vdash \hat{\phi}^\ddagger \Rrightarrow \mathsf{int} \approx \mathsf{int}} \qquad \dfrac{\mathsf{int} \to \beta \in \hat{\phi}^\ddagger(\beta) \qquad \dfrac{\vdots}{\emptyset \vdash \hat{\phi}^\ddagger \Rrightarrow \mathsf{int} \to \beta \approx \alpha}}{\emptyset \vdash \hat{\phi}^\ddagger \Rrightarrow \beta \approx \alpha}}{\emptyset \vdash \hat{\phi}^\ddagger \Rrightarrow \mathsf{int} \to \beta \approx \mathsf{int} \to \alpha}}{\emptyset \vdash \hat{\phi}^\ddagger \Rrightarrow \mathsf{int} \to \beta \approx \alpha}$$

To circumvent this issue, we take inspiration from the theory and implementation of regular-tree grammars [12,2,7], which has been used extensively for flow analysis [23,22,17,16] (including type inference [28,3]), but whereas previous work has applied regular-tree grammars to the analysis of values, we apply regular-tree grammars to the analysis of types.

Given an abstract type environment $\hat{\phi}$, we interpret it as a regular-tree grammar as follows: $\mathsf{dom}(\hat{\phi})$ is the set of non-terminals and

[8] Note, however, that this does not require showing, for each abstract value $\hat{w}$ that is an element of both $\hat{\rho}(x)$ and $\hat{\rho}(z)$, that the judgment $\vdash \hat{\phi} \Rrightarrow \hat{w} :\approx \tau_z$ is derivable; the constraint is satisfied whether or not the judgment is derivable.

[9] It is not derivable.

$\{\alpha \Rightarrow \hat{\pi} \mid \alpha \in \mathsf{dom}(\hat{\phi}) \wedge \hat{\pi} \in \hat{\phi}(\alpha)\}$ is the set of productions. The language $\mathcal{L}_{\hat{\phi}}(\hat{\pi})$ generated by the grammar $\hat{\phi}$ for the starting term $\hat{\pi}$ is $\mathcal{L}_{\hat{\phi}}(\hat{\pi}) \stackrel{\text{def}}{=} \{\hat{\pi}' \in \overline{\mathit{AType}} \mid \hat{\pi} \Rightarrow^*_{\hat{\phi}} \hat{\pi}'\}$, where $\overline{\mathit{AType}}$ is the set of closed abstract types and $\hat{\pi}_1 \Rightarrow_{\hat{\phi}} \hat{\pi}_2$ is the relation that (capture-avoidingly) substitutes for one (free) non-terminal of $\hat{\pi}_1$ the right-hand side of one of its productions to obtain $\hat{\pi}_2$.[10]

Intuitively, $\emptyset \vdash \hat{\phi}^{\ddagger} \Rrightarrow \mathsf{int} \to \beta \approxeq \alpha$ is not derivable because

$$\begin{aligned}\mathcal{L}_{\hat{\phi}^{\ddagger}}(\mathsf{int} \to \beta) &= \{\mathsf{int} \to \mathsf{int} \to \mathsf{bool}, \mathsf{int} \to \mathsf{int} \to \mathsf{int} \to \mathsf{bool}, \ldots\} \\ \mathcal{L}_{\hat{\phi}^{\ddagger}}(\alpha) &= \{\mathsf{int} \to \mathsf{int}, \mathsf{int} \to \mathsf{int} \to \mathsf{int}, \mathsf{int} \to \mathsf{int} \to \mathsf{int} \to \mathsf{int}, \ldots\}\end{aligned}$$

and $\mathcal{L}_{\hat{\phi}^{\ddagger}}(\mathsf{int} \to \beta) \cap \mathcal{L}_{\hat{\phi}^{\ddagger}}(\alpha) = \emptyset$; there is no closed type that is generated by $\hat{\phi}^{\ddagger}$ from both $\mathsf{int} \to \beta$ and α. Formally,

Theorem 6 (**Analysis-Time Type Compatibility iff Languages Intersect**). *$\emptyset \vdash \hat{\phi} \Rrightarrow \hat{\pi}_1 \approxeq \hat{\pi}_2$ if and only if $\mathcal{L}_{\hat{\phi}}(\hat{\pi}_1) \cap \mathcal{L}_{\hat{\phi}}(\hat{\pi}_2) \neq \emptyset$.*

A proof is given in a companion technical report [11].

An immediate corollary to Theorem 6 is the the decidability of type compatibility, since regular-tree grammars are closed under intersection and the emptiness of a regular-tree grammar is decidable [12,28]. In turn, we have that the minimum, finite acceptable abstract environments are computable for every program. Furthermore, given a program P, it is straightforward to read the analysis of Figure 5 as defining a monotone function from abstract environments to abstract environments; the "input" abstract environments are used for terms of the form $\hat{w} \in \hat{\rho}(x)$ and $\vdash \hat{\phi} \Rrightarrow \hat{w} :\approxeq \tau$, while the "output" abstract environments are formed from the "input" abstract environments joined with $\hat{\phi}_\perp[\beta \mapsto \hat{\Pi}]$ and $\hat{\phi}_\perp[x \mapsto \hat{W}]$ for terms of the form $\hat{\phi}(\beta) \supseteq \hat{\Pi}$ and $\hat{\rho}(x) \supseteq \hat{W}$. The least fixed point of this monotone function, computable using a standard fixed-point computation, is the minimum, finite acceptable pair of abstract environments for the program P. Further considerations regarding the implementation of our type- and control-flow analysis are given in Section 5.

We briefly sketch implementations of testing emptiness and intersection of regular-tree grammars, based on those given by Aiken and Murphy [2]. Recall that, for a given program P, we may restrict ourselves to finite abstract type environments $\hat{\phi}^P \in \mathit{ATEnv}_P$.

We define the function Ψ as follows:

$$\begin{aligned}\Psi &:: \mathit{AEnv} \to (\mathit{TyVar} \to \mathbb{B}) \\ \Psi(\hat{\phi}) &= \operatorname{lfp} F \\ &\quad \text{where} \qquad F :: (\mathit{TyVar} \to \mathbb{B}) \to (\mathit{TyVar} \to \mathbb{B}) \\ &\qquad\qquad F(\psi)(\alpha) = \begin{cases}\top & \text{if } \exists \hat{\pi} \in \hat{\phi}(\alpha).\ \forall \beta \in \mathsf{FTV}(\hat{\pi}).\ \psi(\beta) = \top \\ \bot & \text{if } \forall \hat{\pi} \in \hat{\phi}(\alpha).\ \exists \beta \in \mathsf{FTV}(\hat{\pi}).\ \psi(\beta) = \bot\end{cases}\end{aligned}$$

[10] There are some subtleties in the treatment of $\forall$-bound type variables, which are perhaps best dealt with by adopting a locally-nameless representation [4] for types.

If $\hat{\phi}$ is finite, then $\Psi(\hat{\phi})$ is computable using a standard fixed-point computation. The language $\mathcal{L}_{\hat{\phi}}(\hat{\pi})$ is non-empty if and only if $\forall \beta \in \mathsf{FTV}(\hat{\pi}).\ \psi(\beta) = \top$.

In order to intersect the languages generated by the finite regular-tree grammar $\hat{\phi}$ for the starting terms $\hat{\pi}_1$ and $\hat{\pi}_2$, we extend $\hat{\phi}$ with finitely many additional non-terminals and productions to obtain $\hat{\phi}^\star$ and generate a starting term $\hat{\pi}^\star$ such that $\mathcal{L}_{\hat{\phi}}(\hat{\pi}_1) \cap \mathcal{L}_{\hat{\phi}}(\hat{\pi}_2) = \mathcal{L}_{\hat{\phi}^\star}(\hat{\pi}^\star)$. The idea is that each new non-terminal represents the intersection of a type variable in $\mathsf{dom}(\hat{\phi})$ and a type; a global mapping from pairs of type variables and types to new non-terminals is maintained to ensure that the same new non-terminal is used whenever the same pair is encountered.

To illustrate the technique, consider intersecting the languages generated by $\hat{\phi}^\ddagger$ for the starting terms $\mathsf{int} \to \beta$ and α. First, extend the grammar with a new non-terminal Z and no productions (i.e., extend $\hat{\phi}^\ddagger$ with the mapping $Z \mapsto \{\}$); the non-terminal Z will serve as starting term for an empty language. We are trying to intersect $\mathsf{int} \to \beta$ and α; since α is a non-terminal, generate a new non-terminal A_0 mapped from the pair $\langle \mathsf{int} \to \beta; \alpha \rangle$, add the triple $\langle A_0; \{\mathsf{int} \to \beta\}; \hat{\phi}^\ddagger(\alpha) \rangle$ to a work list; and return A_0 as the result of intersecting $\mathsf{int} \to \beta$ and α. The work list contains new non-terminals whose productions should be generated by intersecting all pairs of elements from the two sets. Therefore, we next add productions corresponding to $A_0 \Rightarrow \mathsf{int} \to \beta \oslash \mathsf{int} \to \mathsf{int}$ and $A_0 \Rightarrow \mathsf{int} \to \beta \oslash \mathsf{int} \to \alpha\}$. Intersecting $\mathsf{int} \to \beta$ and $\mathsf{int} \to \mathsf{int}$ generates a new non-terminal A_1 mapped from $\langle \beta; \mathsf{int} \rangle$, adds $\langle A_1; \hat{\phi}^\ddagger(\beta); \{\mathsf{int}\} \rangle$ to the worklist, and returns $\mathsf{int} \to A_1$. Intersecting $\mathsf{int} \to \beta$ and $\mathsf{int} \to \alpha$ generates a new non-terminal A_2 mapped from $\langle \beta; \alpha \rangle$, adds $\langle A_1; \hat{\phi}^\ddagger(\beta); \hat{\phi}^\ddagger(\alpha) \rangle$ to the worklist, and returns $\mathsf{int} \to A_2$. Therefore, we extend with the mapping $A_0 \mapsto \{\mathsf{int} \to A_1\} \cup \{\mathsf{int} \to A_2\}$. Returning to the work list, we next add productions corresponding to $A_1 \Rightarrow \mathsf{int} \to \mathsf{bool} \oslash \mathsf{int}$ and $A_1 \Rightarrow \mathsf{int} \to \beta \oslash \mathsf{int}$. Intersecting $\mathsf{int} \to \mathsf{bool}$ and int returns Z (since clearly the intersection of the languages generated from these two starting terms is empty), as does intersecting $\mathsf{int} \to \beta$ and int; therefore, we extend with the mapping $A_1 \mapsto \{Z\} \cup \{Z\}$. Returning to the work list, we next add productions corresponding to $A_2 \Rightarrow \mathsf{int} \to \mathsf{bool} \oslash \mathsf{int} \to \mathsf{int}$ (this intersection returns Z), $A_2 \Rightarrow \mathsf{int} \to \mathsf{bool} \oslash \mathsf{int} \to \alpha$ (this intersection generates a new non-terminal A_3 mapped from $\langle \mathsf{bool}; \alpha \rangle$, adds $\langle A_3; \{\mathsf{bool}\}; \hat{\phi}^\ddagger(\alpha) \rangle$ to the worklist, and returns $\mathsf{int} \to A_3$), $A_2 \Rightarrow \mathsf{int} \to \beta \oslash \mathsf{int} \to \mathsf{int}$ (this intersection returns $\mathsf{int} \to A_1$, using the global map), and $A_2 \Rightarrow \mathsf{int} \to \beta \oslash \mathsf{int} \to \alpha$ (this intersection returns $\mathsf{int} \to A_2$); therefore, we extend with the mapping $A_2 \mapsto \{Z\} \cup \{\mathsf{int} \to A_3\} \cup \{\mathsf{int} \to A_1\} \cup \{\mathsf{int} \to A_2\}$. Finally, we add productions corresponding to $A_3 \Rightarrow \mathsf{bool} \oslash \mathsf{int} \to \mathsf{int}$ (this intersection returns Z) and $A_3 \Rightarrow \mathsf{bool} \oslash \mathsf{int} \to \alpha$ (this intersection returns Z); therefore, we extend with the mapping $A_3 \mapsto \{Z\} \cup \{Z\}$. In summary, we have

Global map	New productions
$\langle \mathsf{int} \to \beta; \alpha \rangle \mapsto A_0$	$A_0 \mapsto \{\mathsf{int} \to A_1\} \cup \{\mathsf{int} \to A_2\}$
$\langle \beta; \mathsf{int} \rangle \mapsto A_1$	$A_1 \mapsto \{Z\} \cup \{Z\}$
$\langle \beta; \alpha \rangle \mapsto A_2$	$A_2 \mapsto \{Z\} \cup \{\mathsf{int} \to A_3\} \cup \{\mathsf{int} \to A_1\} \cup \{\mathsf{int} \to A_2\}$
$\langle \mathsf{bool}; \alpha \rangle \mapsto A_3$	$A_3 \mapsto \{Z\} \cup \{Z\}$
	$Z \mapsto \{\}$

To conclude, we return $\hat{\phi}^\star$ equal to $\hat{\phi}^\ddagger$ extended with the new productions and $\hat{\pi}^\star$ equal to A_0. Finally, note that $\Psi(\hat{\phi}^\star)(\hat{\pi}^\star) = \perp$, confirming that $\mathcal{L}_{\hat{\phi}^\ddagger}(\mathsf{int} \to \beta) \cap \mathcal{L}_{\hat{\phi}^\ddagger}(\alpha) = \emptyset$.

4 Related Work

There is surprisingly little work on control-flow analyses for statically-typed languages with polymorphic types. Control-flow analyses have typically been formulated for dynamically- or simply-typed languages.[11] Production implementations of control-flow analyses for Standard ML, a language with rank-1 polymorphism (i.e., "let"-polymorphism), typically handle the polymorphism either by monomorphisation [5] (explicitly eliminating the polymorphism before analysis) or by polyvariance [16] (implicitly eliminating the polymorphism during analysis).

The most closely related work is the "Type-Directed Flow Analysis for Typed Intermediate Languages" of Jagannathan, Weeks, and Wright [20], which describes a framework for polyvariant flow analyses of Λ_i, the predicative subset of System F extended with recursive procedures. A specific analysis called $S_{\mathcal{RT}}$ uses types to control polyvariance; essentially, $S_{\mathcal{RT}}$ introduces a distinct polyvariance context for each closed type at which a polymorphic function is applied. Furthermore, $S_{\mathcal{RT}}$ respects types, meaning that if $\hat{v} \in F(x)$ (the abstract value $\hat{v}$ is assigned to x by the analysis) and $x : \sigma$ (the type scheme σ is assigned to x by the type system), then $[\![\hat{v}]\!] \subseteq [\![\sigma]\!]$, where $[\![\cdot]\!]$ denotes a set of values. Unfortunately, $S_{\mathcal{RT}}$ does not terminate on programs that use polymorphic recursion [30,18,24]; such programs may instantiate a polymorphic function at an infinite number of closed types during execution. In contrast, our type- and control-flow analysis is computable for all programs in (impredicative) System F extended with recursive functions.

Another closely related work is the "Type-sensitive Control-Flow Analysis" of Reppy [37], which describes an extension of Serrano's version of 0CFA [42] that uses a program's type information to compute more precise results. Serrano's and Reppy's analyses are modular and use an abstract value $\top$ to denote an unknown value; variables bound outside the unit of analysis are assigned $\top$,

[11] Again, we draw a distinction between flow analyses expressed as sophisticated type systems and flow analyses of languages with sophisticated type systems.

as are the parameters of functions that escape the unit of analysis. Reppy's insight is that values of an abstract type can only be created within their defining module; hence, "unknown" values of the abstract type can be soundly approximated by the set of escaping values of the abstract type (a subset of the set of values of the abstract type created within the defining module). This leads to a type-indexed family of abstract values for unknown values, in addition to the $\top$ abstract value. Reppy's analysis is formulated for a simply-typed language with top-level abstract types; he suggests extending the analysis to a language with polymorphism by mapping type variables to the $\top$ abstract value. Our type- and control-flow analysis is a whole-program analysis, but has a more precise treatment of type variables.

5 Future Work

There are many directions for future work.

While Section 3.3 established the computability of the minimum, finite acceptable pair of abstract environments for every program, we would our type- and control-flow analysis to be efficiently computable. A popular approach for computing control-flow analyses is as a constraint-based analysis [1]; an initial phase generates constraints that a solution to the analysis must satisfy, while a subsequent phase solves the constraints.[12] The syntax-directed 0CFA that we adapt to our type- and control-flow analysis has an $O(n^3)$ algorithm following this approach [32, Section 3.4]. However, algorithms for solving a set of constraints are sensitive to the syntax of constraints; the filtering of sets by the derivability of the type-compatibility judgment may prove problematic, especially since the derivability of the type-compatibility judgment depends upon the abstract type environment.

Independent of the overall approach to computing our type- and control-flow analysis, it seems clear that we will need to efficiently decide the derivability of type-compatibility judgments with respect to abstract type environments. Section 3.3 established that this decision could be made by intersecting and testing the emptiness of regular-tree grammars. Both operations are (worst-case) quadratic time in the size of the regular-tree grammar. Aiken and Murphy [2, Section 4] suggest maintaining a regular-tree grammar with an invariant that makes testing the emptiness (of a non-terminal) constant time. Aiken and Murphy [2, Section 5.3] also suggest that the algorithm given previously, which generates only the intersections necessary to express the result, performs well in the typical case. We further observe that, for a fixed abstract type environment, we can maintain the global map from pairs of type variables and types to new non-terminals (where each new non-terminal represents the intersection of the expansions of the type variable under the abstract type environment and the type) across decisions of the derivability of type-compatibility. Hence, the

[12] More sophisticated approaches exist where additional constraints are generated during the solving phase.

(worst-case) quadratic time bounds all queries with a given abstract type environment, not each query. We may also be able to exploit the fact that we are computing the emptiness of an intersection of regular-tree grammars and are not interested in the intersection itself.

Another direction of future work is to extend the type- and control-flow analysis to handle unknown and escaping values. It should be straightforward to introduce a $\top$ abstract type and a $\top$ abstract value; conservatively, the $\top$ abstract type should be judged compatible with any other abstract type. A more interesting direction is to consider primitives that make essential use of higher-rank polymorphism, such as Haskell's `runST` [25,26].

Yet another direction is to extend the monovariant type- and control-flow analysis to a polyvariant analysis.

Finally, we would like to extend type- and control-flow analysis to languages with even more sophisticated type systems. Of particular interest is System F with guarded algebraic data types (GADTs), as we are interested in combining the flow-directed defunctionalization of Cejtin, Jagannathan, and Weeks [5] with the polymorphic typed defunctionalization of Pottier and Gauthier [35]. Also of interest is System F_ω, the higher-order polymorphic lambda-calculus: System F_ω has used as a target language for the elaboration of a full-featured, higher-order ML-like module language [41] and System F_ω extended with type equality coercions [46] is used as a typed intermediate language in the Glasgow Haskell Compiler (GHC).

6 Conclusion

References

1. Aiken, A.: Introduction to set constraint-based program analysis. Science of Computer Programming 35(2-3), 79–111 (1999)
2. Aiken, A., Murphy, B.R.: Implementing regular tree expressions. In: Hughes, J. (ed.) Proceedings of the Fifth ACM Conference on Functional Programming Languages and Computer Architecture. Lecture Notes in Computer Science, vol. 523, pp. 427–447. Springer-Verlag, Cambridge, Massachusetts (Aug 1991)
3. Aiken, A., Murphy, B.R.: Static type inference in a dynamically typed language. In: Cartwright, R.C. (ed.) Proceedings of the Eighteenth Annual ACM Symposium on Principles of Programming Languages. pp. 279–290. Orlando, Florida (Jan 1991)
4. Aydemir, B., Charguéraud, A., Pierce, B.C., Pollack, R., Weirich, S.: Engineering formal metatheory. In: Necula, G.C., Wadler, P. (eds.) Proceedings of the 35th Annual ACM Symposium on Principles of Programming Languages. pp. 3–15. ACM, San Francisco, California (Jan 2008)
5. Cejtin, H., Jagannathan, S., Weeks, S.: Flow-directed closure conversion for typed languages. In: Smolka [45], pp. 56–71
6. Cousot, P., Cousot, R.: Static determination of dynamic properties of programs. In: Robinet, B. (ed.) Proceedingsof the Second International Symposium on Programming. pp. 106–130. Paris, France (Apr 1976)
7. Cousot, P., Cousot, R.: Formal language, grammar and set-constraint-based program analysis by abstract interpretation. In: Peyton Jones, S. (ed.) Proceedings of

the Seventh ACM Conference on Functional Programming Languages and Computer Architecture. pp. 170–181. La Jolla, California (Jun 1995)
8. Danvy, O.: Three steps for the CPS transformation. Tech. Rep. CIS-92-2, Kansas State University, Manhattan, Kansas (1991)
9. Faxén, K.F.: Polyvariance, polymorphism and flow analysis. In: Dam, M. (ed.) Selected papers from the 5th LOMAPS Workshop on Analysis and Verification of Multiple-Agent Languages. Lecture Notes in Computer Science, vol. 1192, pp. 260–278. Springer-Verlag (1997)
10. Flanagan, C., Sabry, A., Duba, B.F., Felleisen, M.: The essence of compiling with continuations. In: Cartwright, R. (ed.) Proceedings of the ACM SIGPLAN 1993 Conference on Programming Languages Design and Implementation. pp. 237–247. ACM, Albuquerque, New Mexico (Jun 1993)
11. Fluet, M.: A type- and control-flow analysis for System F. Tech. rep., Rochester Institute of Technology (August 2012), `http://www.cs.rit.edu/~mtf/research/tcfa/IFL12/techrpt.pdf`
12. Gecseg, F., Steinby, M.: Tree Automata. Akademiai Kiado, Budapest, Hungary (1984)
13. Girard, J.Y.: Une extension de l'interpretation de Gödel à l'analyse, et son application à l'élimination des coupures dans l'analyse et la théorie des types. In: Fenstad, J.E. (ed.) Proceedings of the 2nd Scandinavian Logic Symposium. pp. 63–92. Amsterdam, Netherlands (1971)
14. Harper, R., Morrisett, G.: Compiling polymorphism using intensional type analysis. In: Lee, P. (ed.) Proceedings of the 22nd Annual ACM Symposium on Principles of Programming Languages. pp. 130–141. ACM, ACM, San Francisco, California (Jan 1995)
15. Harrison III, W.L.: Compiler analysis of the value ranges for variables. IEEE Transactions on Software Engineering SE-3(3), 243–250 (May 1977)
16. Heintze, N.: Set-based program analysis of ML programs. In: Talcott, C.L. (ed.) Proceedings of the 1994 ACM Conference on Lisp and Functional Programming. pp. 306–317. LISP Pointers, Vol. VII, No. 3, Orlando, Florida (Jun 1994)
17. Heintze, N., Jaffar, J.: A finite presentation theorem for approximating logic programs. In: Hudak, P. (ed.) Proceedings of the Seventeenth Annual ACM Symposium on Principles of Programming Languages. pp. 197–209. San Francisco, California (Jan 1990)
18. Henglein, F.: Type inference with polymorphic recursion. ACM Transactions on Programming Languages and Systems 15(2), 253–289 (1993)
19. Holdermans, S., Hage, J.: Polyvariant flow analysis with higher-ranked polymorphic types and higher-order effect operators. In: Weirich, S. (ed.) Proceedings of the Fifteenth ACM SIGPLAN International Conference on Functional Programming (ICFP'10). pp. 63–74. ACM, Baltimore, Maryland (Sep 2010)
20. Jagannathan, S., Weeks, S., Wright, A.K.: Type-directed flow analysis for typed intermediate languages. In: Hentenryck, P.V. (ed.) Static Analysis, 4th International Symposium, SAS '97. Lecture Notes in Computer Science, vol. 1302, pp. 232–249. Springer-Verlag, Paris, France (Sep 1997)
21. Jones, N.D.: Flow analysis of lambda expressions (preliminary version). In: Even, S., Kariv, O. (eds.) Automata, Languages and Programming, 8th Colloquium, Acre (Akko). Lecture Notes in Computer Science, vol. 115, pp. 114–128. Springer-Verlag, Israel (Jul 1981)
22. Jones, N.D.: Flow analysis of lazy higher-order functional programs. In: Abramsky, S., Hankin, C. (eds.) Abstract Interpretation of Declarative Languages, pp. 103–122. Ellis Horwood (1987)

23. Jones, N.D., Muchnick, S.S.: Flow analysis and optimization of LISP-like structures. In: Rosen, B.K. (ed.) Proceedings of the Sixth Annual ACM Symposium on Principles of Programming Languages. pp. 244–256. San Antonio, Texas (Jan 1979)
24. Kfoury, A., Tiuryn, J., Urzyczyn, P.: Type reconstruction in the presence of polymorphic recursion. ACM Transactions on Programming Languages and Systems 15(2), 290–311 (1993)
25. Launchbury, J., Peyton Jones, S.: Lazy functional state threads. In: Sarkar, V. (ed.) Proceedings of the ACM SIGPLAN 1994 Conference on Programming Languages Design and Implementation. pp. 24–35. SIGPLAN Notices, Vol. 29, No 6, ACM, ACM, Orlando, Florida (Jun 1994)
26. Launchbury, J., Peyton Jones, S.: State in Haskell. Lisp and Symbolic Computation 8(4), 293–341 (1995)
27. Midtgaard, J.: Control-flow analysis of functional programs. ACM Computing Surveys 44(3), 10:1–10:33 (Jun 2012)
28. Mishra, P., Reddy, U.S.: Declaration-free type checking. In: Van Deusen, M.S., Galil, Z., Reid, B.K. (eds.) Proceedings of the Twelfth Annual ACM Symposium on Principles of Programming Languages. pp. 7–21. ACM, ACM, New Orleans, Louisiana (Jan 1985)
29. Mossin, C.: Exact flow analysis. Mathematical Structures in Computer Science 13(1), 125–156 (2003)
30. Mycroft, A.: Polymorphic type schemes and recursive definitions. In: Paul, M., Robinet, B. (eds.) Proceedingsof International Symposium on Programming. Lecture Notes in Computer Science, vol. 167, pp. 217–228. Springer-Verlag, Toulouse, France (Apr 1984)
31. Nielson, F., Nielson, H.R.: Interprocedural control flow analysis. In: Swierstra, S.D. (ed.) Proceedings of the Eighth European Symposium on Programming. Lecture Notes in Computer Science, vol. 1576, pp. 20–39. Springer-Verlag, Amsterdam, The Netherlands (Mar 1999)
32. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer-Verlag (1999)
33. Palsberg, J.: Type-based analysis and applications. In: Field, J., Snelting, G. (eds.) PASTE '01: Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering. pp. 20–27 (2001)
34. Peyton Jones, S.: Compiling Haskell by program transformation: A report from the trenches. In: Nielson, H.R. (ed.) Proceedings of the Sixth European Symposium on Programming. Lecture Notes in Computer Science, vol. 1058, pp. 18–44. Springer-Verlag, Linköping, Sweden (Apr 1996)
35. Pottier, F., Gauthier, N.: Polymorphic typed defunctionalization and concretization. Higher-Order and Symbolic Computation 19(1), 125–162 (2006), a preliminary version was presented at the Thirty-First Annual ACM Symposium on Principles of Programming Languages (POPL 2004)
36. Rehof, J., Fähndrich, M.: Type-based flow analysis: from polymorphic subtyping to CFL-reachability. In: Nielson, H.R. (ed.) Proceedings of the 28th Annual ACM Symposium on Principles of Programming Languages. pp. 54–66. London, United Kingdom (Jan 2001)
37. Reppy, J.: Type-sensitive control-flow analysis. In: Kennedy, A., Pottier, F. (eds.) ML'06: Proceedings of the ACM SIGPLAN 2006 workshop on ML. pp. 74–83 (Sep 2006)

38. Reynolds, J.: Towards a theory of type structure. In: Robinet, B. (ed.) Proceeding-sof Programming Symposium (Colloque sur la Programmation). Lecture Notes in Computer Science, vol. 19, pp. 408–425. Springer-Verlag, Paris, France (Apr 1974)
39. Reynolds, J.C.: Definitional interpreters for higher-order programming languages. In: Proceedings of 25th ACM National Conference. pp. 717–740. Boston, Massachusetts (1972), reprinted in Higher-Order and Symbolic Computation 11(4):363–397, 1998, with a foreword [40]
40. Reynolds, J.C.: Definitional interpreters revisited. Higher-Order and Symbolic Computation 11(4), 355–361 (1998)
41. Rossberg, A., Russo, C., Dreyer, D.: F-ing modules. In: Benton, N. (ed.) TLDI'10: Proceedings of the Fifth ACM SIGPLAN Workshop on Types in Language Design and Implementation. pp. 89–102 (Jan 2010)
42. Serrano, M.: Control flow analysis: a functional languages compilation paradigm. In: Proceedings of the 1995 ACM Symposium on Applied Computing. pp. 118–122. Nashville, Tennessee (Feb 1995)
43. Sestoft, P.: Replacing function parameters by global variables. In: Stoy, J.E. (ed.) Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture. pp. 39–53. London, England (Sep 1989)
44. Shivers, O.: Control-flow analysis in Scheme. In: Schwartz, M.D. (ed.) Proceedings of the ACM SIGPLAN 1988 Conference on Programming Languages Design and Implementation. pp. 164–174. Atlanta, Georgia (Jun 1988)
45. Smolka, G. (ed.): Proceedings of the Ninth European Symposium on Programming, Lecture Notes in Computer Science, vol. 1782. Springer-Verlag, Berlin, Germany (Mar 2000)
46. Sulzmann, M., Chakravarty, M.M.T., Peyton Jones, S., Donnelly, K.: System F with type equality coercions. In: Necula, G. (ed.) TLDI'07: Proceedings of the Third ACM SIGPLAN Workshop on Types in Language Design and Implementation. pp. 53–66 (Jan 2007)
47. Tarditi, D.: Design and implementation of code optimizations for a type-directed compiler for Standard ML. Ph.D. thesis, Carnegie Mellon University, Pittsburgh, Pensylvania (1997)
48. Tarditi, D., Morrisett, G., Cheng, P., Stone, C., Harper, R., Lee, P.: Til: a type-directed optimizing compiler for ml. In: Fischer, C. (ed.) Proceedings of the ACM SIGPLAN 1996 Conference on Programming Languages Design and Implementation. pp. 181–192. ACM, Philadelphia, Pennsylvania (May 1996)
49. Tolmach, A., Oliva, D.P.: From ML to Ada: Strongly-typed language interoperability via source translation. Journal of Functional Programming 8(4), 367–412 (1998)
50. Wells, J.B., Dimock, A., Muller, R., Turbak, F.: A calculus with polymorphic and polyvariant flow types. Journal of Functional Programming 12(3), 183–227 (2002)
51. Wright, A.K., Felleisen, M.: A syntactic approach to type soundness. Information and Computation 115(1), 38–94 (1994)