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. [4] The type-flow information approximates the type expressions that may instantiate type variables and the control-flow information approximates the λ- and Λ-expressions that may be bound to variables. Furthermore, the two flows are mutually beneficial: control flow determines which Λ-expressions may be applied to which type expressions (and, hence, which type expressions may instantiate which type variables), while type flow filters the λ- and Λ-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 of the analysis 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 $O(l^3 + m^4)$.

1. Introduction

2. Language and Semantics

2.1 Syntax

In IFL'12, we introduced an ANF Intermediary Representation for System F. We give a specification for a modified ANF System F language in Figure 1.

We maintain the same types as before, but we restrict the form of syntactic types in order to remove the recursion found in the previous definition of types. Instead of having type constructors be composed of smaller types (ie. for function and forall types), they must be constructed with type variables. We also introduce DeBruijn indices into the type system to describe any type parameterized by a Λ-abstraction.

In addition to adding a distinction between simple binds (values) and complex binds (applications), we expand upon the definition of expressions to accommodate the changes made to the type system. The syntax is extended to include `let`-bindings for type variables. Additionally, we mandate that all binding occurances of expression variables be annotated with a type variable instead of a type. Similarly, the recursive function variables and λ-parameter variables appearing in simple binds must also have type variable annotations. We changed the specification of type applications so that only type variables may be passed as arguments.

Our motivation for using type variables is to promote type reuse, limiting the pool of possible types to as little a number as possible. As we will see later, having as few types as possible will reduce the overall complexity of the algorithm.

2.2 Scanning Input Program

Figure 2 introduces relations that finds all nested expressions, expression binds, type binds, type variables, and expression variables in a program.

We begin our recursive descent into a program by saying the entire program can be found inside itself. An expression can be found in the program if it is the body of a `let`-binding or if it belongs to a λ- or Λ-abstraction. An expression bind belongs to a program if the program contains a `let` x with that particular expression bind on the right-hand side. Similarly, a type bind belongs to a program if it participates in a `let` α expression that also belongs to the program. All type variables found in any nested `let`α or as parameters to Λ-abstractions belong to the program. The relation behaves similarly with expression variables and `let` x bindings and λ-abstractions. We must also consider that the recursive function variables, f, belong to a program if the encapsulating λ- or Λ-abstraction also belongs to the program.

2.3 Semantics

We assume the usual operational semantics for the ANF System using a CESK machine. More details can be found in [4].

The static semantics will need to be extended to include support for DeBruijn indices. In particular, when a value is used, we need to ensure that all indices present in the type are encapsulated by the proper amount of foralls. We also must mandate that type variables are bound before use, which has been implemented successfully in [12]. Whenever an expression variable is bound to a value, the annotated type variable must map to a type in the current context that is compatible with the type of the value.

It is worth noting that all forms of the analysis and algorithm will hold for a program even if it is untyped. It is only when the program is well typed that the flow-graph based analysis and the algorithm will yield a sound result.

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

$$\begin{array}{rcl}
\mathsf{ResOf}(\cdot) & :: & \mathit{Exp} \to \mathit{ExpVar} \\
\mathsf{ResOf}(\mathtt{let}\ \alpha = \tau\ \mathtt{in}\ e) & = & \mathsf{ResOf}(e) \\
\mathsf{ResOf}(\mathtt{let}\ x{:}\alpha_x = b\ \mathtt{in}\ e) & = & \mathsf{ResOf}(e) \\
\mathsf{ResOf}(x) & = & x
\end{array}$$

$$\begin{array}{rcl}
\mathsf{TyOf}(\cdot) & :: & \mathit{ExpBnd}_{\mathsf{s}} \to \mathit{TyVar} \\
\mathsf{TyOf}(\mu f{:}\alpha_f\,.\,\lambda z{:}\alpha_z\,.\,e_b) & = & \alpha_f \\
\mathsf{TyOf}(\mu f{:}\alpha_f\,.\,\Lambda\beta\,.\,e_b) & = & \alpha_f
\end{array}$$

Figure 1. Syntax of ANF System F

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

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

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

$\boxed{b \preceq_{\mathit{ExpBnd}} P}$

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

$\boxed{\tau \preceq_{\mathit{TyBnd}} P}$

$$\frac{\mathtt{let}\ \alpha = \tau\ \mathtt{in}\ e \preceq_{\mathit{Exp}} P}{\tau \preceq_{\mathit{TyBnd}} P}$$

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

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

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

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

Figure 2. Sub-term Relations

3. Specification-Based Formulation of TCFA

The specification formulation is safe under an abstract type environment, which keeps track of type variables and the possibly many types that may be bound to the variables, and also an abstract value environment, which behaves the same way with expression variables and binders. In order to check for type compatibility, we make use of an abstract type environment to recursively replace all type variables with any type found in the environment's entry for the variable. We say that a type is closed when there are no longer any type variables in the expanded type. Two type variables, α_1 and α_2, are compatible under an abstract type environment if it is possible to derive the same closed type from both α_1 and α_2.

An abstract type environment, $\hat{\phi}$, and an abstract value environment, $\hat{\rho}$, safely approximate an expression by an inductive analysis of the expression.

If the expression is a `let`-α expression, we must make sure that the τ being bound in the syntax appears in the possible bindings for α as described by $\hat{\phi}$ and that the remaining expressions are also safe under the same $\hat{\phi}$ and $\hat{\rho}$.

If the expression is a `let`-x expression and the binder is a simple bind, we read the type off of x and the binder. If the two are compatible we need to make sure that the binder appears in the possible bindings of x in $\hat{\rho}$. Additionally, we need to push the analysis through the body of the binder and the remaining expressions.

Otherwise, if a complex bind appears on the right side of a `let`-x expression, we need to iterate through the possible λ-abstractions recorded in $\hat{\rho}$ if the binder is a value application. For every possible argument in the entry for x_a, if the type of the argument is compatible with the type of the formal parameter under $\hat{\phi}$, we need to assert that the argument also appears in the entry for the parameter. Likewise, for all values returned by the abstraction, if the type of the value is compatible with the type of the bound variable, it must appear in the entry for the bound variable in $\hat{\phi}$.

Similarly, if a type application is being performed, we iterate through all possible Λ-abstractions that could be bound to the function variable. We assert that all types potentially bound to the type argument must appear in the description of the type parameter in $\hat{\phi}$. As with value application, any type-compatible results returned from the abstraction must be present in $\hat{\phi}$'s entry for the bound variable.

Finally, in the case that the expression is a simple variable, we assume that the analysis is sound.

3.1 Soundness, Decidability, and Computability

Our previous work [4, 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 $\hat{\phi}; \hat{\rho} \models_{\mathsf{S}} \alpha_1 \approx \alpha_2$ is derivable whenever there is a common closed type at which both α_1 and α_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 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 free, we cannot simply enumerate the (potentially infinite sets of) closed types θ_1 and θ_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, 3, 6]. 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 θ from the starting non-terminal α in the regular-tree grammar $\hat{\phi}$ and the judgment $\hat{\phi} \models_{\mathsf{S}} \alpha_1 \approx \alpha_2$ is derivable iff languages generated by taking α_1 and α_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 [6, 9], 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 α/τ and x/b_{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 $\hat{\rho}(x_f)$ and $\hat{\rho}(\mathsf{ResOf}(e_b))$ ($O(n)$), loops over all of the elements of $\hat{\rho}(x_a)$ ($O(n)$), 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 size of 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: $O(n^{13}) = (O(n^2) + O(n^2)) \times (O(n) \times O(n) \times O(n) \times (O(n^4) + O(n^8)))$.

[1] See Sections 5.2.1 and 5.2.2 for more discussion of assumptions about the representation of the input program and data structures and operations.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Figure 3. Specification-Based Formulation of TCFA

4. Flow-Graph-Based Formulation of TCFA

We give judgments pertaining to the behavior of the Flow-Graph analysis in Figure 4. These judgments are analogous to building a flow-graph of a program P where the edges are the definite flows between types, type variables, simple binders and expression variables. There is also a conditional edge between a binder and an expression variable that is guarded by the type compatibility of the two in P. Once it is learned that the types are indeed compatible, the edge is activated and belongs to the final result. After the graph is constructed, Typed- and Control-Flow Analysis is reduced to Graph Reachability across the flow-graph.

4.1 Flow-Graph Analysis

We define a series of judgments to perform a program parameterized flow-graph analysis, dependent upon the Sub-part relation in Figure 2.

The analysis uses a program based type compatibiliity, defined mutually with type variable compatibility. Any two DeBruijn indices are compatible if they are the same index number. Otherwise, for functions and forall types that have type variable components, the two types are compatible if the corresponding components are compatible under the program. Two type variables, α_1 and α_2 are compatible in a program if there is a τ_1 and τ_2 flowing to α_1 and α_2, respectively, such that τ_1 is compatible with τ_2 in the program.

The flow-graph constructed by the analysis performed on a program, P, consists of both type-flow information and value-flow information.

We know that a type, τ, flows to a given type variable, α_1, when one of two cases is true: there is an explicit `let`-α binding in the program involving τ and α_1; if not, there is some α_2 and we have learned that α_1 flows to α_2. Such an edge between type variables is constructed if there exists a type application in P where α_2 is the formal type parameter and α_1 is the supplied type argument.

For a simple binder, b_s to flow to an expression variable, x, we need to know two pieces of information. We must have already seen that the binder could possibly flow to the variable and also that the type of the binder is compatible with the type of the variable. Initially, for all λ- and Λ-abstractions, we assert that the abstraction flows to its own recursive function variable. Other conditional flow edges are added whenever the program contains a `let`-x expression using b_s and x and whenever we learn of a transitive variable flow. Whenever we see an expression application and we already know that a λ-abstraction flows to the function variable, we add a flow edge between the argument variable and the formal parameter variable dependent upon the type of the parameter and also between the return of the function and the variable being bound by the `let`-x expression. The return of Λ-abstraction also flows to a `let`-x bound variable dependent upon the bound variable's type if the binder is a type application and the Λ-abstraction flows the function variable.

4.2 Soundness

From our flow-based analysis, we prove that if we have a $\hat{\phi}$ and $\hat{\rho}$ that are "flow-induced" from a well-typed program, then they soundly model the program. Before we begin, we introduce the following lemma:

Lemma 1. *For all P, $\hat{\phi}$, if*
$\forall \alpha, \tau \,.\, \tau \in \hat{\phi}(\alpha) \Leftrightarrow P \vDash_{\mathsf{G}} \tau \rightarrowtail \alpha$
then $P \vDash_{\mathsf{G}} \alpha_1 \approxeq \alpha_2 \Leftrightarrow \hat{\phi} \vDash_{\mathsf{S}} \alpha_1 \approxeq \alpha_2$

We assert that the lemma is true without an accompanying proof. By inspection, the forward case must be true because the derivation of $P \vDash_{\mathsf{G}} \alpha_1 \approxeq \alpha_2$ tells us that there exists a τ_1 flowing to α_1 and a τ_2 flowing to α_2 such that $P \vDash_{\mathsf{G}} \tau_1 \approxeq \tau_2$. We assume that this can be translated to $\hat{\phi} \vDash_{\mathsf{S}} \tau_1 \Rrightarrow \theta$ and $\hat{\phi} \vDash_{\mathsf{S}} \tau_2 \Rrightarrow \theta$, thus allowing us to derive our conclusion. The backwards case is a reflection of the logic in the forward case.

Theorem 1. *For all P, $\hat{\phi}$, and $\hat{\rho}$, if*

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

then $\hat{\phi}; \hat{\rho} \vDash_{\mathsf{s}} P$.

Proof. By induction on P. The interesting cases are when P is a `let`-x bound to a λ-abstraction and also when P is a `let`-x bound to a type application.

Case P of `let` $x{:}\alpha_x$ = $\mu f{:}\alpha_f\,.\,\lambda z{:}\alpha_z\,.\,e_b$ `in` e: Knowing that P is unconditionally a sub-term of itself, we can deduce that `let` $x{:}\alpha_x$ = $\mu f{:}\alpha_f\,.\,\lambda z{:}\alpha_z\,.\,e_b$ `in` e is a subterm of P. Thus we can build a conditional edge in the flow graph from the abstraction to x. Since P is well-typed, we know that $P \vDash_{\mathsf{G}} \alpha_f \approxeq \alpha_x$ and can thus derive $P \vDash_{\mathsf{G}} \mu f{:}\alpha_f\,.\,\lambda z{:}\alpha_z\,.\,e_b \rightarrowtail x$ and $\hat{\phi} \vDash_{\mathsf{S}} \alpha_f \approxeq \alpha_x$. Our assumption thus gives us $b_{\mathsf{s}} \in \hat{\rho}(x)$ from $P \vDash_{\mathsf{G}} \mu f{:}\alpha_f\,.\,\lambda z{:}\alpha_z\,.\,e_b \rightarrowtail x$. We have $\hat{\phi}; \hat{\rho} \vDash_{\mathsf{S}} e_b$ and $\hat{\phi}; \hat{\rho} \vDash_{\mathsf{S}} e$ by our inductive hypothesis, and thus we have our derivation for $\hat{\phi}; \hat{\rho} \vDash_{\mathsf{S}}$ `let` $x{:}\alpha_x$ = $\mu f{:}\alpha_f\,.\,\lambda z{:}\alpha_z\,.\,e_b$ `in` e.

Case P of `let` $x{:}\alpha_x$ = x_f $[\alpha_a]$ `in` e: We again start by asserting that α_x = x_f $[\alpha_a]$ `in` e is a subterm of P. If there is a $\mu f{:}\alpha_f\,.\,\Lambda\beta\,.\,e_b$ that flows to x_f, then we can create a conditional edge from the result of the Λ-abstraction to x and an unconditional edge from α_a and β. The conditional edge only allows binders that are type compatible in the flow specification to flow transitively from the result variable to x. By our Lemma we can show that the binders that flow to x are type compatible with α_x and our assumption tells us that those binders are in the entry for x in $\hat{\rho}$. The graph also tells us that any τ flowing to α_a flows to β, which we can use to show that τ is in the entry for β in $\hat{\phi}$ from our assumption. Our inductive hypothesis allows us to show that $\hat{\phi}$ and $\hat{\rho}$ soundly model e, and we thus derive that $\hat{\phi}; \hat{\rho} \vDash_{\mathsf{G}}$ `let` $x{:}\alpha_x$ = x_f $[\alpha_a]$ `in` e. □

$\boxed{P \vDash_{\mathsf{G}} \tau_1 \mathrel{≋} \tau_2}$

$$\textsc{TyCompatArrow}\quad \frac{P \vDash_{\mathsf{G}} \alpha_{a1} \mathrel{≋} \alpha_{a2} \qquad P \vDash_{\mathsf{G}} \alpha_{b1} \mathrel{≋} \alpha_{b2}}{P \vDash_{\mathsf{G}} \alpha_{a1} \rightarrow \alpha_{b1} \mathrel{≋} \alpha_{a2} \rightarrow \alpha_{b2}}$$

$$\textsc{TyCompatForAll}\quad \frac{P \vDash_{\mathsf{G}} \alpha_{b1} \mathrel{≋} \alpha_{b2}}{P \vDash_{\mathsf{G}} \forall.\ \alpha_{b1} \mathrel{≋} \forall.\ \alpha_{b2}}$$

$$\textsc{TyCompatTyIdx}\quad \frac{}{P \vDash_{\mathsf{G}} \#n \mathrel{≋} \#n}$$

$\boxed{P \vDash_{\mathsf{G}} \alpha_1 \mathrel{≋} \alpha_2}$

$$\textsc{TyVarCompat}\quad \frac{P \vDash_{\mathsf{G}} \tau_1 \rightarrowtail \alpha_1 \qquad P \vDash_{\mathsf{G}} \tau_2 \rightarrowtail \alpha_2 \qquad P \vDash_{\mathsf{G}} \tau_1 \mathrel{≋} \tau_2}{P \vDash_{\mathsf{G}} \alpha_1 \mathrel{≋} \alpha_2}$$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Figure 4. Flow-Graph-Based Formulation of TCFA

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. The algorithm returns a result set R whose elements correspond to judgements 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 $\alpha_1 \approxeq \alpha_2$ to a queue of conclusions that may be added to R when $\alpha_1 \approxeq \alpha_2$ is proved to be derivable; the queues in T will serve as "banks" holding credit for the amortized complexity analysis.

5.1 Commentary

5.1.1 Initialization Phase

The first initialization phase (lines 5–18) adds to R and W all elements of the form $b_s \rightarrowtail^? x : \alpha_x$ that are derivable using the rules LetExpBnd$_s$, $\mu\lambda$ExpBnd$_s$, and $\mu\Lambda$ExpBnd$_s$, rules whose conclusion follows directly from the input program. 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$_s$.

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 τ_1 and τ_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 τ_1 and τ_2 will be known to be compatible when two pairs of type variables are known to be compatible; when $\alpha_{a1} \approxeq \alpha_{a2}$ 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 τ_1 and τ_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 τ_1 and τ_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.

5.1.2 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$_s$ need be inspected. For each $b_s \rightarrowtail x$ that is already known to be derivable, then TransExpBnd$_s$ may derive $b_s \rightarrowtail^? y : \alpha_y$ and it is added to R and W.

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

When the work-queue element is of the form $b_s \rightarrowtail x$ (lines 69–104), the rules TransExpBnd$_s$, 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$_s$ may derive $b_s \rightarrowtail^? y : \alpha_y$ and it is added to R and W. When b_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_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 `[`α_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 $\pi \rightarrowtail \beta$ that is already known to be derivable, if τ and π are already known to be compatible, then TyVarCompat may derive $\alpha \approxeq \beta$ 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$_s$ and TyVarCompat need to be inspected. Each time that $b_s \rightarrowtail x : \alpha_x$ was known to be derivable but $\mathsf{TyOf}(b_s)$ and α_x were not yet known to be compatible (preventing TyVarCompatExpBnd$_s$ from deriving $b_s \rightarrowtail x$), an element of the form $b_s \rightarrowtail x$ was added to the queue given by $\mathsf{Map.get}(T, \mathsf{TyOf}(b_s) \approxeq \alpha_x)$ (see line 66); hence, processing these elements of the queue will add each $b_s \rightarrowtail x$ that may be derived by TyVarCompatExpBnd$_s$ to R and W. For each pair of type binds τ_1 and τ_2 whose compatibility depends upon the compatibility of α_1 and α_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 τ_1 and τ_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.

5.1.3 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.

```
Ensure: $\forall \tau_1 \preceq_{TyBnd} P \,.\, \forall \tau_2 \preceq_{TyBnd} P \,.\, \tau_1 \approxeq \tau_2 \in R \Leftrightarrow P \vDash_{\mathsf{G}} \tau_1 \approxeq \tau_2$
Ensure: $\alpha_1 \approxeq \alpha_2 \in R \Leftrightarrow P \vDash_{\mathsf{G}} \alpha_1 \approxeq \alpha_2$
Ensure: $\tau \rightarrowtail \alpha \in R \Leftrightarrow P \vDash_{\mathsf{G}} \tau \rightarrowtail \alpha$
Ensure: $\alpha \rightarrowtail \beta \in R \Leftrightarrow P \vDash_{\mathsf{G}} \alpha \rightarrowtail \beta$
Ensure: $b_{\mathsf{s}} \rightarrowtail x \in R \Leftrightarrow P \vDash_{\mathsf{G}} b_{\mathsf{s}} \rightarrowtail x$
Ensure: $b_{\mathsf{s}} \rightarrowtail^{?} x : \alpha_x \in R \Leftrightarrow P \vDash_{\mathsf{G}} b_{\mathsf{s}} \rightarrowtail^{?} x : \alpha_x$
Ensure: $x \rightarrowtail y : \alpha_y \in R \Leftrightarrow P \vDash_{\mathsf{G}} x \rightarrowtail y : \alpha_y$

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

    for all let $x{:}\alpha_x$ = $b_{\mathsf{s}}$ in $e \preceq_{Exp} P$ do      ▷ $O(l) = O(l) \times O(1)$
        Set.insert($R, b_{\mathsf{s}} \rightarrowtail^{?} x : \alpha_x$)
        Queue.push($W, b_{\mathsf{s}} \rightarrowtail^{?} x : \alpha_x$)
        match $b_{\mathsf{s}}$ with
            case $\mu f{:}\alpha_f \,.\, \lambda z{:}\alpha_z \,.\, e_b$ do
                Set.insert($R, b_{\mathsf{s}} \rightarrowtail^{?} f : \alpha_f$)
                Queue.push($W, b_{\mathsf{s}} \rightarrowtail^{?} f : \alpha_f$)
            end case
            case $\mu f{:}\alpha_f \,.\, \Lambda\beta \,.\, e_b$ do
                Set.insert($R, b_{\mathsf{s}} \rightarrowtail^{?} f : \alpha_f$)
                Queue.push($W, b_{\mathsf{s}} \rightarrowtail^{?} f : \alpha_f$)
            end case
        end match
    end for

    for all let $\alpha$ = $\tau$ in $e \preceq_{Exp} P$ do              ▷ $O(m) = O(m) \times O(1)$
        Set.insert($R, \tau \rightarrowtail \alpha$)
        Queue.push($W, \tau \rightarrowtail \alpha$)
    end for

    for all $\alpha_1 \preceq_{TyVar} P$ do                     ▷ $O(m^2) = O(m) \times O(m)$
        for all $\alpha_2 \preceq_{TyVar} P$ do                       ▷ $O(m) = O(m) \times O(1)$
            Map.set($T, \alpha_1 \approxeq \alpha_2$, Queue.newEmpty())
        end for
    end for

    for all $\tau_1 \preceq_{TyBnd} P$ do                      ▷ $O(m^2) = O(m) \times O(m)$
        for all $\tau_2 \preceq_{TyBnd} P$ do                        ▷ $O(m) = O(m) \times O(1)$
            match $\langle \tau_1, \tau_2 \rangle$ with
                case $\langle \alpha_{a1} \to \alpha_{b1}, \alpha_{a2} \to \alpha_{b2} \rangle$ do
                    $c \leftarrow$ Counter.new(2)
                    Queue.push(Map.get($T, \alpha_{a1} \approxeq \alpha_{a2}$), $\langle c, \tau_1 \approxeq \tau_2 \rangle$)      ▷ $O(1)$ credit into $T[\alpha_{a1} \approxeq \alpha_{a2}]$ queue
                    Queue.push(Map.get($T, \alpha_{b1} \approxeq \alpha_{b2}$), $\langle c, \tau_1 \approxeq \tau_2 \rangle$)      ▷ $O(1)$ credit into $T[\alpha_{b1} \approxeq \alpha_{b2}]$ queue
                end case
                case $\langle \forall.\ \alpha_{b1}, \forall.\ \alpha_{b2} \rangle$ do
                    $c \leftarrow$ Counter.new(1)
                    Queue.push(Map.get($T, \alpha_{b1} \approxeq \alpha_{b2}$), $\langle c, \tau_1 \approxeq \tau_2 \rangle$)      ▷ $O(1)$ credit into $T[\alpha_{b1} \approxeq \alpha_{b2}]$ queue
                end case
                case $\langle \#n, \#m \rangle$ do
                    if $n = m$ then
                        Set.insert($R, \tau_1 \approxeq \tau_2$)
                        Queue.push($W, \tau_1 \approxeq \tau_2$)
                    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 $b_{\mathsf{s}} \rightarrowtail x \in R$ do                      $\triangleright\ O(l) = O(l) \times O(1)$
                if $b_{\mathsf{s}} \rightarrowtail^? y : \alpha_y \notin R$ then
                    Set.insert$(R, b_{\mathsf{s}} \rightarrowtail^? y : \alpha_y)$
                    Queue.push$(W, b_{\mathsf{s}} \rightarrowtail^? y : \alpha_y)$
                end if
            end for
        end case

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

        case $b_{\mathsf{s}} \rightarrowtail x$ 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 $b_{\mathsf{s}} \rightarrowtail^? y : \alpha_y \notin R$ then
                    Set.insert$(R, b_{\mathsf{s}} \rightarrowtail^? y : \alpha_y)$
                    Queue.push$(W, b_{\mathsf{s}} \rightarrowtail^? y : \alpha_y)$
                end if
            end for
            match $b_{\mathsf{s}}$ 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**                              ▷ $O(m^4) = O(m^2) \times O(m^2)$
            **for all** $\alpha \rightarrowtail \beta \in R$ **do**                     ▷ $O(m) = O(m) \times O(1)$
                **if** $\tau \rightarrowtail \beta \notin R$ **then**
                    Set.insert$(R, \tau \rightarrowtail \beta)$
                    Queue.push$(W, \tau \rightarrowtail \beta)$
                **end if**
            **end for**
            **for all** $\pi \rightarrowtail \beta \in R$ **do**                     ▷ $O(m^2) = O(m^2) \times O(1)$
                **if** $\tau \approxeq \pi \in R$ **then**
                    **if** $\alpha \approxeq \beta \notin R$ **then**
                        Set.insert$(R, \alpha \approxeq \beta)$
                        Queue.push$(W, \alpha \approxeq \beta)$
                    **end if**
                **end if**
            **end for**
        **end case**

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

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

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

5.2 Complexity

5.2.1 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`-, μ-, and λ-bound expression variables and all `let`-, and Λ-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.[2] Given the assumption that all `let`-, μ-, and λ-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 μ-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.[3] We do not assume that each type bind in the program is 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.[4]

5.2.2 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 judgements 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_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_s instantiating x or for fixed x instantiating b_s can be implemented as a linear scan of an array column or array row.

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.

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.

5.2.3 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 α_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 $b_\mathsf{s} \rightarrowtail^? x : \alpha_x$ and processing a $b_\mathsf{s} \rightarrowtail^? x : \alpha_x$ 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 $\mathtt{let}\, x_r\!:\!\alpha_r = x\; x_a\; \mathtt{in}\; e \preceq_{Exp} P$ and to find all $\mathtt{let}\, x_r\!:\!\alpha_r = x\; \mathtt{[}\alpha_a\mathtt{]}\; \mathtt{in}\; e \preceq_{Exp} P$. 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, 7, 10, 11], though recently improved to $O(n^2 \log n)$ [8].

5.2.4 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.

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

[3] 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 numbering of all simple expression binds in the program, similar to the labeling found in textbook presentations of CFA [10].

[4] 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.

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 `int` $\rightarrow$ `int` functions, all of which can share the same (top-level) `let` $\alpha_{\mathtt{i}}$ = `int in let` $\alpha_{\mathtt{i}\rightarrow\mathtt{i}}$ = $\alpha_{\mathtt{i}} \rightarrow \alpha_{\mathtt{i}}$ `in` ... 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 $O(l)$ are $O(n)$. We further assume an $O(n)$ preprocessing step that provides an enumeration of all `let` $x{:}\alpha_x$ = b `in` $e \preceq_{Exp} P$ in $O(l)$ time and an enumeration of all `let` $\alpha = \tau$ `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.

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.[5] 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 $O(l^3 + m^4)$.

[5] 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 $O(l^3 + l^2 m^2 + m^4)$.

6. Conclusion

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] P. Cousot and R. Cousot. Formal language, grammar and set-constraint-based program analysis by abstract interpretation. In S. Peyton Jones, editor, *FPCA'95: Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture*, pages 170–181, La Jolla, California, June 1995.

[4] 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, Oxford, England, 2013. Springer-Verlag. To appear.

[5] 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`.

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

[7] 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.

[8] 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.

[9] 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.

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

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

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