# An Efficient Type- and Control-Flow Analysis for System F

Connor Adsit
Rochester Institute of Technology
cda8519@rit.edu

Matthew Fluet
Rochester Institute of Technology
mtf@cs.rit.edu

## Abstract

At IFL'12, we presented a novel monovariant flow analysis for System F (with recursion) that yields both *type-flow* and *control-flow* information. [5] The type-flow information approximates the type expressions that may instantiate type variables and the control-flow information approximates the $\lambda$- and $\Lambda$-expressions that may be bound to variables. Furthermore, the two flows are mutually beneficial: control flow determines which $\Lambda$-expressions may be applied to which type expressions (and, hence, which type expressions may instantiate which type variables), while type flow filters the $\lambda$- and $\Lambda$-expressions that may be bound to variables (by rejecting expressions with static types that are incompatible with the static type of the variable under the type flow).

Using a specification-based formulation of the type- and control-flow analysis, we proved the analysis to be sound, decidable, and computable. Unfortunately, naïvely implementing the analysis using a standard least fixed-point iteration yields an $O(n^{13})$ algorithm.

In this work, we give an alternative flow-graph-based formulation of the type- and control-flow analysis. We prove that the flow-graph-based formulation induces solutions satisfying the specification-based formulation and, hence, that the flow-graph-based formulation of the analysis is sound. We give a direct algorithm implementing the flow-graph-based formulation and demonstrate that it is $O(n^4)$. By distinguishing the size $l$ of expressions in the program from the size $m$ of types in the program and performing an amortized complexity analysis, we further demonstrate that the algorithm is $O(l^3 + m^4)$.

***Categories and Subject Descriptors*** F.3.2 [*Semantics of Programming Languages*]: Program analysis

***General Terms*** Languages, Theory

***Keywords*** control-flow analysis, type-flow analysis, System F, quartic algorithm

## 1. Introduction

In previous work [5], we introduced a novel flow analysis for System F (with recursion). Our flow analysis is an extension of 0CFA [17, 21], the classic monovariant control-flow analysis that was formulated for the untyped lambda calculus. Like 0CFA, our flow analysis yields *control-flow* information via a global context-insensitive environment that maps expression variables to sets of $\lambda$-expressions that may be bound to the variable during evaluation. In addition, our flow analysis yields *type-flow* information via a global context-insensitive environment that maps type variables to sets of types that may instantiate the variable during evaluation. Finally, our flow analysis exploits the well-typedness of the program to improve the precision of the analysis.

As an example, consider the following program:

$id = \Lambda\alpha.\lambda x{:}\alpha.\ x$
$app = \Lambda\beta.\Lambda\gamma.\lambda f{:}\beta\to\gamma.\lambda z{:}\beta.$ `let` $g{:}\beta\to\gamma = id\ [\beta\to\gamma]\ f$ `in` $g\ z$

$h1 = \lambda a1{:}\mathsf{int}.\lambda a2{:}\mathsf{int}.\ a1{+}a2$
$h2 = \lambda b1{:}\mathsf{bool}.\lambda b2{:}\mathsf{int}.$ `if` $b1$ `then` $b2$+1 `else` $b2$
$h3 = \lambda c1{:}\mathsf{str}.\lambda c2{:}\mathsf{int}.\ \mathsf{len}(c1){+}c2$

$res1{:}\mathsf{int}\to\mathsf{int}\to\mathsf{int} = id\ [\mathsf{int}\to\mathsf{int}\to\mathsf{int}]\ h1$
$res2{:}\mathsf{bool}\to\mathsf{int}\to\mathsf{int} = id\ [\mathsf{bool}\to\mathsf{int}\to\mathsf{int}]\ h2$
$res3{:}\mathsf{int}\to\mathsf{int} = app\ [\mathsf{str}]\ [\mathsf{int}\to\mathsf{int}]\ h3$ `"zzz"`

The results of both 0CFA and our type- and control-flow analysis are given by the following table:

| | 0CFA | TCFA |
|---|---|---|
| $\alpha$ | – | $\mathsf{int}\to\mathsf{int}\to\mathsf{int}$ , $\mathsf{bool}\to\mathsf{int}\to\mathsf{int}$ , $\beta\to\gamma$ |
| $x$ | $\lambda a1$ , $\lambda b1$ , $\lambda c1$ | $\lambda a1$ , $\lambda b1$ , $\lambda c1$ |
| $res1$ | $\lambda a1$ , $\lambda b1$ , $\lambda c1$ | $\lambda a1$ |
| $res2$ | $\lambda a1$ , $\lambda b1$ , $\lambda c1$ | $\lambda b1$ |
| $\beta$ | – | $\mathsf{str}$ |
| $\gamma$ | – | $\mathsf{int}\to\mathsf{int}$ |
| $f$ | $\lambda c1$ | $\lambda c1$ |
| $g$ | $\lambda a1$ , $\lambda b1$ , $\lambda c1$ | $\lambda c1$ |
| $res3$ | $\lambda a2$ , $\lambda b2$ , $\lambda c2$ | $\lambda c2$ |

Note that 0CFA conflates all functions that flow through the $id$ function and, hence, concludes that each of each of $x$, $res1$, $res2$, and $g$ might be bound to $\{\lambda a1, \lambda b1, \lambda c1\}$ and that $res3$ might be bound to $\{\lambda a2, \lambda b2, \lambda c2\}$. However, type soundness ensures that $res1$ may only be bound to values of type $\mathsf{int}\to\mathsf{int}\to\mathsf{int}$ and therefore cannot be bound to $\lambda b1$ or $\lambda c1$ and, similarly, $res2$ cannot be bound to $\lambda a1$ or $\lambda c1$. More subtly, $g$ cannot be bound to $\lambda a1$ or $\lambda b1$, due to the static type of $g$ and the type-flow information about the types at which $\beta$ and $\gamma$ may be instantiated; this improvement in precision for $g$ leads to an improvement in precision for $res3$. Note that it is critical to filter by types during the analysis as one cannot obtain the TCFA results by post-processing the 0CFA results; in particular, both of $\lambda a2$ and $\lambda b2$ have types that are compatible with that of $res3$.

Of course, when judging the utility of a program analysis, one must take into account both the precision of the analysis and the cost of computing the analysis. Although our type- and control-flow analysis can be more precise than 0CFA, it would not be an attractive analysis if it were significantly more expensive to compute than 0CFA. Computing 0CFA via a naïve least fixed-point iteration is $O(n^5)$ [17], but many other algorithms for 0CFA have been shown to be $O(n^3)$ [2, 9, 17, 19], and recently improved to


*IFL '14*, October 01–03, 2014, Boston, MA, USA.

http://dx.doi.org/10.1145/2746325.2746327

$$
\begin{array}{llcccl}
\textit{Type variables} & \textit{TyVar} & \ni & \alpha, \beta, \ldots & & \\
\textit{Type indices} & \textit{TyIdx} & \ni & n & ::= & \mathtt{0} \mid \mathtt{1} \mid \cdots \\
\textit{Type binds} & \textit{TyBnd} & \ni & \tau & ::= & \alpha_a \to \alpha_b \mid \forall.\ \alpha_b \mid \#n \\
\\
\textit{Expression variables} & \textit{ExpVar} & \ni & x, y, z, f, g, \ldots & & \\
\textit{Expression binds (simple)} & \textit{ExpBnd}_{\mathsf{s}} & \ni & b_{\mathsf{s}} & ::= & \mu f{:}\alpha_f\,.\,\lambda z{:}\alpha_z\,.\,e_b \mid \mu f{:}\alpha_f\,.\,\Lambda\beta\,.\,e_b \\
\textit{Expression binds (complex)} & \textit{ExpBnd}_{\mathsf{c}} & \ni & b_{\mathsf{c}} & ::= & x_f\ x_a \mid x_f\ [\alpha_a] \\
\textit{Expression binds} & \textit{ExpBnd} & \ni & b & ::= & b_{\mathsf{s}} \mid b_{\mathsf{c}} \\
\\
\textit{Expressions} & \textit{Exp} & \ni & e & ::= & x \mid \mathtt{let}\ \alpha = \tau\ \mathtt{in}\ e \mid \mathtt{let}\ x{:}\alpha_x = b\ \mathtt{in}\ e \\
\textit{Programs} & \textit{Prog} & \ni & P & ::= & e
\end{array}
$$

**Figure 1.** Syntax of ANF System F

$$
\begin{array}{rcl}
\mathsf{ResOf}(\cdot) & :: & \textit{Exp} \to \textit{ExpVar} \\
\mathsf{ResOf}(x) & = & x \\
\mathsf{ResOf}(\mathtt{let}\ \alpha = \tau\ \mathtt{in}\ e) & = & \mathsf{ResOf}(e) \\
\mathsf{ResOf}(\mathtt{let}\ x{:}\alpha_x = b\ \mathtt{in}\ e) & = & \mathsf{ResOf}(e)
\end{array}
\qquad
\begin{array}{rcl}
\mathsf{TyOf}(\cdot) & :: & \textit{ExpBnd}_{\mathsf{s}} \to \textit{TyVar} \\
\mathsf{TyOf}(\mu f{:}\alpha_f\,.\,\lambda z{:}\alpha_z\,.\,e_b) & = & \alpha_f \\
\mathsf{TyOf}(\mu f{:}\alpha_f\,.\,\Lambda\beta\,.\,e_b) & = & \alpha_f
\end{array}
$$

$$\boxed{e \preceq_{\mathit{Exp}} P}$$

$$
\frac{}{P \preceq_{\mathit{Exp}} P}
\qquad
\frac{\mathtt{let}\ \alpha = \tau\ \mathtt{in}\ e \preceq_{\mathit{Exp}} P}{e \preceq_{\mathit{Exp}} P}
\qquad
\frac{\mathtt{let}\ x{:}\alpha_x = b\ \mathtt{in}\ e \preceq_{\mathit{Exp}} P}{e \preceq_{\mathit{Exp}} P}
$$

$$
\frac{\mu f{:}\alpha_f\,.\,\lambda z{:}\alpha_z\,.\,e_b \preceq_{\mathit{ExpBnd}} P}{e_b \preceq_{\mathit{Exp}} P}
\qquad
\frac{\mu f{:}\alpha_f\,.\,\Lambda\beta\,.\,e_b \preceq_{\mathit{ExpBnd}} P}{e_b \preceq_{\mathit{Exp}} P}
$$

$$\boxed{b \preceq_{\mathit{ExpBnd}} P} \qquad \boxed{\tau \preceq_{\mathit{TyBnd}} P}$$

$$
\frac{\mathtt{let}\ x{:}\alpha_x = b\ \mathtt{in}\ e \preceq_{\mathit{Exp}} P}{b \preceq_{\mathit{ExpBnd}} P}
\qquad
\frac{\mathtt{let}\ \alpha = \tau\ \mathtt{in}\ e \preceq_{\mathit{Exp}} P}{\tau \preceq_{\mathit{TyBnd}} P}
$$

$$\boxed{x \preceq_{\mathit{ExpVar}} P}$$

$$
\frac{\mathtt{let}\ x{:}\alpha_x = b\ \mathtt{in}\ e \preceq_{\mathit{Exp}} P}{x \preceq_{\mathit{ExpVar}} P}
\qquad
\frac{\mu f{:}\alpha_f\,.\,\lambda z{:}\alpha_z\,.\,e_b \preceq_{\mathit{ExpBnd}} P}{f \preceq_{\mathit{ExpVar}} P}
\qquad
\frac{\mu f{:}\alpha_f\,.\,\lambda z{:}\alpha_z\,.\,e_b \preceq_{\mathit{ExpBnd}} P}{z \preceq_{\mathit{ExpVar}} P}
\qquad
\frac{\mu f{:}\alpha_f\,.\,\Lambda\beta\,.\,e_b \preceq_{\mathit{ExpBnd}} P}{f \preceq_{\mathit{ExpVar}} P}
$$

$$\boxed{\alpha \preceq_{\mathit{TyVar}} P}$$

$$
\frac{\mathtt{let}\ \alpha = \tau\ \mathtt{in}\ e \preceq_{\mathit{Exp}} P}{\alpha \preceq_{\mathit{TyVar}} P}
\qquad
\frac{\mu f{:}\alpha_f\,.\,\Lambda\beta\,.\,e_b \preceq_{\mathit{ExpBnd}} P}{\beta \preceq_{\mathit{TyVar}} P}
$$

**Figure 2.** Auxiliary Functions and Relations on Syntax

$O(n^3/\log n)$ [12] using fast sets [3, 20]. While our previous work demonstrated that our type- and control-flow analysis is sound and computable, the algorithm using a naïve least fixed-point is, disappointingly, $O(n^{13})$.

In this work, we greatly improve the situation. After recalling our previous specification-based formulation of the type- and control-flow analysis, we introduce an alternative flow-graph-based formulation. We prove that the flow-graph-based formulation induces solutions satisfying the specification-based formulation and, hence, that the flow-graph-based formulation of the analysis is sound. We give a direct algorithm implementing the flow-graph-based formulation and demonstrate that it is $O(n^4)$. By distinguishing the size $l$ of expressions in the program from the size $m$ of types in the program and performing an amortized complexity analysis, we further demonstrate that the algorithm is $O(l^3 + m^4)$.

## 2. Language

***Syntax*** Figure 1 gives the syntax of our language, which is a variant of System F, extended with recursive functions and type abstractions and presented in administrative normal form (ANF). Of note, we use an ANF representation for both expressions and types, similar to the $\lambda_{\mathrm{gc}}^{\to\forall}$ language [14].

Programs are (closed, well-typed) expressions. An expression is either an expression variable, a `let`-binding of a type bind to a type variable, or a `let`-binding of an expression bind to an expression variable. A simple expression bind is either a recursive functions or a recursive type abstractions and a complex expression bind is either a function application or a type application, with constituents restricted to variables. Note that every bound expression variable is annotated with a type variable.

Intuitively, a type is either a function type, a universal type, a type (de Bruijn) index ("bound" by an enclosing $\forall$ type), or a type variable (bound by an enclosing $\Lambda$ expression bind). Using de Bruijn indices for universal types ensures that type equality corresponds to syntactic identity (without introducing $\alpha$-equivalence classes), simplifying type compatibility in the flow analysis. To further simplify type compatibility in our efficient flow-analysis algorithm, we use an ANF representation for types, with constituents of function and universal types restricted to type variables. Thus, expressions include a `let`-binding of a type bind to a type variable and a type bind is either a function type, a universal type, or a type (de Bruijn) index. Note that this ANF representation models and promotes sharing of types, leading to a smaller contribution from types to overall program size than in a typical direct-style representation.

Figure 2 defines a number of auxiliary functions and relations. The function $\mathsf{ResOf}(\cdot)$ on expressions extracts the expression variable that yields the expression's value and the function $\mathsf{TyOf}(\cdot)$ on simple expression binds extracts the type variable annotating the $\mu$-bound variable; in a well-typed program, this type variable will denote the type of the recursive function or recursive type abstraction. The various $\cdot \preceq P$ judgments relate a program $P$ to its constituent expressions, expression binds, type binds, bound expression variables, and bound type variables.

***Operational Semantics and Type System*** An operational semantics for our language can be given in the style of the ANF environment- and continuation-based $C_aEK$ abstract machine [4], where the environment component of the abstract machine consists of a value environment mapping expression variables to *value closures* (pairs of simple expression binds (i.e., functions or type abstractions) and an environment, which provides the free expression variables and type variables of the simple expression bind) and a type environment mapping type variables to *type closures* (pairs of type binds and an environment, which provides the free type variables of the type bind). The use of type closures makes our abstract machine similar in some ways to the $\lambda_{\mathrm{gc}}^{\rightarrow\forall}$ abstract machine [14]. For more details, please see our previous work [5, 6].

A type system for our language can be given by extending the standard type system for System F with support for type definitions [14, 22]. Essentially, the type variable context must distinguish between abstract declarations (introduced by a $\Lambda$ expression bind) and transparent declarations (introduced by a `let`-binding of a type bind to a type variable) and occurrences of type variables at bound expression variables and in type applications must be expanded according to the transparent declarations of the typing context and checked for well-formedness with respect to type (de Bruijn) indices.

We purposefully omit the details of the operational semantics and type system in this work, since they are not directly related to the contributions of this paper. In particular, the specification-based formulation of the flow analysis given in Section 3 and the flow-graph-based formulation given in Section 4 are well-defined for all input programs, including ill-typed programs. Similarly, the proof that the flow-graph-based formulation induces solutions satisfying the specification-based formulation holds for all input programs and does not depend upon the judgments or rules of the type system. Finally, the algorithm of Section 5 correctly implements the flow-graph-based formulation for all input programs. The operational semantics is only required to characterize soundness and the type system is only required to prove that the specification-based formulation of the flow analysis is sound for well-typed programs, which we established in our previous work [5, 6].

## 3. Specification-Based Formulation of TCFA

Figure 3 recalls the specification-based formulation of the type- and control-flow analysis from our previous work [5]. As a specification-based formulation [7, 15–18], it is presented as a judgment that asserts the various constraints that an acceptable analysis result must satisfy.

For our type- and control-flow analysis, a result is a pair of abstract environments. An abstract type environment $\hat{\phi}$ is a map from type variables to sets of type binds and an abstract value environment $\hat{\rho}$ is a map from expression variables to sets of simple expression binds. Intuitively, acceptable abstract type and value environments must (conservatively) describe every type and value environment that arises during the evaluation of the program.

The judgment $\hat{\phi}; \hat{\rho} \models_{\mathsf{S}} e$ asserts that an abstract type environment $\hat{\phi}$ and an abstract value environment $\hat{\rho}$ are an acceptable type- and control-flow analysis result for the expression $e$. The judgement is syntax directed and the constraints asserted by the rules are standard for a monovariant control-flow analysis, except that each assertion of the form $b_{\mathsf{s}} \in \hat{\rho}(x)$, which asserts that a simple expression bind flows to an expression variable, is guarded by an assertion of the form $\hat{\phi} \models_{\mathsf{S}} \mathsf{TyOf}(b_{\mathsf{s}}) \approxeq \alpha_x$, which asserts that (the type variable denoting) the type of the simple expression bind is *compatible* with (the type variable denoting) the type of the expression variable.

The judgement $\hat{\phi} \models_{\mathsf{S}} \alpha_1 \approxeq \alpha_2$ asserts that the type variables $\alpha_1$ and $\alpha_2$ are compatible under the abstract type environment $\hat{\phi}$ by asserting that $\alpha_1$ and $\alpha_2$ expand to a common closed type $\theta$. The judgments $\hat{\phi} \models_{\mathsf{S}} \tau \Rrightarrow \theta$ and $\hat{\phi} \models_{\mathsf{S}} \alpha \Rrightarrow \theta$ handle expanding type binds and type variables to closed types. Note that when expanding a type variable, the rule is free to choose any type bind from the abstract type environment's entry for the type variable; when used in the context of the compatibility judgment, this rule must "guess" a satisfying type bind from among those in $\hat{\phi}(\alpha)$.

Now consider the rules for the $\hat{\phi}; \hat{\rho} \models_{\mathsf{S}} e$ judgment. If the expression is a `let` $\alpha$ `=` $\tau$ expression, then the type bind $\tau$ flows to $\alpha$ and we require that $\tau$ occur in $\hat{\phi}(\alpha)$. If the expression is a `let` $x$`:`$\alpha_x$ `=` $b_{\mathsf{s}}$ expression and the type of the simple expression bind $b_{\mathsf{s}}$ is compatible with the type of $x$, then simple expression bind $b_{\mathsf{s}}$ flows to $x$ and we require $b_{\mathsf{s}} \in \hat{\rho}(x)$. Similarly, if the type of the simple expression bind $b_{\mathsf{s}}$ is compatible with the type of the $\mu$-bound expression variable $f$, then we require $b_{\mathsf{s}} \in \hat{\rho}(f)$.

If the expression is a `let` $x$`:`$\alpha_x$ `=` $x_f\ x_a$ expression, then we iterate through all the recursive functions in $\hat{\rho}(x_f)$. For each simple expression bind $b_{\mathsf{s}}$ in $\hat{\rho}(x_a)$, representing a potential actual value argument at this function application, if the type of $b_{\mathsf{s}}$ is compatible with the type of the formal parameter $z$, then we require $b_{\mathsf{s}} \in \hat{\rho}(z)$. Similarly, for each simple expression bind $b_{\mathsf{s}}$ in $\hat{\rho}(\mathsf{ResOf}(e_b))$, representing a potential result at this function application, if the type of $b_{\mathsf{s}}$ is compatible with the type of the receiving `let`-bound expression variable $x$, then we require $b_{\mathsf{s}} \in \hat{\rho}(x)$.

Finally, if the expression is a `let` $x$`:`$\alpha_x$ `=` $x_f$ `[`$\alpha_a$`]` expression, then we iterate through all the recursive type abstractions in $\hat{\rho}(x_f)$. For each type bind $\tau$ in $\hat{\phi}(\alpha_a)$, representing a potential actual type argument at this type application, the type bind $\tau$ flows to the formal parameter $\beta$ and we require $\tau \in \hat{\phi}(\beta)$. As for the function application rule, for each simple expression bind $b_{\mathsf{s}}$ in $\hat{\rho}(\mathsf{ResOf}(e_b))$, representing a potential result at this type application, if the type of $b_{\mathsf{s}}$ is compatible with the type of the receiving `let`-bound expression variable $x$, then we require $b_{\mathsf{s}} \in \hat{\rho}(x)$.

$$
\begin{array}{lrcll}
\textit{Closed Types} & \textit{TyClsd} & \ni & \theta & ::= \theta_a \to \theta_b \mid \forall\,.\,\theta_b \mid \#n \\
\textit{Abstract type environments} & \textit{ATyEnv} = \textit{TyVar} \to \mathcal{P}(\textit{TyBnd}) & \ni & \hat{\phi} & ::= \{\alpha \mapsto \{\tau, \ldots\}, \ldots\} \\
\textit{Abstract value environments} & \textit{AValEnv} = \textit{ExpVar} \to \mathcal{P}(\textit{ExpBnd}_{\mathsf{s}}) & \ni & \hat{\rho} & ::= \{x \mapsto \{b_{\mathsf{s}}, \ldots\}, \ldots\}
\end{array}
$$

$\boxed{\hat{\phi} \vDash_{\mathsf{S}} \tau \Rrightarrow \theta}$

$$
\frac{\hat{\phi} \vDash_{\mathsf{S}} \alpha_a \Rrightarrow \theta_a \qquad \hat{\phi} \vDash_{\mathsf{S}} \alpha_b \Rrightarrow \theta_b}{\hat{\phi} \vDash_{\mathsf{S}} \alpha_a \to \alpha_b \Rrightarrow \theta_a \to \theta_b}
\qquad
\frac{\hat{\phi} \vDash_{\mathsf{S}} \alpha_b \Rrightarrow \theta_b}{\hat{\phi} \vDash_{\mathsf{S}} \forall\,.\,\alpha_b \Rrightarrow \forall\,.\,\theta_b}
\qquad
\frac{}{\hat{\phi} \vDash_{\mathsf{S}} \#n \Rrightarrow \#n}
$$

$\boxed{\hat{\phi} \vDash_{\mathsf{S}} \alpha \Rrightarrow \theta}$

$$
\frac{\tau \in \hat{\phi}(\alpha) \qquad \hat{\phi} \vDash_{\mathsf{S}} \tau \Rrightarrow \theta}{\hat{\phi} \vDash_{\mathsf{S}} \alpha \Rrightarrow \theta}
$$

$\boxed{\hat{\phi} \vDash_{\mathsf{S}} \alpha_1 \approxeq \alpha_2}$

$$
\frac{\hat{\phi} \vDash_{\mathsf{S}} \alpha_1 \Rrightarrow \theta \qquad \hat{\phi} \vDash_{\mathsf{S}} \alpha_2 \Rrightarrow \theta}{\hat{\phi} \vDash_{\mathsf{S}} \alpha_1 \approxeq \alpha_2}
$$

$\boxed{\hat{\phi}; \hat{\rho} \vDash_{\mathsf{S}} e}$

$$
\frac{}{\hat{\phi}; \hat{\rho} \vDash_{\mathsf{S}} x}
\qquad
\frac{\tau \in \hat{\phi}(\alpha) \qquad \hat{\phi}; \hat{\rho} \vDash_{\mathsf{S}} e}{\hat{\phi}; \hat{\rho} \vDash_{\mathsf{S}} \texttt{let}\ \alpha = \tau\ \texttt{in}\ e}
$$

$$
\frac{b_{\mathsf{s}} = \mu f{:}\alpha_f\,.\,\lambda z{:}\alpha_z\,.\,e_b \qquad \hat{\phi} \vDash_{\mathsf{S}} \mathsf{TyOf}(b_{\mathsf{s}}) \approxeq \alpha_x \Rightarrow b_{\mathsf{s}} \in \hat{\rho}(x) \qquad \hat{\phi} \vDash_{\mathsf{S}} \mathsf{TyOf}(b_{\mathsf{s}}) \approxeq \alpha_f \Rightarrow b_{\mathsf{s}} \in \hat{\rho}(f) \qquad \hat{\phi}; \hat{\rho} \vDash_{\mathsf{S}} e_b \qquad \hat{\phi}; \hat{\rho} \vDash_{\mathsf{S}} e}{\hat{\phi}; \hat{\rho} \vDash_{\mathsf{S}} \texttt{let}\ x{:}\alpha_x = \mu f{:}\alpha_f\,.\,\lambda z{:}\alpha_z\,.\,e_b\ \texttt{in}\ e}
$$

$$
\frac{b_{\mathsf{s}} = \mu f{:}\alpha_f\,.\,\Lambda\beta\,.\,e_b \qquad \hat{\phi} \vDash_{\mathsf{S}} \mathsf{TyOf}(b_{\mathsf{s}}) \approxeq \alpha_x \Rightarrow b_{\mathsf{s}} \in \hat{\rho}(x) \qquad \hat{\phi} \vDash_{\mathsf{S}} \mathsf{TyOf}(b_{\mathsf{s}}) \approxeq \alpha_f \Rightarrow b_{\mathsf{s}} \in \hat{\rho}(f) \qquad \hat{\phi}; \hat{\rho} \vDash_{\mathsf{S}} e_b \qquad \hat{\phi}; \hat{\rho} \vDash_{\mathsf{S}} e}{\hat{\phi}; \hat{\rho} \vDash_{\mathsf{S}} \texttt{let}\ x{:}\alpha_x = \mu f{:}\alpha_f\,.\,\Lambda\beta\,.\,e_b\ \texttt{in}\ e}
$$

$$
\frac{\forall \mu f{:}\alpha_f\,.\,\lambda z{:}\alpha_z\,.\,e_b \in \hat{\rho}(x_f)\,.\,\left(\begin{array}{l}\forall b_{\mathsf{s}} \in \hat{\rho}(x_a)\,.\,\hat{\phi} \vDash_{\mathsf{S}} \mathsf{TyOf}(b_{\mathsf{s}}) \approxeq \alpha_z \Rightarrow b_{\mathsf{s}} \in \hat{\rho}(z) \wedge \\ \forall b_{\mathsf{s}} \in \hat{\rho}(\mathsf{ResOf}(e_b))\,.\,\hat{\phi} \vDash_{\mathsf{S}} \mathsf{TyOf}(b_{\mathsf{s}}) \approxeq \alpha_x \Rightarrow b_{\mathsf{s}} \in \hat{\rho}(x)\end{array}\right) \qquad \hat{\phi}; \hat{\rho} \vDash_{\mathsf{S}} e}{\hat{\phi}; \hat{\rho} \vDash_{\mathsf{S}} \texttt{let}\ x{:}\alpha_x = x_f\ x_a\ \texttt{in}\ e}
$$

$$
\frac{\forall \mu f{:}\alpha_f\,.\,\Lambda\beta\,.\,e_b \in \hat{\rho}(x_f)\,.\,\left(\begin{array}{l}\forall \tau \in \hat{\phi}(\alpha_a)\,.\,\tau \in \hat{\phi}(\beta) \wedge \\ \forall b_{\mathsf{s}} \in \hat{\rho}(\mathsf{ResOf}(e_b))\,.\,\hat{\phi} \vDash_{\mathsf{S}} \mathsf{TyOf}(b_{\mathsf{s}}) \approxeq \alpha_x \Rightarrow b_{\mathsf{s}} \in \hat{\rho}(x)\end{array}\right) \qquad \hat{\phi}; \hat{\rho} \vDash_{\mathsf{S}} e}{\hat{\phi}; \hat{\rho} \vDash_{\mathsf{S}} \texttt{let}\ x{:}\alpha_x = x_f\ [\alpha_a]\ \texttt{in}\ e}
$$

**Figure 3.** Specification-Based Formulation of TCFA

***Soundness, Decidability, and Computability*** Our previous work [5] showed that the specification-based formulation of the type- and control-flow analysis is sound with respect to the operational semantics, that the acceptability of given (finite) abstract type and value environments with respect to a program is decidable, and that the minimum acceptable abstract type and value environments for a program are computable in polynomial time. We briefly recall the essence of these arguments.

Soundness of the specification-based formulation of the type- and control-flow analysis asserts that any acceptable pair of abstract environments for a well-typed program approximates the run-time behavior of the program. In particular, the abstract type and value environments approximate every concrete type and value environment that arises during execution of the program. Flow soundness relies crucially on the well-typedness of the program. Soundness of the type system guarantees that, at run time, an expression variable will only be bound to a well-typed closed value of a closed type and that the expression variable's type annotation must be interpreted as that closed type. Hence, if there is no closed type at which both the static type of the expression variable and the static type of the value might be instantiated, then that variable will never be bound to that value at run time. The critical component of the proof is that the type compatibility judgment $\hat{\phi}; \hat{\rho} \vDash_{\mathsf{S}} \alpha_1 \approxeq \alpha_2$ is derivable whenever there is a common closed type at which both $\alpha_1$ and $\alpha_2$ are instantiated.

Although there are an infinite number of pairs of abstract type and value environments that are acceptable for a given program, we are primarily interested in more precise pairs over less precise pairs. For a given program, we can limit our attention to the "finite" abstract type and value environments that map the type variables that occur in the program to sets of type binds that appear in the program (and map all type variables that do not occur in the program to the empty set) and map the expression variables that occur in the program to sets of simple expression binds that appear in the program (and map all expression variables that do not occur in the program to the empty set).

The decidability of the acceptability judgment $\hat{\phi}; \hat{\rho} \models_{\mathsf{S}} e$ relies upon the decidability of the type compatibility judgment $\hat{\phi} \models_{\mathsf{S}} \alpha_1 \approx \alpha_2$. Due to "recursion" in the abstract type environment, whereby a type variable may be mapped (directly or indirectly) to a set of type binds in which the type variable itself occurs, we cannot simply enumerate the (potentially infinite sets of) closed types $\theta_1$ and $\theta_2$ such that $\hat{\phi} \models_{\mathsf{S}} \alpha_1 \Rrightarrow \theta_1$ and $\hat{\phi} \models_{\mathsf{S}} \alpha_2 \Rrightarrow \theta_2$ in order to decide whether or not the judgment $\hat{\phi} \models_{\mathsf{S}} \alpha_1 \approx \alpha_2$ is derivable. To address this issue, we take inspiration from the theory and implementation of regular-tree grammars [1, 8, 13]. By interpreting an abstract type environment as (the productions for) a regular-tree grammar, a derivation of the judgment $\hat{\phi} \models_{\mathsf{S}} \alpha \Rrightarrow \theta$ is exactly a parse tree witnessing the derivation of the ground tree $\theta$ from the starting non-terminal $\alpha$ in the regular-tree grammar $\hat{\phi}$ and the judgment $\hat{\phi} \models_{\mathsf{S}} \alpha_1 \approx \alpha_2$ is derivable iff the languages generated by taking $\alpha_1$ and $\alpha_2$, respectively, as the starting non-terminal in the regular-tree grammar $\hat{\phi}$ have a non-empty intersection. Since regular-tree grammars are closed under intersection and the emptiness of a regular-tree grammar is decidable [1, 8, 13], the type compatibility judgment $\hat{\phi} \models_{\mathsf{S}} \alpha_1 \approx \alpha_2$ is decidable.

Finally, the minimum acceptable pair of abstract type and value environments for a given program is computable via a standard least-fixed point iteration. We interpret the acceptability judgment $\hat{\phi}; \hat{\rho} \models_{\mathsf{S}} e$ as defining a monotone function from pairs of abstract environments to pairs of abstract environments; the "output" abstract environments are formed from the "input" abstract environments joined with all discovered violations.

We conclude with a crude upper-bound on computing the minimum acceptable pair of abstract type and value environments for a given program, of size $n$, via a standard least-fixed point computation. We represent $\hat{\phi}$ and $\hat{\rho}$ as two-dimensional arrays (indexed by $\alpha/\tau$ and $x/b_{\mathsf{s}}$, respectively), requiring $O(n^2)$ space.[1] Thus, the two abstract environments are lattices of height $O(n^2)$. Each (naïve) iteration of the monotone function is syntax directed ($O(n)$) and dominated by the function-application bind, which loops over all of the elements of $\hat{\rho}(x_f)$ ($O(n)$), loops over all of the elements of $\hat{\rho}(x_a)$ ($O(n)$) and $\hat{\rho}(\mathsf{ResOf}(e_b))$ ($O(n)$), and computes type compatibility via a regular-tree grammar intersection ($O((n^2)^2)$, because the output regular-tree grammar is, worst-case, quadratic space with respect to the input regular-tree grammar) and emptiness test ($O(((n^2)^2)^2)$, because the emptiness query is quadratic time with respect to the input regular-tree grammar). Hence, our analysis is computable in polynomial time: $O(n^{13}) = (O(n^2) + O(n^2)) \times (O(n) \times O(n) \times (O(n) + O(n)) \times (O(n^4) + O(n^8)))$.[2]

## 4. Flow-Graph-Based Formulation of TCFA

Figure 4 gives an alternative flow-graph-based formulation of the type- and control-flow analysis. As a flow-graph-based formulation [10, 11], it is presented as a collection of judgements that define a directed graph, with nodes corresponding to program constituents and edges corresponding to the flow of abstract values from one node to another.

For our type- and control-flow analysis, the primary judgments are $P \models_{\mathsf{G}} b_{\mathsf{s}} \rightarrowtail x$ and $P \models_{\mathsf{G}} \tau \rightarrowtail \alpha$; the former corresponds to an edge representing the flow of the simple expression bind $b_{\mathsf{s}}$ to the expression variable $x$ and the latter corresponds to an edge representing the flow the type bind $\tau$ to the type variable $\alpha$. These two judgements are closely related to the abstract environments of the specification-based formulation, in a sense to be made precise below. The judgment $P \models_{\mathsf{G}} b_{\mathsf{s}} \rightarrowtail^{?} x : \alpha_x$ corresponds to a conditional edge that represents the flow of the simple expression bind $b_{\mathsf{s}}$ to the expression variable $x$ with type annotation $\alpha_x$ when $\mathsf{TyOf}(b_{\mathsf{s}})$ and $\alpha_x$ are compatible. The judgements $P \models_{\mathsf{G}} x \rightarrowtail y : \alpha_y$ and $P \models_{\mathsf{G}} \alpha \rightarrowtail \beta$ correspond to edges representing the flow of simple expression binds through expression variable $x$ to expression variable $y$ with type annotation $\alpha_y$ and the flow of type binds through type variable $\alpha$ to type variable $\beta$. Note that these judgements are not syntax directed, but the rules make use of the $e \preceq_{Exp} P$ judgment to limit the analysis to a given program $P$.

The judgment $P \models_{\mathsf{G}} \alpha_1 \approx \alpha_2$ asserts that the type variables $\alpha_1$ and $\alpha_2$ are compatible by asserting the existence of type binds $\tau_1$ and $\tau_2$ flowing to $\alpha_1$ and $\alpha_2$, respectively, such that $\tau_1$ and $\tau_2$ are compatible. The judgment $P \models_{\mathsf{G}} \tau_1 \approx \tau_2$ asserts that the type binds $\tau_1$ and $\tau_2$ are compatible. Type indices are compatible if they are equal, while function and universal types are compatible if their corresponding constituent type variables are compatible.

Now consider the rules for the various $P \models_{\mathsf{G}} \cdot \rightarrowtail \cdot$ judgements. The TyVarCompatExpBind$_{\mathsf{s}}$ rule realizes a conditional flow as a confirmed flow when the type of the simple expression bind is compatible with the type of the expression variable. The LetTyBind rule captures the flow of $\tau$ to $\alpha$ due to a `let` $\alpha$ = $\tau$ expression in the program. Similarly, the LetExpBnd$_{\mathsf{s}}$, $\mu\lambda$ExpBnd$_{\mathsf{s}}$, and $\mu\Lambda$ExpBnd$_{\mathsf{s}}$ rules capture the conditional flows of $b_{\mathsf{s}}$ to $x$ and to the $\mu$-bound expression variable $f$ due to a `let` $x$:$\alpha_x$ = $b_{\mathsf{s}}$ expression in the program. The TransTyBnd and TransExpBnd$_{\mathsf{s}}$ rules capture transitive flows due to type variable to type variable edges and expression variable to expression variable edges.

The ExpAppArg and ExpAppRes rules capture the flows due to a `let` $x_r$:$\alpha_r$ = $x$ $x_a$ expression in the program for each recursive function that flows to $x$: the flow of simple expression binds through the actual argument $x_a$ to the formal parameter $z$ and through the result variable $\mathsf{ResOf}(e_b)$ to the receiving `let`-bound expression variable $x$.

Similarly, the TyAppArg and TyAppRes rules capture the flows due to a `let` $x_r$:$\alpha_r$ = $x$ [$\alpha_a$] expression in the program for each recursive type abstraction that flows to $x$: the flow of type binds through the actual argument $\alpha_a$ to the formal parameter $\beta$ and the flow of simple expression binds through the result variable $\mathsf{ResOf}(e_b)$ to the receiving `let`-bound expression variable $x$.

***Soundness*** In order to show that the flow-graph-based formulation of the type- and control-flow analysis is sound with respect to the operational semantics, we relate the flow-graph-based formulation to the specification-based formulation. In particular, we have that the abstract type and value environments *induced* by the flow-graph-based formulation for a given program are acceptable for that program according to the specification-based formulation:

**Theorem 1** *For all programs $P$, abstract type environments $\hat{\phi}$, and abstract value environments $\hat{\rho}$, if*
- $\forall \alpha, \tau \,.\, \tau \in \hat{\phi}(\alpha) \;\Leftrightarrow\; P \models_{\mathsf{G}} \tau \rightarrowtail \alpha$ *and*
- $\forall x, b_{\mathsf{s}} \,.\, b_{\mathsf{s}} \in \hat{\rho}(x) \;\Leftrightarrow\; P \models_{\mathsf{G}} b_{\mathsf{s}} \rightarrowtail x,$

*then $\hat{\phi}; \hat{\rho} \models_{\mathsf{S}} P$.*

[1] See Sections 5.2 for more discussion of assumptions about the representation of the input program and data structures and operations.

[2] Our previous work [5] reported an erroneous upper-bound of $O(n^9)$, due to incorrectly using $n$ as the size of the regular-tree grammer for type compatibility (rather than $n^2$, the size of the abstract type environment).

$\boxed{P \models_{\mathsf{G}} \tau_1 \approxeq \tau_2}$

$$\textsc{TyCompatArrow}\quad \frac{P \models_{\mathsf{G}} \alpha_{a1} \approxeq \alpha_{a2} \qquad P \models_{\mathsf{G}} \alpha_{b1} \approxeq \alpha_{b2}}{P \models_{\mathsf{G}} \alpha_{a1} \to \alpha_{b1} \approxeq \alpha_{a2} \to \alpha_{b2}}$$

$$\textsc{TyCompatForAll}\quad \frac{P \models_{\mathsf{G}} \alpha_{b1} \approxeq \alpha_{b2}}{P \models_{\mathsf{G}} \forall.\ \alpha_{b1} \approxeq \forall.\ \alpha_{b2}}$$

$$\textsc{TyCompatTyIdx}\quad \frac{}{P \models_{\mathsf{G}} \texttt{\#}n \approxeq \texttt{\#}n}$$

$\boxed{P \models_{\mathsf{G}} \alpha_1 \approxeq \alpha_2}$

$$\textsc{TyVarCompat}\quad \frac{P \models_{\mathsf{G}} \tau_1 \rightarrowtail \alpha_1 \qquad P \models_{\mathsf{G}} \tau_2 \rightarrowtail \alpha_2 \qquad P \models_{\mathsf{G}} \tau_1 \approxeq \tau_2}{P \models_{\mathsf{G}} \alpha_1 \approxeq \alpha_2}$$

$\boxed{P \models_{\mathsf{G}} \tau \rightarrowtail \alpha}$

$$\textsc{LetTyBnd}\quad \frac{\texttt{let}\ \alpha = \tau\ \texttt{in}\ e \preceq_{Exp} P}{P \models_{\mathsf{G}} \tau \rightarrowtail \alpha}$$

$$\textsc{TransTyBnd}\quad \frac{P \models_{\mathsf{G}} \tau \rightarrowtail \alpha \qquad P \models_{\mathsf{G}} \alpha \rightarrowtail \beta}{P \models_{\mathsf{G}} \tau \rightarrowtail \beta}$$

$\boxed{P \models_{\mathsf{G}} \alpha \rightarrowtail \beta}$

$$\textsc{TyAppArg}\quad \frac{P \models_{\mathsf{G}} \mu f{:}\alpha_f\,.\,\Lambda\beta\,.\,e_b \rightarrowtail x \qquad \texttt{let}\ x_r{:}\alpha_r = x\ [\alpha_a]\ \texttt{in}\ e \preceq_{Exp} P}{P \models_{\mathsf{G}} \alpha_a \rightarrowtail \beta}$$

$\boxed{P \models_{\mathsf{G}} b_{\mathsf{s}} \rightarrowtail x}$

$$\textsc{TyVarCompatExpBnd}_{\mathsf{s}}\quad \frac{P \models_{\mathsf{G}} b_{\mathsf{s}} \rightarrowtail^{?} x : \alpha_x \qquad P \vdash \mathsf{TyOf}(b_{\mathsf{s}}) \approxeq \alpha_x}{P \models_{\mathsf{G}} b_{\mathsf{s}} \rightarrowtail x}$$

$\boxed{P \models_{\mathsf{G}} b_{\mathsf{s}} \rightarrowtail^{?} x : \alpha_x}$

$$\textsc{LetExpBnd}_{\mathsf{s}}\quad \frac{\texttt{let}\ x{:}\alpha_x = b_{\mathsf{s}}\ \texttt{in}\ e \preceq_{Exp} P}{P \models_{\mathsf{G}} b_{\mathsf{s}} \rightarrowtail^{?} x : \alpha_x}$$

$$\textsc{TransExpBnd}_{\mathsf{s}}\quad \frac{P \models_{\mathsf{G}} b_{\mathsf{s}} \rightarrowtail x \qquad P \models_{\mathsf{G}} x \rightarrowtail y : \alpha_y}{P \models_{\mathsf{G}} b_{\mathsf{s}} \rightarrowtail^{?} y : \alpha_y}$$

$$\mu\lambda\textsc{ExpBnd}_{\mathsf{s}}\quad \frac{\texttt{let}\ x{:}\alpha_x = \mu f{:}\alpha_f\,.\,\lambda z{:}\alpha_z\,.\,e_b\ \texttt{in}\ e \preceq_{Exp} P}{P \models_{\mathsf{G}} \mu f{:}\alpha_f\,.\,\lambda z{:}\alpha_z\,.\,e_b \rightarrowtail^{?} f : \alpha_f}$$

$$\mu\Lambda\textsc{ExpBnd}_{\mathsf{s}}\quad \frac{\texttt{let}\ x{:}\alpha_x = \mu f{:}\alpha_f\,.\,\Lambda\beta\,.\,e_b\ \texttt{in}\ e \preceq_{Exp} P}{P \models_{\mathsf{G}} \mu f{:}\alpha_f\,.\,\Lambda\beta\,.\,e_b \rightarrowtail^{?} f : \alpha_f}$$

$\boxed{P \models_{\mathsf{G}} x \rightarrowtail y : \alpha_y}$

$$\textsc{ExpAppArg}\quad \frac{P \models_{\mathsf{G}} \mu f{:}\alpha_f\,.\,\lambda z{:}\alpha_z\,.\,e_b \rightarrowtail x \qquad \texttt{let}\ x_r{:}\alpha_r = x\ x_a\ \texttt{in}\ e \preceq_{Exp} P}{P \models_{\mathsf{G}} x_a \rightarrowtail z : \alpha_z}$$

$$\textsc{ExpAppRes}\quad \frac{P \models_{\mathsf{G}} \mu f{:}\alpha_f\,.\,\lambda z{:}\alpha_z\,.\,e_b \rightarrowtail x \qquad \texttt{let}\ x_r{:}\alpha_r = x\ x_a\ \texttt{in}\ e \preceq_{Exp} P}{P \models_{\mathsf{G}} \mathsf{ResOf}(e_b) \rightarrowtail x_r : \alpha_r}$$

$$\textsc{TyAppRes}\quad \frac{P \models_{\mathsf{G}} \mu f{:}\alpha_f\,.\,\Lambda\beta\,.\,e_b \rightarrowtail x \qquad \texttt{let}\ x_r{:}\alpha_r = x\ [\alpha_a]\ \texttt{in}\ e \preceq_{Exp} P}{P \models_{\mathsf{G}} \mathsf{ResOf}(e_b) \rightarrowtail x_r : \alpha_r}$$

**Figure 4.** Flow-Graph-Based Formulation of TCFA

This theorem combined with the soundness of the specification-based formulation of the type- and control-flow analysis given in previous work [5] (asserting that any acceptable pair of abstract environments for a well-typed program approximates the run-time behavior of the program) establishes the soundness of the flow-graph-based formulation of the type- and control-flow analysis: given a well-typed program, the flow-graph-induced abstract type and value environments approximate every concrete type and value environment that arises during execution of the program, because the flow-graph-induced abstract environments are acceptable for the program and acceptable abstract environments for well-typed programs are sound with respect to the operational semantics.

The proof of Theorem 1 relies on two supporting lemmas. The first relates type compatibility in the specification-based formulation and the flow-graph-based formulation:

**Lemma 1** *For all programs $P$ and abstract type environments $\hat{\phi}$, if*

- $\forall \alpha, \tau \,.\, \tau \in \hat{\phi}(\alpha) \;\Leftrightarrow\; P \vDash_{\mathsf{G}} \tau \rightarrowtail \alpha$,

*then*

- $\forall \alpha_1, \alpha_2 \,.\, \hat{\phi} \vDash_{\mathsf{S}} \alpha_1 \approx \alpha_2 \;\Leftrightarrow\; P \vDash_{\mathsf{G}} \alpha_1 \approx \alpha_2$,

PROOF

For the $\Rightarrow$ direction, the proposition is equivalent to $\forall \theta \,.\, \hat{\phi} \vDash_{\mathsf{S}} \alpha_1 \Rrightarrow \theta \,\wedge\, \hat{\phi} \vDash_{\mathsf{S}} \alpha_2 \Rrightarrow \theta \;\Rightarrow\; P \vDash_{\mathsf{G}} \alpha_1 \approx \alpha_2$ and the proof is by induction on $\theta$.

For the $\Leftarrow$ direction, the proof is by induction the derivation of $P \vDash_{\mathsf{G}} \alpha_1 \approx \alpha_2$; inversion of the induction hypothesis reveals the constituent closed types needed to witness the common closed type in the derivation of $\hat{\phi} \vDash_{\mathsf{S}} \alpha_1 \approx \alpha_2$.

The second strengthens the theorem statement to all constituent expressions of the given program:

**Lemma 2** *For all programs $P$, abstract type environments $\hat{\phi}$, and abstract value environments $\hat{\rho}$, if*

- $\forall \alpha, \tau \,.\, \tau \in \hat{\phi}(\alpha) \;\Leftrightarrow\; P \vDash_{\mathsf{G}} \tau \rightarrowtail \alpha$,
- $\forall x, b_{\mathsf{s}} \,.\, b_{\mathsf{s}} \in \hat{\rho}(x) \;\Leftrightarrow\; P \vDash_{\mathsf{G}} b_{\mathsf{s}} \rightarrowtail x$, *and*
- $e \preceq_{Exp} P$,

*then $\hat{\phi}; \hat{\rho} \vDash_{\mathsf{S}} e$.*

PROOF

The proof is by induction on the expression $e$.

- $e \equiv x$: We can derive

$$\frac{}{\hat{\phi}; \hat{\rho} \vDash_{\mathsf{S}} x}\,.$$

- $e \equiv \mathtt{let}\ \alpha = \tau\ \mathtt{in}\ e'$: By the induction hypothesis, we have $\hat{\phi}; \hat{\rho} \vDash_{\mathsf{S}} e'$. By LETTYBND, we have $P \vDash_{\mathsf{G}} \tau \rightarrowtail \alpha$ and, therefore, by the flow-graph-induced abstract environments, we have $\tau \in \hat{\phi}(\alpha)$. We can derive

$$\frac{\tau \in \hat{\phi}(\alpha) \qquad \hat{\phi}; \hat{\rho} \vDash_{\mathsf{S}} e'}{\hat{\phi}; \hat{\rho} \vDash_{\mathsf{S}} \mathtt{let}\ \alpha = \tau\ \mathtt{in}\ e'}\,.$$

- $e \equiv \mathtt{let}\ x{:}\alpha_x = b_{\mathsf{s}}\ \mathtt{in}\ e'$ where $b_{\mathsf{s}} = \mu f{:}\alpha_f \,.\, \lambda z{:}\alpha_z \,.\, e_b$: By the induction hypothesis, we have $\hat{\phi}; \hat{\rho} \vDash_{\mathsf{S}} e_b$ and $\hat{\phi}; \hat{\rho} \vDash_{\mathsf{S}} e'$. By LETEXPBND$_{\mathsf{s}}$ and $\mu\lambda$EXPBND$_{\mathsf{s}}$, we have $P \vDash_{\mathsf{G}} b_{\mathsf{s}} \rightarrowtail^{?} x : \alpha_x$ and $P \vDash_{\mathsf{G}} b_{\mathsf{s}} \rightarrowtail^{?} f : \alpha_f$. Assume $\hat{\phi} \vDash_{\mathsf{S}} \mathsf{TyOf}(b_{\mathsf{s}}) \approx \alpha_x$; by Lemma 1, we have $P \vDash_{\mathsf{G}} \mathsf{TyOf}(b_{\mathsf{s}}) \approx \alpha_x$; by TYVARCOMPATEXPBND$_{\mathsf{s}}$, we have $P \vDash_{\mathsf{G}} b_{\mathsf{s}} \rightarrowtail x$; by the flow-graph-induced abstract environments, we have $b_{\mathsf{s}} \in \hat{\rho}(x)$. Assume $\hat{\phi} \vDash_{\mathsf{S}} \mathsf{TyOf}(b_{\mathsf{s}}) \approx \alpha_f$; by Lemma 1, we have $P \vDash_{\mathsf{G}} \mathsf{TyOf}(b_{\mathsf{s}}) \approx \alpha_f$; by TYVARCOMPATEXPBND$_{\mathsf{s}}$, we have $P \vDash_{\mathsf{G}} b_{\mathsf{s}} \rightarrowtail f$; by the flow-graph-induced abstract environments, we have $b_{\mathsf{s}} \in \hat{\rho}(f)$. We can derive

$$\frac{\begin{array}{c} b_{\mathsf{s}} = \mu f{:}\alpha_f \,.\, \lambda z{:}\alpha_z \,.\, e_b \\ \hat{\phi} \vDash_{\mathsf{S}} \mathsf{TyOf}(b_{\mathsf{s}}) \approx \alpha_x \Rightarrow b_{\mathsf{s}} \in \hat{\rho}(x) \qquad \hat{\phi} \vDash_{\mathsf{S}} \mathsf{TyOf}(b_{\mathsf{s}}) \approx \alpha_f \Rightarrow b_{\mathsf{s}} \in \hat{\rho}(f) \\ \hat{\phi}; \hat{\rho} \vDash_{\mathsf{S}} e_b \qquad \hat{\phi}; \hat{\rho} \vDash_{\mathsf{S}} e' \end{array}}{\hat{\phi}; \hat{\rho} \vDash_{\mathsf{S}} \mathtt{let}\ x{:}\alpha_x = \mu f{:}\alpha_f \,.\, \lambda z{:}\alpha_z \,.\, e_b\ \mathtt{in}\ e'}\,.$$

- $e \equiv \mathtt{let}\ x{:}\alpha_x = b_{\mathsf{s}}\ \mathtt{in}\ e'$ where $b_{\mathsf{s}} = \mu f{:}\alpha_f \,.\, \Lambda\beta \,.\, e_b$: Similar to the previous case.
- $e \equiv \mathtt{let}\ x{:}\alpha_x = x_f\ x_a\ \mathtt{in}\ e'$: By the induction hypothesis, we have $\hat{\phi}; \hat{\rho} \vDash_{\mathsf{S}} e'$. Assume $\mu f{:}\alpha_f \,.\, \lambda z{:}\alpha_z \,.\, e_b \in \hat{\rho}(x_f)$. By the flow-graph-induced abstract environments, we have $P \vDash_{\mathsf{G}} \mu f{:}\alpha_f \,.\, \lambda z{:}\alpha_z \,.\, e_b \rightarrowtail x_f$. By EXPAPPARG and EXPAPPRES, we have $P \vDash_{\mathsf{G}} x_a \rightarrowtail z : \alpha_z$ and $P \vDash_{\mathsf{G}} \mathsf{ResOf}(e_b) \rightarrowtail x : \alpha_x$. Assume $b_{\mathsf{s}} \in \hat{\rho}(x_a)$ and $\hat{\phi} \vDash_{\mathsf{S}} \mathsf{TyOf}(b_{\mathsf{s}}) \approx \alpha_z$; by the flow-graph-induced abstract environments, we have $P \vDash_{\mathsf{G}} b_{\mathsf{s}} \rightarrowtail x_a$ and by Lemma 1, we have $P \vDash_{\mathsf{G}} \mathsf{TyOf}(b_{\mathsf{s}}) \approx \alpha_z$; by TRANSEXPBND$_{\mathsf{s}}$, we have $P \vDash_{\mathsf{G}} b_{\mathsf{s}} \rightarrowtail z : \alpha_z$ and by TYVARCOMPATEXPBND$_{\mathsf{s}}$, we have $P \vDash_{\mathsf{G}} b_{\mathsf{s}} \rightarrowtail z$; by the flow-graph-induced abstract environments, we have $b_{\mathsf{s}} \in \hat{\rho}(z)$. Assume $b_{\mathsf{s}} \in \hat{\rho}(\mathsf{ResOf}(e_b))$ and $\hat{\phi} \vDash_{\mathsf{S}} \mathsf{TyOf}(b_{\mathsf{s}}) \approx \alpha_x$; by the flow-graph-induced abstract environments, we have $P \vDash_{\mathsf{G}} b_{\mathsf{s}} \rightarrowtail \mathsf{ResOf}(e_b)$ and by Lemma 1, we have $P \vDash_{\mathsf{G}} \mathsf{TyOf}(b_{\mathsf{s}}) \approx \alpha_x$; by TRANSEXPBND$_{\mathsf{s}}$, we have $P \vDash_{\mathsf{G}} b_{\mathsf{s}} \rightarrowtail x : \alpha_x$ and by TYVARCOMPATEXPBND$_{\mathsf{s}}$, we have $P \vDash_{\mathsf{G}} b_{\mathsf{s}} \rightarrowtail x$; by the flow-graph-induced abstract environments, we have $b_{\mathsf{s}} \in \hat{\rho}(x)$. We can derive

$$\frac{\begin{array}{l} \forall \mu f{:}\alpha_f \,.\, \lambda z{:}\alpha_z \,.\, e_b \in \hat{\rho}(x_f) \,. \\ \left(\begin{array}{l} \forall b_{\mathsf{s}} \in \hat{\rho}(x_a) \,.\, \hat{\phi} \vDash_{\mathsf{S}} \mathsf{TyOf}(b_{\mathsf{s}}) \approx \alpha_z \Rightarrow b_{\mathsf{s}} \in \hat{\rho}(z) \,\wedge \\ \forall b_{\mathsf{s}} \in \hat{\rho}(\mathsf{ResOf}(e_b)) \,.\, \hat{\phi} \vDash_{\mathsf{S}} \mathsf{TyOf}(b_{\mathsf{s}}) \approx \alpha_x \Rightarrow b_{\mathsf{s}} \in \hat{\rho}(x) \end{array}\right) \end{array} \qquad \hat{\phi}; \hat{\rho} \vDash_{\mathsf{S}} e'}{\hat{\phi}; \hat{\rho} \vDash_{\mathsf{S}} \mathtt{let}\ x{:}\alpha_x = x_f\ x_a\ \mathtt{in}\ e'}\,.$$

- $e \equiv \mathtt{let}\ x{:}\alpha_x = x_f\ [\alpha_a]\ \mathtt{in}\ e'$. By the induction hypothesis, we have $\hat{\phi}; \hat{\rho} \vDash_{\mathsf{S}} e'$. Assume $\mu f{:}\alpha_f \,.\, \Lambda\beta \,.\, e_b \in \hat{\rho}(x_f)$. By the flow-graph-induced abstract environments, we have $P \vDash_{\mathsf{G}} \mu f{:}\alpha_f \,.\, \Lambda\beta \,.\, e_b \rightarrowtail x_f$. By TYAPPARG and TYAPPRES, we have $P \vDash_{\mathsf{G}} \alpha_a \rightarrowtail \beta$ and $P \vDash_{\mathsf{G}} \mathsf{ResOf}(e_b) \rightarrowtail x : \alpha_x$. Assume $\tau \in \hat{\phi}(\alpha_a)$; by the flow-graph-induced abstract environments, we have $P \vDash_{\mathsf{G}} \tau \rightarrowtail \alpha_a$; by TRANSTYBND, we have $P \vDash_{\mathsf{G}} \tau \rightarrowtail \beta$; by the flow-graph-induced abstract environments, we have $\tau \in \hat{\phi}(\beta)$. Assume $b_{\mathsf{s}} \in \hat{\rho}(\mathsf{ResOf}(e_b))$ and $\hat{\phi} \vDash_{\mathsf{S}} \mathsf{TyOf}(b_{\mathsf{s}}) \approx \alpha_x$; by the flow-graph-induced abstract environments, we have $P \vDash_{\mathsf{G}} b_{\mathsf{s}} \rightarrowtail \mathsf{ResOf}(e_b)$ and by Lemma 1, we have $P \vDash_{\mathsf{G}} \mathsf{TyOf}(b_{\mathsf{s}}) \approx \alpha_x$; by TRANSEXPBND$_{\mathsf{s}}$, we have $P \vDash_{\mathsf{G}} b_{\mathsf{s}} \rightarrowtail x : \alpha_x$ and by TYVARCOMPATEXPBND$_{\mathsf{s}}$, we have $P \vDash_{\mathsf{G}} b_{\mathsf{s}} \rightarrowtail x$; by the flow-graph-induced abstract environments, we have $b_{\mathsf{s}} \in \hat{\rho}(x)$. We can derive

$$\frac{\begin{array}{l} \forall \mu f{:}\alpha_f \,.\, \Lambda\beta \,.\, e_b \in \hat{\rho}(x_f) \,. \\ \left(\begin{array}{l} \forall \tau \in \hat{\phi}(\alpha_a) \,.\, \tau \in \hat{\phi}(\beta) \,\wedge \\ \forall b_{\mathsf{s}} \in \hat{\rho}(\mathsf{ResOf}(e_b)) \,.\, \hat{\phi} \vDash_{\mathsf{S}} \mathsf{TyOf}(b_{\mathsf{s}}) \approx \alpha_x \Rightarrow b_{\mathsf{s}} \in \hat{\rho}(x) \end{array}\right) \end{array} \qquad \hat{\phi}; \hat{\rho} \vDash_{\mathsf{S}} e'}{\hat{\phi}; \hat{\rho} \vDash_{\mathsf{S}} \mathtt{let}\ x{:}\alpha_x = x_f\ [\alpha_a]\ \mathtt{in}\ e'}\,.$$

We conjecture that the flow-induced abstract environments for a given program are, in fact, the minimum acceptable abstract type and value environments for the program.

## 5. Algorithm

In Figures 5, 6, and 7, we give a direct algorithm implementing the flow-graph-based formulation of the type- and control-flow analysis, based on Midtgaard and VanHorn's lucid presentation of a cubic algorithm for 0CFA [12]. The algorithm returns a result set $R$ whose elements correspond to judgments from Figure 4 that are proven to be derivable with respect to the input program $P$. After an initialization phase, the algorithm uses a work-queue $W$ to process each element that is added to $R$; when a newly added element is processed, all of the inference rules for which the newly added element could be an antecedent are inspected to determine if the corresponding conclusion can now be added to $R$. In order to achieve our desired time complexity, there is a map $T$ from elements of the form $\alpha_1 \approx \alpha_2$ to a queue of conclusions that may be added to $R$ when $\alpha_1 \approx \alpha_2$ is proved to be derivable; the queues in $T$ will also serve as "banks" holding credit for the amortized complexity analysis.

```
procedure TCFA(P)                                       ▷ O(l^3 + m^4) = O(l^2) + O(m^2) + O(1) + O(m^2)
                                                                       + O(l) + O(m) + O(m^2) + O(m^2)
                                                                       + O(l^3) + O(l^2) + O(l^3)
                                                                       + O(m^4) + O(m^3) + O(m^4) + O(m^2)
    R ← Set.newEmpty()                                  ▷ O(l^2) + O(m^2)
    W ← Queue.newEmpty()                                ▷ O(1)
    T ← Map.newEmpty()                                  ▷ O(m^2)

    for all let x:α_x = b_s in e ⪯_Exp P do             ▷ O(l) = O(l) × O(1)
        Set.insert(R, b_s ↣^? x : α_x)
        Queue.push(W, b_s ↣^? x : α_x)
        match b_s with
            case μf:α_f.λz:α_z.e_b do
                Set.insert(R, b_s ↣^? f : α_f)
                Queue.push(W, b_s ↣^? f : α_f)
            end case
            case μf:α_f.Λβ.e_b do
                Set.insert(R, b_s ↣^? f : α_f)
                Queue.push(W, b_s ↣^? f : α_f)
            end case
        end match
    end for

    for all let α = τ in e ⪯_Exp P do                   ▷ O(m) = O(m) × O(1)
        Set.insert(R, τ ↣ α)
        Queue.push(W, τ ↣ α)
    end for

    for all α_1 ⪯_TyVar P do                            ▷ O(m^2) = O(m) × O(m)
        for all α_2 ⪯_TyVar P do                            ▷ O(m) = O(m) × O(1)
            Map.set(T, α_1 ≋ α_2, Queue.newEmpty())
        end for
    end for

    for all τ_1 ⪯_TyBnd P do                            ▷ O(m^2) = O(m) × O(m)
        for all τ_2 ⪯_TyBnd P do                            ▷ O(m) = O(m) × O(1)
            match ⟨τ_1, τ_2⟩ with
                case ⟨α_a1 → α_b1, α_a2 → α_b2⟩ do
                    c ← Counter.new(2)
                    Queue.push(Map.get(T, α_a1 ≋ α_a2), ⟨c, τ_1 ≋ τ_2⟩)    ▷ O(1) credit into T[α_a1 ≋ α_a2] queue
                    Queue.push(Map.get(T, α_b1 ≋ α_b2), ⟨c, τ_1 ≋ τ_2⟩)    ▷ O(1) credit into T[α_b1 ≋ α_b2] queue
                end case
                case ⟨∀. α_b1, ∀. α_b2⟩ do
                    c ← Counter.new(1)
                    Queue.push(Map.get(T, α_b1 ≋ α_b2), ⟨c, τ_1 ≋ τ_2⟩)    ▷ O(1) credit into T[α_b1 ≋ α_b2] queue
                end case
                case ⟨#n, #m⟩ do
                    if n = m then
                        Set.insert(R, τ_1 ≋ τ_2)
                        Queue.push(W, τ_1 ≋ τ_2)
                    end if
                end case
            end match
        end for
    end for
```

**Figure 5.** TCFA Algorithm

**while** $\neg$Queue.empty?$(W)$ **do**
**match** Queue.pop$(W)$ **with**

**case** $x \rightarrowtail y : \alpha_y$ **do** $\triangleright\ O(l^3) = O(l^2) \times O(l)$
**for all** $b_s \rightarrowtail x \in R$ **do** $\triangleright\ O(l) = O(l) \times O(1)$
**if** $b_s \rightarrowtail^? y : \alpha_y \notin R$ **then**
Set.insert$(R, b_s \rightarrowtail^? y : \alpha_y)$
Queue.push$(W, b_s \rightarrowtail^? y : \alpha_y)$
**end if**
**end for**
**end case**

**case** $b_s \rightarrowtail^? x : \alpha_x$ **do** $\triangleright\ O(l^2) = O(l^2) \times O(1)$
**if** TyOf$(b_s) \approxeq \alpha_x \in R$ **then**
**if** $b_s \rightarrowtail x \notin R$ **then**
Set.insert$(R, b_s \rightarrowtail x)$
Queue.push$(W, b_s \rightarrowtail x)$
**end if**
**else**
Queue.push(Map.get$(T$, TyOf$(b_s) \approxeq \alpha_x), b_s \rightarrowtail x)$ $\triangleright\ O(1)$ credit into $T[$TyOf$(b_s) \approxeq \alpha_x]$ queue
**end if**
**end case**

**case** $b_s \rightarrowtail x$ **do** $\triangleright\ O(l^3) = O(l^2) \times O(l)$
**for all** $x \rightarrowtail y : \alpha_y \in R$ **do** $\triangleright\ O(l) = O(l) \times O(1)$
**if** $b_s \rightarrowtail^? y : \alpha_y \notin R$ **then**
Set.insert$(R, b_s \rightarrowtail^? y : \alpha_y)$
Queue.push$(W, b_s \rightarrowtail^? y : \alpha_y)$
**end if**
**end for**
**match** $b_s$ **with**
**case** $\mu f{:}\alpha_f\,.\,\lambda z{:}\alpha_z\,.\,e_b$ **do**
$x_b \leftarrow$ ResOf$(e_b)$
**for all** `let` $x_r{:}\alpha_r$ `=` $x\ x_a$ `in` $e \preceq_{Exp} P$ **do** $\triangleright\ O(l) = O(l) \times O(1)$
**if** $x_a \rightarrowtail z : \alpha_z \notin R$ **then**
Set.insert$(R, x_a \rightarrowtail z : \alpha_z)$
Queue.push$(W, x_a \rightarrowtail z : \alpha_z)$
**end if**
**if** $x_b \rightarrowtail x_r : \alpha_r \notin R$ **then**
Set.insert$(R, x_b \rightarrowtail x_r : \alpha_r)$
Queue.push$(W, x_b \rightarrowtail x_r : \alpha_r)$
**end if**
**end for**
**end case**
**case** $\mu f{:}\alpha_f\,.\,\Lambda\beta\,.\,e_b$ **do**
$x_b \leftarrow$ ResOf$(e_b)$
**for all** `let` $x_r{:}\alpha_r$ `=` $x$ `[`$\alpha_a$`]` `in` $e \preceq_{Exp} P$ **do** $\triangleright\ O(l) = O(l) \times O(1)$
**if** $\alpha_a \rightarrowtail \beta \notin R$ **then**
Set.insert$(R, \alpha_a \rightarrowtail \beta)$
Queue.push$(W, \alpha_a \rightarrowtail \beta)$
**end if**
**if** $x_b \rightarrowtail x_r : \alpha_r \notin R$ **then**
Set.insert$(R, x_b \rightarrowtail x_r : \alpha_r)$
Queue.push$(W, x_b \rightarrowtail x_r : \alpha_r)$
**end if**
**end for**
**end case**
**end match**
**end case**

**Figure 6.** TCFA Algorithm (continued)

**case** $\tau \rightarrowtail \alpha$ **do** $\triangleright\ O(m^4) = O(m^2) \times O(m^2)$
  **for all** $\alpha \rightarrowtail \beta \in R$ **do** $\triangleright\ O(m) = O(m) \times O(1)$
    **if** $\tau \rightarrowtail \beta \notin R$ **then**
      $\mathsf{Set.insert}(R, \tau \rightarrowtail \beta)$
      $\mathsf{Queue.push}(W, \tau \rightarrowtail \beta)$
    **end if**
  **end for**
  **for all** $\tau' \rightarrowtail \alpha' \in R$ **do** $\triangleright\ O(m^2) = O(m^2) \times O(1)$
    **if** $\tau \approxeq \tau' \in R$ **then**
      **if** $\alpha \approxeq \alpha' \notin R$ **then**
        $\mathsf{Set.insert}(R, \alpha \approxeq \alpha')$
        $\mathsf{Queue.push}(W, \alpha \approxeq \alpha')$
      **end if**
    **end if**
  **end for**
**end case**

**case** $\alpha \rightarrowtail \beta$ **do** $\triangleright\ O(m^3) = O(m^2) \times O(m)$
  **for all** $\tau \rightarrowtail \alpha \in R$ **do** $\triangleright\ O(m) = O(m) \times O(1)$
    **if** $\tau \rightarrowtail \beta \notin R$ **then**
      $\mathsf{Set.insert}(R, \tau \rightarrowtail \beta)$
      $\mathsf{Queue.push}(R, \tau \rightarrowtail \beta)$
    **end if**
  **end for**
**end case**

**case** $\tau_1 \approxeq \tau_2$ **do** $\triangleright\ O(m^4) = O(m^2) \times O(m^2)$
  **for all** $\tau_1 \rightarrowtail \alpha_1 \in R$ **do** $\triangleright\ O(m^2) = O(m) \times O(m)$
    **for all** $\tau_2 \rightarrowtail \alpha_2 \in R$ **do** $\triangleright\ O(m) = O(m) \times O(1)$
      **if** $\alpha_1 \approxeq \alpha_2 \notin R$ **then**
        $\mathsf{Set.insert}(R, \alpha_1 \approxeq \alpha_2)$
        $\mathsf{Queue.push}(W, \alpha_1 \approxeq \alpha_2)$
      **end if**
    **end for**
  **end for**
**end case**

**case** $\alpha_1 \approxeq \alpha_2$ **do** $\triangleright\ O(m^2) = O(m^2) \times O(1)$
  **while** $\neg\mathsf{Queue.empty?}(\mathsf{Map.get}(T, \alpha_1 \approxeq \alpha_2))$ **do**
    **match** $\mathsf{Queue.pop}(\mathsf{Map.get}(T, \alpha_1 \approxeq \alpha_2))$ **with** $\triangleright\ O(1)$ credit from $T[\alpha_{a1} \approxeq \alpha_{a2}]$ queue
      **case** $b_{\mathsf{s}} \rightarrowtail x$ **do**
        **if** $b_{\mathsf{s}} \rightarrowtail x \notin R$ **then**
          $\mathsf{Set.insert}(R, b_{\mathsf{s}} \rightarrowtail x)$
          $\mathsf{Queue.push}(W, b_{\mathsf{s}} \rightarrowtail x)$
        **end if**
      **end case**
      **case** $\langle c, \tau_1 \approxeq \tau_2 \rangle$ **do**
        $\mathsf{Counter.dec}(c)$
        **if** $\mathsf{Counter.get}(c) = 0$ **then**
          **if** $\tau_1 \approxeq \tau_2 \notin R$ **then**
            $\mathsf{Set.insert}(R, \tau_1 \approxeq \tau_2)$
            $\mathsf{Queue.push}(W, \tau_1 \approxeq \tau_2)$
          **end if**
        **end if**
      **end case**
    **end match**
  **end while**
**end case**

**end match**
**end while**

**return** $R$
**end procedure**

**Figure 7.** TCFA Algorithm (continued)

### 5.1 Commentary

***Initialization Phase*** The first initialization phase (lines 5–18) adds to $R$ and $W$ all elements of the form $b_{\mathsf{s}} \rightarrowtail^{?} x : \alpha_x$ that are derivable using the rules whose conclusion follows directly from the input program: LetExpBnd$_{\mathsf{s}}$, $\mu\lambda$ExpBnd$_{\mathsf{s}}$, and $\mu\Lambda$ExpBnd$_{\mathsf{s}}$. Similarly, the second initialization phase (lines 19–22) adds to $R$ and $W$ all elements of the form $\tau \rightarrowtail \alpha$ that are derivable using the rule LetTyBnd$_{\mathsf{s}}$.

The third initialization phase (lines 23–27) prepares the map $T$, creating an empty queue for each pair of type variables that appear in the input program.

The fourth initialization phase (lines 28–48) handles the rules TyCompatArrow, TyCompatForAll, and TyCompatTyIdx for all type binds that appear in the input program. When $\tau_1$ and $\tau_2$ are arrow types, then $\tau_1 \approxeq \tau_2$ is derivable using the rule TyCompatArrow when the argument type variables are known to be compatible and the result type variables are known to be compatible. Therefore, we create a counter $c$ initialized with the value 2 and add the element $\langle c, \tau_1 \approxeq \tau_2 \rangle$ to the queues in map $T$ for the elements $\alpha_{a1} \approxeq \alpha_{a2}$ and $\alpha_{b1} \approxeq \alpha_{b2}$. The element $\langle c, \tau_1 \approxeq \tau_2 \rangle$ indicates that $\tau_1$ and $\tau_2$ will be known to be compatible when two pairs of type variables are known to be compatible; when each of $\alpha_{a1} \approxeq \alpha_{a2}$ and $\alpha_{b1} \approxeq \alpha_{b2}$ are known to be compatible, the counter will be decremented and when the counter is zero, $\tau_1 \approxeq \tau_2$ will be added to $R$ and $W$ (see lines 148–156). Similarly, when $\tau_1$ and $\tau_2$ are forall types, then $\tau_1 \approxeq \tau_2$ is derivable using the rule TyCompatForAll when the result type variables are known to be compatible and we create a counter $c$ initialized with the value 1 and add the element $\langle c, \tau_1 \approxeq \tau_2 \rangle$ to the queue in map $T$ for the element $\alpha_{b1} \approxeq \alpha_{b2}$. Finally, when $\tau_1$ and $\tau_2$ are the same type index, then $\tau_1 \approxeq \tau_2$ is immediately derivable using the rule TyCompatTyIdx and $\tau_1 \approxeq \tau_2$ is added to $R$ and $W$.

***Work-queue Phase*** The work-queue phase repeatedly pops an element from the work-queue $W$ and processes the element (possibly adding new elements to $R$ and $W$) until $W$ is empty. To process an element, all of the inference rules for which the element could be an antecedent are inspected to determine if the corresponding conclusion can now be added to $R$ and $W$.

When the work-queue element is of the form $x \rightarrowtail y : \alpha_y$ (lines 51–58), only the rule TransExpBnd$_{\mathsf{s}}$ need be inspected. For each $b_{\mathsf{s}} \rightarrowtail x$ that is already known to be derivable, then TransExpBnd$_{\mathsf{s}}$ may derive $b_{\mathsf{s}} \rightarrowtail^{?} y : \alpha_y$ and it is added to $R$ and $W$.

When the work-queue element is of the form $b_{\mathsf{s}} \rightarrowtail x : \alpha_x$ (lines 59–68), only the rule TyVarCompatExpBnd$_{\mathsf{s}}$ need be inspected. If $\mathsf{TyOf}(b_{\mathsf{s}})$ and $\alpha_x$ are already known to be compatible, then TyVarCompatExpBnd$_{\mathsf{s}}$ may derive $b_{\mathsf{s}} \rightarrowtail x$ and it is added to $R$ and $W$. If $\mathsf{TyOf}(b_{\mathsf{s}})$ and $\alpha_x$ are not yet known to be compatible, then the element $b_{\mathsf{s}} \rightarrowtail x$ is added to the queue given by $\mathsf{Map.get}(T, \mathsf{TyOf}(b_{\mathsf{s}}) \approxeq \alpha_x)$, indicating that when $\mathsf{TyOf}(b_{\mathsf{s}})$ and $\alpha_x$ are known to be compatible, $b_{\mathsf{s}} \rightarrowtail x$ will be added to $R$ and $W$ (see lines 142–147).

When the work-queue element is of the form $b_{\mathsf{s}} \rightarrowtail x$ (lines 69–104), the rules TransExpBnd$_{\mathsf{s}}$, ExpAppArg, ExpAppRes, TyAppArg, and TyAppRes need to be inspected. For each $x \rightarrowtail y : \alpha_y$ that is already known to be derivable, then TransExpBnd$_{\mathsf{s}}$ may derive $b_{\mathsf{s}} \rightarrowtail^{?} y : \alpha_y$ and it is added to $R$ and $W$. When $b_{\mathsf{s}}$ is of the form $\mu f{:}\alpha_f\,.\,\lambda z{:}\alpha_z\,.\,e_b$ where $x_b = \mathsf{ResOf}(e_b)$ (lines 77–89), all expression applications `let` $x_r{:}\alpha_r$ `=` $x\ x_a$ `in` $e$ in the input program are examined to determine if ExpAppArg may derive $x_a \rightarrowtail z : \alpha_z$ and if ExpAppRes may derive $x_b \rightarrowtail x_r : \alpha_r$. Similarly, when $b_{\mathsf{s}}$ is of the form $\mu f{:}\alpha_f\,.\,\Lambda\beta\,.\,e_b$ where $x_b = \mathsf{ResOf}(e_b)$ (lines 90–102), all expression applications `let` $x_r{:}\alpha_r$ `=` $x$ `[`$\alpha_a$`]` `in` $e$ in the input program are examined to determine if TyAppArg may derive $\alpha_a \rightarrowtail \beta$ and if TyAppRes may derive $x_b \rightarrowtail x_r : \alpha_r$.

When the work-queue element is of the form $\tau \rightarrowtail \alpha$ (lines 105–120), the rules TransTyBnd and TyVarCompat need to be inspected. For each $\alpha \rightarrowtail \beta$ that is already known to be derivable, then TransTyBnd may derive $\tau \rightarrowtail \beta$ and it is added to $R$ and $W$. For each $\tau' \rightarrowtail \alpha'$ that is already known to be derivable, if $\tau$ and $\tau'$ are already known to be compatible, then TyVarCompat may derive $\alpha \approxeq \alpha'$ and it is added to $R$ and $W$.

When the work-queue element is of the form $\alpha \rightarrowtail \beta$ (lines 121–128), only the rule TransTyBnd need be inspected. For each $\tau \rightarrowtail \alpha$ that is already known to be derivable, then TransTyBnd may derive $\tau \rightarrowtail \beta$ and it is added to $R$ and $W$.

When the work-queue element is of the form $\tau_1 \approxeq \tau_2$ (lines 129–138), only the rule TyVarCompat need be inspected. For each $\tau_1 \rightarrowtail \alpha_1$ and $\tau_2 \rightarrowtail \alpha_2$ that are known to be derivable, then TyVarCompat may derive $\alpha_1 \approxeq \alpha_2$ and it is added to $R$ and $W$.

Finally, when the work-queue element is of the form $\alpha_1 \approxeq \alpha_2$ (lines 139–159), the rules TyVarCompatExpBnd$_{\mathsf{s}}$ and TyVarCompat need to be inspected. Each time that $b_{\mathsf{s}} \rightarrowtail x : \alpha_x$ was known to be derivable but $\mathsf{TyOf}(b_{\mathsf{s}})$ and $\alpha_x$ were not yet known to be compatible (preventing TyVarCompatExpBnd$_{\mathsf{s}}$ from deriving $b_{\mathsf{s}} \rightarrowtail x$), an element of the form $b_{\mathsf{s}} \rightarrowtail x$ was added to the queue given by $\mathsf{Map.get}(T, \mathsf{TyOf}(b_{\mathsf{s}}) \approxeq \alpha_x)$ (see line 66); hence, processing these elements of the queue will add each $b_{\mathsf{s}} \rightarrowtail x$ that may be derived by TyVarCompatExpBnd$_{\mathsf{s}}$ to $R$ and $W$. For each pair of type binds $\tau_1$ and $\tau_2$ whose compatibility depends upon the compatibility of $\alpha_1$ and $\alpha_2$ (and possibly upon the compatibility of other pairs of type variables), an element of the form $\langle c, \tau_1 \approxeq \tau_2 \rangle$, where $c$ indicates the total number of pairs of type variables whose compatibility will establish the compatibility of $\tau_1$ and $\tau_2$, was added to the queue given by $\mathsf{Map.get}(T, \alpha_1 \approxeq \alpha_2)$ (see lines 33, 34, and 38); hence, processing these elements of the queue will add each $\tau_1 \approxeq \tau_2$ that may be derived by TyVarCompat to $R$ and $W$.

***Termination*** Note that throughout the algorithm, whenever an element is added to the result set $R$, it is simultaneously added to the work-queue $W$. Furthermore, an element is added to $R$ and $W$ only after checking that the element is not already in $R$, except during the initialization phase when all elements added to $R$ and $W$ are necessarily not already in $R$. Hence, elements are only added to $W$ once and the work-queue phase of the algorithm terminates because, for a given input program, there are only a finite number of elements that may be added to $R$ and $W$.

### 5.2 Complexity

***Preliminaries*** Before analyzing the time complexity of the algorithm, we first make some (standard) assumptions about the representation of the input program.

We assume that all `let`-, $\mu$-, and $\lambda$-bound expression variables and all `let`-, and $\Lambda$-bound type variables in the program are distinct. We further assume that expression variables and type variables can be mapped (in $O(1)$ time) to unique integers (for $O(1)$ time indexing into an array) and that integers can be mapped (in $O(1)$ time) to corresponding expression variables and type variables.[3] Given the assumption that all `let`-, $\mu$-, and $\lambda$-bound expression variables in the program are unique, each expression variable in the program is annotated with a single type variable at its unique binding occurrence. We therefore assume that expression variables can be mapped (in $O(1)$ time) to its annotating type variable. Given the assumption that all $\mu$-bound expression variables in the program are unique, each simple expression bind in the program is unique and can be mapped (in $O(1)$ time) to and from unique integers.[4] We do not assume that each type bind in the program is

[3] These mappings can established with a linear-time preprocessing step.

[4] In a richer language with simple-expression-bind forms that do not include a bound expression variable (e.g., $\langle x_1, x_2 \rangle$ pairs), we can assume a num-

unique, but we do assume that each type bind in the program can be mapped (in $O(1)$ time) to and from unique integers. Finally, we assume that $\mathsf{ResOf}(\cdot)$ can be computed in $O(1)$ time.[5]

***Data Structures and Operations*** We next consider the data structures used to implement the result set $R$ and the map $T$ and the cost of various operations.

The result set $R$ is implemented as seven multi-dimensional arrays, each corresponding to one of the seven judgments from Figure 4. Given the assumptions above, it is easy to see that the arrays corresponding to $\tau_1 \approxeq \tau_2$, $\alpha_1 \approxeq \alpha_2$, $\tau \rightarrowtail \alpha$, $\alpha \rightarrowtail \beta$, and $b_{\mathsf{s}} \rightarrowtail x$ are simple two-dimensional arrays with $O(1)$ time indexing by mapping components to unique integers. Furthermore, the arrays corresponding to $b_{\mathsf{s}} \rightarrowtail x : \alpha_x$ and $x \rightarrowtail y : \alpha_y$ can also be implemented with simple two-dimensional arrays (indexed by $b_{\mathsf{s}}/x$ and $x/y$, respectively), because the type variable is always the single type variable at the unique binding occurrence of the expression variable and can be left implicit. Thus, queries like $b_{\mathsf{s}} \rightarrowtail x \notin R$ and operations like $\mathsf{Set.insert}(R, b_{\mathsf{s}} \rightarrowtail x)$ can be performed in constant time. Loops like "**for all** $b_{\mathsf{s}} \rightarrowtail x \in R$ **do**" for fixed $b_{\mathsf{s}}$ instantiating $x$ or for fixed $x$ instantiating $b_{\mathsf{s}}$ can be implemented as a linear scan of an array column or array row. Initializing $R$ can be performed in quadratic time.

The map $T$ is implemented with a simple two-dimensional array, indexed by pairs of type variables. Operations like $\mathsf{Map.set}(T, \alpha_1 \approxeq \alpha_2, q)$ and $\mathsf{Map.get}(T, \alpha_1 \approxeq \alpha_2)$ can be performed in constant time and initializing $T$ can be performed in quadratic time.

The work-queue $W$ and the queues in map $T$ are implemented with a simple linked-list queue. Queries like $\mathsf{Queue.empty?}(W)$ and operations like $\mathsf{Queue.push}(W, b_{\mathsf{s}} \rightarrowtail x)$ and $\mathsf{Queue.pop}(W)$ can be performed in constant time.

***Coarse Analysis*** We first argue that the algorithm is $O(n^4)$ time, where $n$ is the size of the input program $P$. First, note that there are $O(n)$ type variables, $O(n)$ type binds, $O(n)$ expression variables, and $O(n)$ simple expression binds in the program. Thus, the result set $R$ requires $O(n^2)$ space for (and is $O(n^2)$ time to create) each of the seven two-dimensional arrays and the map $T$ requires $O(n^2)$ space for (and is $O(n^2)$ time to create) the two-dimensional array.

The first initialization phase is $O(n)$ time to traverse the program and process each simple expression bind. Similarly, the second initialization phase is $O(n)$ time to traverse the program and process each type bind. The third initialization phase is $O(n^2)$ time to process each pair of type variables. The fourth initialization phase is $O(n^2)$ time to process each pair of type binds. Altogether, the initialization phase is $O(n^2) = O(n)+O(n)+O(n^2)+O(n^2)$ time.

As noted above, elements are only added to $W$ once. Therefore, the time complexity of the "**while** $\neg\mathsf{Queue.empty?}(W)$ **do**"-loop is the sum of the time required to process an element of each kind times the number of elements of that kind. There are $O(n^2)$ elements of the form $x \rightarrowtail y : \alpha_y$ (recall that the $\alpha_y$ is implicitly determined by the $y$) and processing an $x \rightarrowtail y : \alpha_y$ element is $O(n)$ time to scan for all $b_{\mathsf{s}} \rightarrowtail x \in R$. There are $O(n^2)$ elements of the form $b_{\mathsf{s}} \overset{?}{\rightarrowtail} x : \alpha_x$ and processing a $b_{\mathsf{s}} \overset{?}{\rightarrowtail} x : \alpha_x$ element is $O(1)$ time. There are $O(n^2)$ elements of the form $b_{\mathsf{s}} \rightarrowtail x$ and processing a $b_{\mathsf{s}} \rightarrowtail x$ element is $O(n)$ time to scan all $x \rightarrowtail y : \alpha_y \in R$ and $O(n)$ time to find all $\texttt{let}\ x_r\!:\!\alpha_r\ \texttt{=}\ x\ x_a\ \texttt{in}\ e \preceq_{Exp} P$ and to find all $\texttt{let}\ x_r\!:\!\alpha_r\ \texttt{=}\ x\ \texttt{[}\alpha_a\texttt{]}\ \texttt{in}\ e \preceq_{Exp} P$. There are $O(n^2)$ elements of the form $\tau \rightarrowtail \alpha$ and processing a $\tau \rightarrowtail \alpha$ element is $O(n)$ time to scan for all $\alpha \rightarrowtail \beta \in R$ and $O(n^2)$ to process all $\pi \rightarrowtail \beta \in R$. There are $O(n^2)$ elements of the form $\alpha \rightarrowtail \beta$ and processing an $\alpha \rightarrowtail \beta$ element is $O(n)$ time to scan for all $\tau \rightarrowtail \alpha \in R$. There are $O(n^2)$ elements of the form $\tau_1 \approxeq \tau_2$ and processing a $\tau_1 \approxeq \tau_2$ element is $O(n^2)$ time to scan for all $\tau_1 \rightarrowtail \alpha_1 \in R$ and $\tau_2 \rightarrowtail \alpha_2 \in R$. There are $O(n^2)$ elements of the form $\alpha_1 \approxeq \alpha_2$, processing an $\alpha_1 \approxeq \alpha_2$ element must process each element in the queue $\mathsf{Map.get}(T, \alpha_1 \approxeq \alpha_2)$, and, therefore, the time complexity to process an $\alpha_1 \approxeq \alpha_2$ element is the sum of the time required to process the elements in the queue of each kind times the number of elements of that kind; there are $O(n^2)$ elements of the form $b_{\mathsf{s}} \rightarrowtail x$ in the queue (since an element of the form $b_{\mathsf{s}} \rightarrowtail x$ are added at most once to at most one queue (see line 66)) and processing an $b_{\mathsf{s}} \rightarrowtail x$ element is $O(1)$ time and there are $O(n^2)$ elements of the form $\langle c, \tau_1 \approxeq \tau_2 \rangle$ (since an element of the form $\langle c, \tau_1 \approxeq \tau_2 \rangle$ is added at most twice to at most one queue (see lines 33–34 and line 38)) and processing a $\langle c, \tau_1 \approxeq \tau_2 \rangle$ element is $O(1)$ time. Altogether, the work-queue phase is $O(n^4) = O(n^2) \times O(n) + O(n^2) \times O(1) + O(n^2) \times (O(n) + O(n) + O(n)) + O(n^2) \times (O(n) + O(n^2)) + O(n^2) \times O(n) + O(n^2) \times O(n^2) + O(n^2) \times (O(n^2) \times O(1) + O(n^2) \times O(1))$.

Thus, the entire algorithm is $O(n^4)$. Recall that algorithms for classic (untyped) control-flow analysis have been shown to be $O(n^3)$ [2, 9, 17, 19].

***Refined Analysis*** In order to clarify the relationship between the time complexity of algorithms for classic (untyped) control-flow analysis and our algorithm for type- and control-flow analysis, we perform a refined analysis of our algorithm.

First, note that the quartic components of the algorithm are due to the processing of elements of the form $\tau \rightarrowtail \alpha$, $\tau_1 \approxeq \tau_2$, and $\alpha_1 \approxeq \alpha_2$. Intuitively, the increased time complexity of the algorithm for type- and control-flow analysis compared to algorithms for classic (untyped) control-flow analysis is due to the computation of the type-compatibility relations.

Second, in typical programs of interest, we expect that the total size of the program to be dominated by the contribution of (bound) expression variables and expression binds, with the contribution of (bound) type variables and type binds significantly (asymptotically?) less. For example, a program may have many definitions of and uses of $\texttt{int} \rightarrow \texttt{int}$ functions, all of which can share the same (top-level) $\texttt{let}\ \alpha_{\texttt{i}}\ \texttt{= int in let}\ \alpha_{\texttt{i}\rightarrow\texttt{i}}\ \texttt{=}\ \alpha_{\texttt{i}} \rightarrow \alpha_{\texttt{i}}\ \texttt{in}\ \ldots$ type bindings. Indeed, our ANF representation of types encourages type-level optimizations such as `let`-floating, common subexpression elimination (CSE), and copy propagation, which would further reduce the contribution of types to the total program size. Therefore, we consider it useful to distinguish $l$, the size of (bound) expression variables and expression binds, and $m$, the size of (bound) type variables and type binds, where we have $O(l) + O(m)$ is $O(n)$ and we expect $O(l) \gg O(m)$, though, in the worst-case, both $O(l)$ and $O(m)$ are $O(n)$. We further assume an $O(n)$ preprocessing step that provides an enumeration of all $\texttt{let}\ x\!:\!\alpha_x\ \texttt{=}\ b\ \texttt{in}\ e \preceq_{Exp} P$ in $O(l)$ time and an enumeration of all $\texttt{let}\ \alpha\ \texttt{=}\ \tau\ \texttt{in}\ e \preceq_{Exp} P$ in $O(m)$ time.

We now argue that the algorithm is $O(l^3 + m^4)$ time. First, note that there are $O(m)$ type variables, $O(m)$ type binds, $O(l)$ expression variables, and $O(l)$ simple expression binds in the program. Thus, the result set $R$ requires $O(l^2 + m^2)$ space for (and is $O(l^2 + m^2)$ time to create) the seven two-dimensional arrays and the map $T$ requires $O(m^2)$ space for (and is $O(m^2)$ time to create) the two-dimensional array.

bering of all simple expression binds in the program, similar to the labeling found in textbook presentations of CFA [17].

[5] This can be established either by a linear-time preprocessing step (associating each result variable with its corresponding abstraction) or by changing the representation of expressions to a list of $\alpha$ = $\tau$ and $x\!:\!\alpha_x$ = $b$ bindings paired with the result variable.

The first initialization phase is $O(l)$ time to process each simple expression bind. Similarly, the second initialization phase is $O(m)$ time to process each type bind. The third initialization phase is $O(m^2)$ time to process each pair of type variables. The fourth initialization phase is $O(m^2)$ time to process each pair of type binds; included in this processing time is an $O(1)$ credit "deposited" into the queues in $T$ when pushing elements, which "pre-pays" for the processing of the elements when popped. Altogether, the initialization phase is $O(l + m^2) = O(l) + O(m) + O(m^2) + O(m^2)$ time.

The analysis of the work-queue phase is similar to that performed above: the time complexity of the "**while** $\neg\mathsf{Queue.empty?}(W)$ **do**"-loop is the sum of the time required to process an element of each kind times the number of elements of that kind; we simply refine $n$ to $l$ or $m$ as appropriate. We further perform an amortized analysis of the time complexity to process an $b_{\mathsf{s}} \rightarrowtail^{?} x : \alpha_x$ element and to process an $\alpha_1 \approxeq \alpha_2$ element. Included in the time to process an $b_{\mathsf{s}} \rightarrowtail^{?} x : \alpha_x$ element is an $O(1)$ credit "deposited" into the queue given by $\mathsf{Map.get}(T, \mathsf{TyOf}(b_{\mathsf{s}}) \approxeq \alpha_x)$ when pushing elements, which "pre-pays" for the processing of the elements when popped. As before, processing an $\alpha_1 \approxeq \alpha_2$ element must process each element in the queue $\mathsf{Map.get}(T, \alpha_1 \approxeq \alpha_2)$; however, an $O(1)$ credit may be "withdrawn" from the queue $\mathsf{Map.get}(T, \alpha_1 \approxeq \alpha_2)$ when popping elements and this $O(1)$ credit may be used to "pay" for the popping and processing of the element. Thus, processing an $\alpha_1 \approxeq \alpha_2$ element is (amortized) $O(1)$ time.[6] Altogether, the work-queue phase is $O(l^3 + m^4) = O(l^2) \times O(l) + O(l^2) \times O(1) + O(l^2) \times (O(l) + O(l) + O(l)) + O(m^2) \times (O(m) + O(m^2)) + O(m^2) \times O(m) + O(m^2) \times O(m^2) + O(m^2) \times O(1)$.

Thus, the entire algorithm is $O(l^3 + m^4)$.

## 6. Conclusion

We have given an $O(l^3 + m^4)$ algorithm for a type- and control-flow analysis for System F (with recursion), where $l$ is the size of expressions in the program and $m$ is the size of types in the program. Compared to 0CFA, our type- and control-flow analysis can be more precise, by exploiting the well-typedness of the program under analysis, but is also slightly more expensive, due to the $O(m^4)$ term, though in typical programs, with $l \gg m$, we expect our flow analysis to be an attractive alternative to 0CFA. Furthermore, this is a *worst-case* complexity, which assumes that *every* $\lambda$- and $\Lambda$-expression flows to *every* expression variable and *every* type flows to *every* type variable. Control-flow analyses are useful because they typically find many expression variables with few $\lambda$- and $\Lambda$-expressions and we expect type-flow analysis to also find many type variables with few types. The algorithm only explores the consequences of found flows and should perform well in practice.

## Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. 1065099. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

## References

[1] A. Aiken and B. R. Murphy. Implementing regular tree expressions. In J. Hughes, editor, *FPCA'91: Proceedings of the Fifth ACM Conference on Functional Programming Languages and Computer Architecture*, volume 523 of *Lecture Notes in Computer Science*, pages 427–447, Cambridge, Massachusetts, Aug. 1991. Springer-Verlag.

[2] A. E. Ayers. Efficient closure analysis with reachability. In M. Billaud, P. Castéran, M.-M. Corsini, K. Musumbu, and A. Rauzy, editors, *Actes WSA'92 Workshop on Static Analysis*, Bigre, pages 126–134, Bordeaux, France, Sept. 1992. Atelier Irisa, IRISA, Campus de Beaulieu.

[3] S. Chaudhuri. Subcubic algorithms for recursive state machines. In G. C. Necula and P. Wadler, editors, *POPL'08: Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 159–169, San Francisco, California, Jan. 2008.

[4] C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. In R. Cartwright, editor, *PLDI'93: Proceedings of the ACM SIGPLAN 1993 Conference on Programming Languages Design and Implementation*, pages 237–247, Albuquerque, New Mexico, June 1993. ACM.

[5] M. Fluet. A type- and control-flow analysis for System F. In R. Hinze, editor, *IFL'12: Post-Proceedings of the 24th International Symposium on Implementation and Application of Functional Languages*, Lecture Notes in Computer Science, pages 122–139, Oxford, England, 2013. Springer-Verlag.

[6] M. Fluet. A type- and control-flow analysis for System F. Technical report, Rochester Institute of Technology, February 2013. `https://ritdml.rit.edu/handle/1850/15920`.

[7] K. L. S. Gasser, F. Nielson, and H. R. Nielson. Systematic realisation of control flow analyses for CML. In M. Tofte, editor, *ICFP'97: Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming*, pages 38–51, Amsterdam, The Netherlands, June 1997.

[8] F. Gecseg and M. Steinby. *Tree Automata*. Akademiai Kiado, Budapest, Hungary, 1984.

[9] N. Heintze. Set-based program analysis of ML programs. In C. L. Talcott, editor, *LFP'94: Proceedings of the 1994 ACM Conference on Lisp and Functional Programming*, pages 306–317, Orlando, Florida, June 1994.

[10] N. Heintze and D. McAllester. On the cubic bottleneck in subtyping and flow analysis. In G. Winskel, editor, *Proceedings of the 12th Annual IEEE Symposium on Logic in Computer Science (LICS '97)*, pages 342–351, Warsaw, Poland, June 1997.

[11] S. Jagannathan and S. Weeks. A unified treatment of flow analysis in higher-order languages. In P. Lee, editor, *POPL'95: Proceedings of the 22nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 393–407, San Francisco, California, Jan. 1995.

[12] J. Midtgaard and D. V. Horn. Subcubic control flow analysis algorithms. Computer Science Research Report 125, Roskilde University, Roskilde, Denmark, May 2009. Revised version to appear in Higher-Order and Symbolic Computation.

[13] P. Mishra and U. S. Reddy. Declaration-free type checking. In M. S. Van Deusen, Z. Galil, and B. K. Reid, editors, *POPL'85: Proceedings of the Twelfth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 7–21, New Orleans, Louisiana, Jan. 1985. ACM, ACM.

[14] G. Morrisett and R. Harper. Semantics of memory management for polymorphic languages. In A. D. Gordon and A. M. Pitts, editors, *Higher-Order Operational Techniques in Semantics*, Publications of the Newton Institute, pages 175–226. Cambridge University Press, 1998.

[15] F. Nielson and H. R. Nielson. Infinitary control flow analysis: a collecting semantics for closure analysis. In N. D. Jones, editor, *POPL'97: Proceedings of the 24th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 332–345, Paris, France, Jan. 1997.

[16] F. Nielson and H. R. Nielson. Interprocedural control flow analysis. In S. D. Swierstra, editor, *ESOP'99: Proceedings of the Eighth European Symposium on Programming*, volume 1576 of *Lecture Notes in*

[6] Note that without the amortized analysis, processing an $\alpha_1 \approxeq \alpha_2$ element would be $O(l^2) + O(m^2)$ time and the entire algorithm would be $O(l^3 + l^2 m^2 + m^4)$.

*Computer Science*, pages 20–39, Amsterdam, The Netherlands, Mar. 1999. Springer-Verlag.

[17] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.

[18] H. R. Nielson and F. Nielson. Flow logics for constraint based analysis. In K. Koskimies, editor, *CC'98: Proceedings of the 7th International Conference on Compiler Construction*, volume 1383 of *Lecture Notes in Computer Science*, pages 109–127, London, UK, Apr. 1998. Springer-Verlag.

[19] J. Palsberg and M. I. Schwartzbach. Safety analysis versus type inference. *Information and Computation*, 118(1):128–141, 1995.

[20] W. Rytter. Fast recognition of pushdown automaton and context-free languages. *Information and Control*, 67(1-3):12–22, 1985.

[21] O. Shivers. *Control-Flow Analysis of Higher-Order Languages or Taming Lambda*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, May 1991. Technical Report CMU-CS-91-145.

[22] C. A. Stone. Type definitions. In B. C. Pierce, editor, *Advanced Topics in Types and Programming Languages*. The MIT Press, 2005.