Computer Science

Control-Flow Analysis and Type Systems

NEVIN HEINTZE

December 1994
CMU-CS-94-227

Control-Flow Analysis and Type Systems

NEVIN HEINTZE

December 1994

CMU-CS-94-227

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

(Also appears as Fox Memorandum CMU-CS-FOX-94-09.)

Abstract

We establish a series of equivalences between type systems and control-flow analyses. Specifically, we take four type systems from the literature (involving simple types, subtypes and recursion) and conservatively enrich them to reason about control-flow information. Similarly, we take four standard control-flow systems and conservatively enrich them to reason about type consistency. Our main result is that for each type system, there is a control-flow system with equivalent reasoning power. In essence, type systems and control-flow analysis can be viewed as complementary approaches for addressing questions of type consistency and control-flow.

This work was sponsored by the Advanced Research Projects Agency, CSTO, under the title "The Fox Project: Advanced Development of Systems Software", ARPA Order No. 8313, issued by ESD/AVS under Contract No. F19628-91-C-0168.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of ARPA or the U.S. Government.

Keywords: types, type systems, subtypes, recursive types, program analysis, control-flow analysis.

1 Introduction

A central concept in compiler optimization and code generation is a graph of how control can flow from one program point to another. Such a graph identifies block and loop structure in a program, and this is the starting point for a large number of optimizations. In many languages, this *control-flow* graph can be directly constructed from a program because information about flow of control from one point to another is explicit. However, in a language with higher-order functions, the flow of control from one point to another is not readily apparent from program text because a function can be passed around as data and subsequently called from anywhere in the program. If this happens sufficiently often, then the lack of control-flow information can significantly limit compiler performance. To addresses this issue, systems for *control-flow analysis* [3, 4, 10, 2, 8] have been developed. The purpose of control-flow analysis is to compute an approximation of the possible functions that can be called from each program point.

In contrast, the purpose of a type inference system is to derive invariants about the potential bindings of variables in a program. However, in the context of higher-order functions, functions can be data and therefore can be bound to variables. Hence, there is an intuitive connection between reasoning about function (in control-flow analysis) and reasoning about the data values to which variables may be bound (in type inference). In fact it is possible to extend type systems to perform control-flow analysis [11, 12], and conversely, extend control-flow systems to perform type analysis [7, 8]. There are also informal connections between the kinds of reasoning done by type and control-flow systems. For example, consider the essence of the application rule of a subtype system:

if M has type $\tau_1 \rightarrow \tau_2$ and N has type τ_1' such that $\tau_1' \subseteq \tau_1$, then $(M\ N)$ has type τ_2

Contrast this with a control-flow view of $(M\ N)$: if the results of M are described by τ_1' and the functions that flow to M are such that, when called on arguments described by some τ_1 such that τ_1 includes τ_1', they return results that are described by τ_2, then it is correct to use τ_2 to describe the results returned by $(M\ N)$.

Despite these informal similarities, type systems and control-flow system possess quite different structure. Type systems are typically specified using deductive systems. Such systems associate a type with each program expression and this type completely captures the type system's view of the behaviour of the expression. These systems are compositional and often have direct application to modular analysis and separate compilation. In contrast, control-flow analysis systems typically reason globally using finite sets of "function labels". Such presentations are rarely compositional, but they are often more suggestive of feasible implementation strategies.

In this paper we give a unifying presentation and systematic comparison of type systems (extended to reason about control-flow) and control-flow systems (extended to reason about type consistency). The type systems we consider are simple types, partial types, simple recursive types, and recursive subtypes. Each system is extended in a generic way so that the types carry control-flow information. In each case, the extension preserves the structure of the underlying type system: the same sets of terms are typable under the extended system, and in fact we can fairly easily translate between derivations in the extended system and the original system. The control-flow systems we consider are "control-flow-0" (a standard system in the control-flow literature), a system based on unification, and two derived systems that exclude recursion. In each case, these systems are modified to reason about type consistency in such a way that their ability to reason about control-flow is not changed. Our main result is that the families of types systems and control-flow systems are equivalent in the following sense: for each type system there is a control-flow system such that (i) both systems compute the same control-flow information, and (ii) the type consistency components of both systems coincide.

Two main technical issues arise in these equivalence proofs. First, type systems and control-flow systems compute over different representations. For the type systems used in this paper, types can be viewed as (possibly infinite) trees that are decorated with control-flow information. Each type system associates one of these trees with each subexpression of the program. In contrast, the control-flow systems used in this paper compute using finite sets (which essentially contain labels of functions). Each control-flow system annotates program subexpressions with control-flow sets. A key part of the proof involves mapping from one representation to the other. While mapping from a type to a control-flow annotation can be achieved by just suppressing type structure, the mapping in the reverse direction is more complex: it involves reconstructing a type tree using the annotations of the entire program. Thus, type information is not explicit in the control-flow annotations, but rather it is distributed over the entire program. In essence, the control-flow annotations of the program collectively represent encodings of (potentially infinite) trees. The second difficulty is that the type and control-flow systems employ different notions of "consistency". For example consider an application $M\ N$. In a control-flow system, consistency involves considering all functions that can flow to M and reasoning about the annotations of these functions. In contrast, a type system would reason about this application by considering just the types associated with M and N. Any information about the functions that can flow to M is already represented in the type associated with M. In short, types involve only local reasoning; control-flow involves highly non-local reasoning.

The equivalences established in the paper reveal much about the underlying structure of control-flow and type systems. For example, adding recursive types to a type system corresponds exactly to allowing cycles in the control-flow graph computed by a control-flow analysis. Similarly, adding subtypes to a type system corresponds exactly to replacing equalities by set containment in the consistency conditions of a control-flow system. Our results also provide a basis for exploiting the advantages of both the type and control-flow view. From the point of view of implementing an analysis, it is usually easier to start from a control-flow system. Conversely, for the purposes of designing a modular analysis system, type systems have an advantage since they are usually presented compositionally (we remark that one of the original motivations for this paper was the development of modular analysis). In addition, certain results are sometimes easier to prove in one system that the other (e.g. *correctness* becomes a subject reduction result in the type system, and such results are often easy to establish).

Related Work

Previous works [7, 8] have established that the typing component of the control-flow-0 system (when extended to reason about type consistency) is more accurate than simple types and partial types. (We remark that the equivalences established in this paper can be viewed as an alterative explanation of these results. Moreover, on combining our equivalence results with obvious relationships between the accuracy of type systems, we obtain a number of additional accuracy relationships.) Very recently, Palsberg and O'Keefe [9] have independently obtained the equivalence of the typing component of control-flow-0 and the recursive subtype system of Amadio and Cardell [1]. We also prove this result, in addition to analogous results for three other control-flow systems. No previous works have established relationships between the control-flow component of (suitably extended) types systems and control-flow systems.

2 Preliminaries

We define a variant of the λ-calculus with labeled abstractions. The reason for the labels is that they allow us to talk about program "control-flow". We remark that the language is suggestive of an intermediate language that might be produced by a compiler front end; it is not intended as a language for writing programs. Let LABELS be a countably infinite set of labels, and define terms e by:

$$e ::= x \mid \lambda^l x.e \mid e_1\, e_2 \mid O \mid Succ\ e.$$

where $l \in$ LABELS. The intention is that labels distinguish different occurrences of abstractions. Free and bound occurrences of variables in a term are defined in the usual way. A term is *closed* if all occurrences of variables are bound. A *program* is a closed term in which each abstraction has a unique label. Evaluation for this language is based on β-reduction:

$$(\lambda^l x.e)\ e' \longrightarrow_\beta e[x/e']$$

where $e[x/e']$ denotes the result replacing of e' by x in e, after appropriate renaming of bound variables. Note that this process preserves labels on abstractions: e.g. $(\lambda^l x.(x\ x))\ (\lambda^{l'} x.(x\ x))$ $\longrightarrow_\beta (\lambda^{l'} x.(x\ x))\ (\lambda^{l'} x.(x\ x))$.

There are no restrictions on the reduction order: the β-reduction can be applied at any subterm at any time. However, we do require that whenever an application $(e_1\ e_2)$ appears in a term, then e_1 cannot be 0 and if e_1 is *Succ* then e_2 must be 0 or of the form $(Succ\ \cdots)$. If these conditions are not met, then program execution terminates with an error. The possibility of this behaviour leads to a very natural typing problem that forms a key part of this paper. Although this typing problem arises precisely because 0 and *Succ* have been added to the language, we remark that our results are largely independent of the specific choice of constants, and could be generalized to other settings.

The language also gives rise to a natural control-flow problem: given a program e_0, construct a set of labels L such that if e_0 evaluates to a lambda expression, the label of this expression will be contained in L. More generally, we are interested in the sets of labels for subexpressions as well.

To formalize type and control-flow systems, we shall introduce various systems for "correctly" annotating a term (and all of its subterms) with either type or control-flow assertions. In general, these annotated terms will have the following form: if $\mathcal{A}$ is a countable set of annotations, then the *$\mathcal{A}$-annotated terms* z (or just *annotated terms*) are defined by:

$$z ::= x\!:\!A \mid (\lambda^l x.z)\!:\!A \mid (z_1\ z_2)\!:\!A \mid O\!:\!A \mid (Succ\ e)\!:\!A$$

where $A \in \mathcal{A}$. Given an annotated term z, we define $annot(z)$ to be the outermost annotation of z. That is, $annot(exp : A)$ is A. We also define $|z|$ to be the (standard) term resulting from stripping all annotations from z. Note that if $\mathcal{A}$ is a collection of types, then an annotated term is essentially an explicitly typed term; such a term carries with it much of the structure of a typing derivation for $|z|$.

3 Type Systems for Control-Flow Analysis

3.1 Simple Types

We begin by describing a type system for control-flow analysis based on simple types [6]. Consider a system of simple types where types τ are constructed from the single base type Int and $\rightarrow$ as follows:

$$\tau ::= Int \mid \tau_1 \rightarrow \tau_2$$

The usual formulation of simple types uses a deductive system involving judgements of the form $\Gamma \vdash e\!:\!\tau$. Such a judgement indicates that, in the *context* Γ (a partial mapping from variables to types), e has type τ. However, for this paper, we shall need to simultaneously associate types with all of the subexpression of a term. Hence, we shall formulate our type system using *type-annotated terms*. Using such terms, the standard simple type system can be presented as:

$$\Gamma \vdash x\!:\!\tau \quad \text{(if } x : \tau \text{ appears in } \Gamma\text{)} \qquad \text{(VAR)}$$

$$\frac{\Gamma, x\!:\!\tau_1 \vdash z}{\Gamma \vdash \lambda^l x.z\!:\!\tau_1 \rightarrow \tau_2} \quad \text{(if } annot(z) \text{ is } \tau_2\text{)} \qquad \text{(ABS)}$$

$$\frac{\Gamma \vdash z \qquad \Gamma \vdash z'}{\Gamma \vdash (z\ z')\!:\!\tau_2} \quad \text{(if } annot(z) \text{ is } \tau_1 \rightarrow \tau_2 \text{ and } annot(z') \text{ is } \tau_1\text{)} \qquad \text{(APP)}$$

$$\Gamma \vdash 0\!:\!Int \qquad \text{(INT)}$$

$$\frac{\Gamma \vdash z}{\Gamma \vdash (Succ\ z)\!:\!Int} \quad \text{(if } annot(z) \text{ is } Int\text{)} \qquad \text{(SUCC)}$$

$$\Gamma \vdash x : \tau \quad (\text{if } x : \tau \text{ appears in } \Gamma) \tag{VAR}$$

$$\frac{\Gamma, x:\tau_1 \vdash z}{\Gamma \vdash \lambda^l x.z : \langle L, \tau_1 \to \tau_2 \rangle} \quad (\text{if } l \in L \text{ and } annot(z) \text{ is } \tau_2) \tag{ABS}$$

$$\frac{\Gamma \vdash z \qquad \Gamma \vdash z'}{\Gamma \vdash (z\, z') : \tau_2} \quad (\text{if } annot(z) \text{ is } \langle L, \tau_1 \to \tau_2 \rangle \text{ and } annot(z') \text{ is } \tau_1 \,) \tag{APP}$$

$$\Gamma \vdash 0 : \langle L, Int \rangle \tag{INT}$$

$$\frac{\Gamma \vdash z}{\Gamma \vdash (Succ\ z) : \langle L, Int \rangle} \quad (\text{if } annot(z) \text{ is } \langle \emptyset, Int \rangle) \tag{SUCC}$$

Figure 1: Inference rules for control-flow type system based on simple types.

We emphasize that this is just a reorganization of the standard formulation of simple types. We now describe a uniform method for enriching a type system to perform control-flow analysis. The first step is to enrich types so that they carry information about labels of abstractions. Corresponding to simple types, we define *control-flow types* by:

$$\tau ::= \langle L, Int \rangle \mid \langle L, \tau_1 \to \tau_2 \rangle$$

where L ranges over finite subsets of LABELS. Note that the set of labels L in $\langle L, Int \rangle$ is unnecessary because no functions can have type *Int*; we retain this notation for uniformity with later definitions. The next step is to modify the typing rules of the system to accommodate control-flow types. For simple types, we modify the above typing rules and obtain the system of Figure 1. Note the side condition on the ABS rule to track labels of abstractions. We write $\vdash_\lambda\ z$ and say that z is *correctly typed*, if the judgement $empty\ \vdash_\lambda\ z$ is derivable ($empty$ is the empty context). A program e_0 is said to be $\vdash_\lambda$*-typable* if there exists a control-flow annotated term z such that $|z| = e_0$ and $\vdash_\lambda\ z$. An important property of the control-flow type system is that it preserves the essential character of the underlying type system. In particular, the set of typable terms remains unchanged: e_0 is $\vdash_\lambda$-typable iff e_0 is typable in the standard simply typed system. In fact there is a very close correspondence between the two systems. If $\vdash_\lambda\ z$, then by stripping away the control-flow information in the annotations appearing in z, we can construct a derivation to show that $|z|$ has simple type τ in the simply typed system, where τ is $annot(z)$ with all control-flow information removed. Conversely, if a term e has simple type τ in the simply typed system, then we can look at the derivation of this judgement, and by systematically inserting trivial control-flow information[1], obtain an annotated term z such that $\vdash_\lambda\ z$ and $|z|$ is e. The control-flow calculus also possesses standard properties such as subject reduction:

Proposition 1 (Subject Reduction) *If* $\vdash_\lambda\ z$ *and* $z \to_\beta z'$ *then* $\vdash_\lambda\ z'$. $\square$

This property establishes not only the correctness of the type aspects of the system, but also its control-flow aspects. For example, if $\vdash_\lambda\ e : \tau$ and e reduces to $\lambda^l x.e'$, then subject reduction implies that τ must have the form $\langle L, \tau_1 \to \tau_2 \rangle$ such that $l \in L$.

[1]That is, each type τ is systematically replaced by the expression $\langle L_e, \tau \rangle$, where L_e is the set of labels in e

We now address the control-flow component of the system. First note that typing is not unique: given an untyped term e_0, there may be many correctly typed terms z such that $|z| = e_0$. For example, corresponding to the untyped term $(\lambda^l x.x)$, we can construct a family of correctly typed terms:

$$(\lambda^l x.(x:\tau)):(L, \tau \to \tau)$$

where τ is any control-flow type, and L is any (finite) set of labels containing l. Part of this choice lies at the "type" level; there is also choice at the "control-flow" level. Recall that the sets of labels that are associated with each expression in a correctly typed term represent an upper bound on the functions that can flow to that expression. Smaller sets mean more accurate control-flow information, and so from a control-flow perspective, we seek annotated terms with minimal label sets.

To address this issue, we first formalize the control-flow content of an annotated term z. For any control-flow type τ, define that $\text{CF}(\tau)$ (the control-flow component of τ) is L such that τ has the form $\langle L, exp\rangle$. Lifting this to annotated terms z, define that $\text{CF}(z)$ denotes the result of replacing each annotation τ appearing in z by $\text{CF}(\tau)$. That is, $\text{CF}(z)$ is an annotated term in which annotations are finite subsets of LABELS; this term identifies exactly the control-flow content of z, stripping away the extra type structure. Now, there is a natural information ordering $\rhd$ between control-flow annotated terms that have the same underlying term structure:

- $e:L \;\rhd\; e:L'$ if $L \supseteq L'$ and e is x or 0.
- $(\lambda^l x.z):L \;\rhd\; (\lambda^l x.z'):L'$ if $L \supseteq L'$ and $z \rhd z'$.
- $(z_1\ z_2):L \;\rhd\; (z_1'\ z_2'):L'$ if $L \supseteq L'$ and $z_1 \rhd z_1'$ and $z_2 \rhd z_2'$.
- $(Succ\ z):L \;\rhd\; (Succ\ z'):L'$ if $L \supseteq L'$ and $z \rhd z'$.

We remark that this ordering can be lifted to order types and also to order correctly τ-annotated terms. The resulting orderings would give a covariant treatment of functions. Note that this ordering cannot be used as the basis for a semantic subtyping relationship – even ignoring the "type" component of control-flow types, it is not the case that "$z_1 \rhd z_2$" implies that a subterm z_1 of z can be replaced by z_2 to obtain another correctly typed term. Importantly, $\rhd$ gives rise to a notion of minimality with respect to control flow information (this follows directly from results in Section 4):

Proposition 2 (Minimality) *For any typable program e_0, the set of control-flow annotated terms $\{\text{CF}(z) : |z| = e$ and $\vdash_\lambda z\}$ has a minimal element, call it $\text{CF}(\vdash_\lambda)(e_0)$.* □

In summary, we can define typing and control-flow aspects of $\vdash_\lambda$

- Let *typable*$(\vdash_\lambda)$ denote the set of terms typable under $\vdash_\lambda$.
- Let *control-flow*$(\vdash_\lambda)$ denote the partial mapping from e_0 into $\text{CF}(\vdash_\lambda)(e_0)$.

3.2 Partial Types

The second control-flow type system we describe is based on partial types [13] (with constants $0 : Int$ and $Succ : Int \rightarrow Int$). Partial types extend simple types with a new type Ω that is a supertype of all other types. To illustrate its effect, consider the term

$$(\lambda f. \ldots (f\, 0) \ldots (f\, \lambda y.0) \ldots)(\lambda x.x)$$

which defines the identity function, binds it to the variable f, and then applies it to an integer in one place and an integer function in another. It is not possible to give this term a type using simple types because the term requires that f simultaneously has a type $Int \rightarrow \tau$ and a type $(\tau' \rightarrow Int) \rightarrow \tau$. However, by using the Ω type, we can give f the type $\Omega \rightarrow \Omega$. In other words, Ω can be used to "unify" different types (or parts of types) that have incompatible structure, but little can be done with expressions that are given this type – they can neither be applied nor used as an argument of *Succ*.

We extend the developments of the previous section to define a type system for control-flow analysis based on partial types. In this extended system, types are defined by:

$$\tau ::= \langle L, Int \rangle \mid \langle L, \Omega \rangle \mid \langle L, \tau_1 \rightarrow \tau_2 \rangle$$

These types are ordered as follows:

- $\langle L, exp \rangle \leq \langle L', \Omega \rangle$ if $L \subseteq L'$
- $\langle L, \tau_1 \rightarrow \tau_2 \rangle \leq \langle L', \tau_1' \rightarrow \tau_2' \rangle$ if $L \subseteq L'$, $\tau_1' \leq \tau_1$ and $\tau_2 \leq \tau_2'$.

Note that this ordering not only involves the "type" structure, but it also involves control-flow information. For example,

$$\langle \{l_1\}, Int \rightarrow Int \rangle \leq \langle \{l_1, l_2\}, Int \rightarrow Int \rangle$$

Typing rules for these types can be obtained by modifying the rules in Figure 1 so that both the assumptions and conclusions of the rules can be weakened using the subtype relationship. The resulting rules are given in Figure 2, and we write $\vdash_{\lambda(\Omega)} z$ if $empty \vdash_{\lambda(\Omega)} z$ is derivable using these rules. We remark that if one considers only typability questions, then we could just augment Figure 1 with a subsumption rule. However, we shall consider properties of type-annotated terms, and make connections with these annotations and annotations obtained using control-flow systems. For these purposes, the system of Figure 2 (for which there is a simple correspondence between program text and typing rules) is advantageous.

A program e_0 is said to be $\vdash_{\lambda(\Omega)}$*typable* if there exists a control-flow annotated term z such that $|z| = e_0$ and $\vdash_{\lambda(\Omega)} z$. As in the previous subsection, the control-flow type system and the underlying basic type system (this time partial types) are closely related. They have the same sets of typable terms, and derivations in one system can be replayed (after appropriate addition or suppression of structure) in the other system. The $\vdash_{\lambda(\Omega)}$ system also satisfies the subject reduction property

$$\Gamma \vdash x:\tau \qquad (\text{if } x:\tau' \text{ appears in } \Gamma \text{ and } \tau' \leq \tau) \tag{VAR}$$

$$\frac{\Gamma, x:\tau_1' \vdash z}{\Gamma \vdash \lambda^l x.z:\langle L, \tau_1 \rightarrow \tau_2\rangle} \quad (\text{if } \tau_1 \leq \tau_1', \mathit{annot}(z) \leq \tau_2 \text{ and } l \in L) \tag{ABS}$$

$$\frac{\Gamma \vdash z \qquad \Gamma \vdash z'}{\Gamma \vdash (z\ z'):\tau} \quad (\text{if } \mathit{annot}(z) \text{ is } \langle L, \tau_1 \rightarrow \tau_2\rangle, \mathit{annot}(z') \leq \tau_1 \text{ and } \tau_2 \leq \tau\ ) \tag{APP}$$

$$\Gamma \vdash 0:\tau \qquad (\text{if } \langle L, \mathit{Int}\rangle \leq \tau\ ) \tag{INT}$$

$$\frac{\Gamma \vdash z}{\Gamma \vdash (\mathit{Succ}\ z):\tau} \quad (\text{if } \mathit{annot}(z) \leq \langle \emptyset, \mathit{Int}\rangle \text{ and } \langle \emptyset, \mathit{Int}\rangle \leq \tau) \tag{SUCC}$$

Figure 2: Rules of inference for simple subtype system.

(the obvious modification of Proposition 1). Moreover, the $\triangleright$ preference ordering can be used to order the accuracy of control-flow information in annotated terms. An appropriate version of the minimality proposition (Proposition 2) also holds, and so, given a $\vdash_{\lambda(\Omega)}$-typable program e_0, we can define a "best" control-flow annotated term $\mathrm{CF}(\vdash_{\lambda(\Omega)})(e_0)$. Hence, typability and control-flow aspects of $\vdash_{\lambda(\Omega)}$ can be given:

- Let $\mathit{typable}(\vdash_{\lambda(\Omega)})$ denote the set of terms typable under $\vdash_{\lambda(\Omega)}$.
- Let $\mathit{control\text{-}flow}(\vdash_{\lambda(\Omega)})$ denote the partial mapping from e_0 into $\mathrm{CF}(\vdash_{\lambda(\Omega)})(e_0)$.

3.3 Recursive Types

The third system is based on recursive types, another extension of simple types. This time types are extended by adding a fixpoint construction $\mu\alpha.\tau$. The effect of this addition is that terms involving recursion such as $(\lambda^l x.(x\ x))\ (\lambda^{l'} x.(x\ x))$ can be typed. To define a control-flow type system based on recursive types, we first define *open types* to be expressions of the following form:

$$\sigma ::= \alpha \mid \langle L, \mathit{Int}\rangle \mid \langle L, \sigma_1 \rightarrow \sigma_2\rangle \mid \mu\alpha.\sigma$$

where α ranges over a countably infinite collection of type variables, and $\mu\alpha.$ is treated as a binding construct. Free and μ-bound occurrences of type variables in open types are defined in the usual way. We define an equality on open types by $\mu\alpha.\sigma = \sigma[\mu\alpha.\sigma/\alpha]$, where $\sigma[\mu\alpha.\sigma/\alpha]$ denotes the result of replacing free occurrences of α by $\mu\alpha.\tau$ in τ (after renaming of bound variables, if necessary). This is extended in the obvious way to become a congruence on open types: if $\sigma_1 = \sigma_2$ and σ' is the result of replacing σ_1 by σ_2 in σ then $\sigma' = \sigma$. Finally, types τ are defined to be those expressions σ that do not contain free type variables.

The typing rules for this new system consist of exactly the rules previously given in Figure 3.1. We write $\vdash_{\lambda(\mu)} z$ if $\mathit{empty} \vdash z$ is derivable. A program e_0 is said to be $\vdash_{\lambda(\mu)}$*-typable* if there exists a control-flow annotated term z such that $|z| = e_0$ and $\vdash_{\lambda(\mu)} z$. This control-flow type system preserves many of the basic properties of the underlying recursive type system. The set of typable terms is the same, and subject reduction is preserved. An appropriate version of the minimality proposition (Proposition 2) also holds; for a $\vdash_{\lambda(\mu)}$-typable program e_0, let $\text{CF}(\vdash_{\lambda(\mu)})(e_0)$ denote the minimal control-flow annotated term thus obtained. Typability and control-flow aspects of $\vdash_{\lambda(\mu)}$ can now be defined:

- Let *typable*$(\vdash_{\lambda(\mu)})$ denote the set of terms typable under $\vdash_{\lambda(\mu)}$.
- Let *control-flow*$(\vdash_{\lambda(\mu)})$ denote the partial mapping from e_0 into $\text{CF}(\vdash_{\lambda(\mu)})(e_0)$.

3.4 Amadio/Cardelli's System

Finally, we define a control-flow type system that combines partial types and recursive types. The starting point of this development is essentially Amadio and Cardelli's recursive subtype system [1] (with extensions for 0 and *Succ*). For definitional convenience, we shall use a formulation of recursive subtypes that differs slightly from those used previously in the literature [1, 5, 9]. We therefore begin by presenting this system without the control-flow information component. Define expressions σ by

$$\sigma ::= \alpha \mid Int \mid \Omega \mid \sigma_1 \rightarrow \sigma_2 \mid \mu\alpha.\sigma$$

where α ranges over type variables. Occurrences of types variables are defined to be free and μ-bound in an expression σ in the usual way. Again we define a congruence on open types: $\mu\alpha.\sigma = \sigma[\mu\alpha.\sigma/\alpha]$. Types τ are defined to be those expressions σ that do not contain free type variables.

Next, we define an ordering between types (strictly speaking, this ordering is between equivalence classes of types). This ordering is defined by successive approximation (see [1, 5] for related constructions). Specifically, define an ordering $\leq_k$, $k \geq 0$, as follows:

- $\tau \leq_0 \tau'$
- $\tau \leq_k \Omega$
- $\mu\alpha.\alpha \leq_k \tau$
- $Int \leq_k Int$
- $\tau_1 \rightarrow \tau_2 \leq_k \tau_1' \rightarrow \tau_2'$ if $\tau_1' \leq_{k-1} \tau_1$ and $\tau_2 \leq_{k-1} \tau_2'$

where τ and τ' are arbitrary types. We now define that $\tau \leq \tau'$ if $\tau \leq_k \tau'$ for all $k \geq 0$. If we ignore *Int*, identify $\top$ with Ω and treat $\bot$ as shorthand for $\mu\alpha.\alpha$, then this ordering is equivalent to the one given by Amadio and Cardelli (call it $\leq_{ac}$) in the following *sense*:

Proposition 3 *For all τ_1 and τ_2 not containing Int,* $\tau_1 \leq \tau_2$ *iff* $\tau_1 \leq_{ac} \tau_2$. □

To extend this system to control-flow, we first define open types σ by:

$$\sigma ::= \alpha \mid \langle L, Int\rangle \mid \langle L, \Omega\rangle \mid \langle L, \sigma_1 \to \sigma_2\rangle \mid \mu\alpha.\sigma$$

where α ranges over type variables. Then, types τ are those expressions σ that do not contain free type variables. A congruence on types is defined by $\mu\alpha.\sigma = \sigma[\mu\alpha.\sigma/\alpha]$, and an ordering $\leq_k$, $k \geq 0$, is defined by:

- $\tau \leq_0 \tau'$
- $\langle L, exp\rangle \leq_k \langle L', \Omega\rangle$ if $L \subseteq L'$
- $\langle L, \mu\alpha.\alpha\rangle \leq_k \langle L', exp\rangle$ if $L \subseteq L'$
- $\langle L, Int\rangle \leq_k \langle L', Int\rangle$ if $L \subseteq L'$
- $\langle L, \tau_1 \to \tau_2\rangle \leq_k \langle L', \tau_1' \to \tau_2'\rangle$ if $\tau_1' \leq_{k-1} \tau_1$, $\tau_2 \leq_{k-1} \tau_2'$ and $L \subseteq L'$.

We define that $\tau \leq \tau'$ if $\tau \leq_k \tau'$ for all $k \geq 0$. The inference rules for these types are the exactly those from Figure 2. We write $\vdash_{\lambda(\Omega,\mu)}$ to denote the type-correctness judgement thus defined. Again subject reduction carries over to the control-flow types system, and typability for the control-flow system coincides with that for the underlying Amadio/Cardelli system. An appropriate version of the minimality proposition (Proposition 2) also holds, and so we can define, given a $\vdash_{\lambda(\Omega,\mu)}$-typable program e_0, the minimal control-flow annotated version of e_0, call it $\textsc{cf}(\vdash_{\lambda(\Omega,\mu)})(e_0)$. Typability and control-flow aspects of $\vdash_{\lambda(\Omega,\mu)}$ can now be given:

- Let *typable*$(\vdash_{\lambda(\Omega,\mu)})$ denote the set of terms typable under $\vdash_{\lambda(\Omega,\mu)}$.
- Let *control-flow*$(\vdash_{\lambda(\Omega,\mu)})$ denote the partial mapping from e_0 into $\textsc{cf}(\vdash_{\lambda(\Omega,\mu)})(e_0)$.

4 Control-Flow Analysis

The purpose of control-flow is to determine information about the functions that can be called from various program points during execution of a program. In the following development, a function shall be identified its label. We define control-flow systems using annotated terms and consistency conditions between "neighbouring" annotations. By varying these consistency conditions and some additional properties, we shall define a family of four control-flow systems.

If we were concerned only with control-flow information, it would be sufficient to annotate terms using control-flow flow information (finite subsets of LABELS). However, we wish to construct control-flow systems that capture typing properties (in addition to control-flow properties), for the purpose of establishing closer links with the type systems presented in Section 3. Hence, we shall use annotations γ that are finite subsets of LABELS $\cup\{Int\}$. Our presentations of control-flow systems is closely related to those by Palsberg and Schwartzbach [7, 8].

4.1 Standard CFA

We begin a system that corresponds to a widely studied notion of control-flow analysis [3, 4, 10, 7, 8], often called control-flow-0. The system we give is equivalent to the system described by Palsberg and Schwartzbach in [7, 8]. The only difference is in presentation: we directly use annotated terms instead of set variables and constraints. We present the system by defining that an annotated term z is *well-annotated* if it satisfies the following conditions:

1. if $\lambda^l x.z' : \gamma$ appears in z then $l \in \gamma$.
2. if $(z_1\ z_2) : \gamma$ appears in z then $Int \notin annot(z_1)$ and for all $l \in annot(z_1)$, if $\lambda^l x.z'$ appears in z then:
   (a) $annot(z') \subseteq \gamma$, and
   (b) if x occurs free in z' with annotation γ_x then $annot(z_2) \subseteq \gamma_x$.
3. if $0 : \gamma$ appears in z then $\gamma \supseteq \{Int\}$
4. if $(Succ\ z') : \gamma$ appears in z then $\gamma \supseteq \{Int\}$ and $annot(z') \subseteq \{Int\}$.

We shall refer to this system as $\text{CFA}_{\subseteq}$. As in the previous section, we shall define typing and control-flow properties of this system by relating untyped programs and annotated terms. First, consider typing. Define that a program e_0 is $\text{CFA}_{\subseteq}$*-type-consistent* if there exists an annotated term z such that $e_0 = |z|$ and z is well-annotated according to the above definition. Next consider control-flow. Clearly there will in general be many well-annotated versions of e_0, each identifying potentially different control-flow information. These terms can be related using the ordering $\triangleright$ defined in Section 3. Strictly speaking, $\triangleright$ orders terms that are annotated with finite subsets of LABELS, but it can clearly be generalized to order any annotated terms where the annotations are sets.

Proposition 4 (Minimality) *For any* $\text{CFA}_{\subseteq}$*-type-consistent term* e_0*, there is* $\triangleright$*-minimal well-annotated term* z *such that* $|z| = e_0$. $\square$

This proposition follows from the fact that well-annotated terms are closed under intersection. Specifically, if z_1 and z_2 are well-annotated terms such that $|z_1| = |z_2|$ then we can define an annotated term $z_1 \cap z_2$ by intersecting the corresponding annotations of z_1 and z_2, and this new term will be well-annotated and such that $|z_1| = |z_2| = |z_1 \cap z_2|$. This follows immediately from inspection of the well-annotated conditions.

Using Proposition 4, we can define the control-flow component of $\text{CFA}_{\subseteq}$. First, recall that annotated terms used in the system may include Int. To give just the control-flow information of an annotated term, let $\text{CF}(z)$ denote the result of replacing each annotation in A appearing in z by $A - \{Int\}$. Finally, given a program e_0 that is $\text{CFA}_{\subseteq}$-type-consistent, define that $\text{CF}(\text{CFA}_{\subseteq})(e_0)$ is $\text{CF}(z)$ where z is the $\triangleright$-minimal well-annotated term such that $|z| = e_0$. We can now define typing and control-flow aspects of $\text{CFA}_{\subseteq}$ as follows:

- Let *typable*$(\text{CFA}_{\subseteq})$ denote the set of type-consistent terms under $\text{CFA}_{\subseteq}$.
- Let *control-flow*$(\text{CFA}_{\subseteq})$ denote the partial mapping from e_0 into $\text{CF}(\text{CFA}_{\subseteq})$

4.2 Standard CFA without Recursion

The CFA$_{\subseteq}$ system defined in the previous subsection can reason about recursion. For example, one well-annotated version of $(\lambda^l x.(x\ x))\ (\lambda^{l'} x.(x\ x))$ is

$$((\lambda^l x.(x:\{l'\}\ x:\{l'\}):\{\}):\{l\}\ \ (\lambda^{l'} x.(x:\{l'\}\ x:\{l'\}):\{\}):\{l'\}):\{\}$$

We now define a control-flow system that is based on equality but specifically excludes programs involving recursion. In this modified system, an annotated term z is well-annotated if it satisfies the conditions given in Subsection 4.1 and in addition there is a non-reflexive ordering $\succ$ on LABELS such that

- if $\lambda^l x.z':\gamma$ appears in z and x occurs free in z' with annotation γ_x, then $(\forall l' \in \gamma_x)(l \succ l')$.

Call this system CFA$_{\subseteq,\neg Y}$. A program e_0 is CFA$_{\subseteq,\neg Y}$-*type-consistent* if there exists an annotated term z such that $e_0 = |z|$ and z is well-annotated according to the above definition. Given, a type-consistent program e_0, we can again use $\rhd$ to define a unique minimal well-annotated term corresponding to e_0, and so using CF we can CF(CFA$_{\subseteq,\neg Y}$)(e_0) in analogy with the previous subsection. Hence, typing and control-flow aspects of CFA$_{\subseteq,\neg Y}$ can be defined:

- Let *typable*(CFA$_{\subseteq,\neg Y}$) denote the set of type-consistent terms under CFA$_{\subseteq,\neg Y}$.
- Let *control-flow*(CFA$_{\subseteq,\neg Y}$) denote the partial mapping from e_0 into CF(CFA$_{\subseteq,\neg Y}$)

4.3 CFA via Equality

The next system we consider essentially corresponds to the analysis considered by Bondorf and Jorgensen [2]. The only differences are that we do not consider arbitrary data-constructors, and we include *Int* to reason about type consistency. The definition of well-annotated programs for this system is a modification of the definition in Subsection 4.1 in which subset is replaced by equality. Specifically, an annotated term z is *well-annotated* if:

1. if $(\lambda^l x.z'):\gamma$ appears in z then $l \in \gamma$.
2. if $(z_1\ z_2):\gamma$ appears in z, $l \in$ *annot*(z_1) and $\lambda^l x.z'$ appears in z then:
   (a) *annot*$(z') = \gamma$, and
   (b) if x occurs free in z' with annotation γ_x then *annot*$(z_2) = \gamma_x$.
3. if $0:\gamma$ appears in z then $\gamma = \{Int\}$
4. if $(Succ\ z'):\gamma$ appears in z then $\gamma =$ *annot*$(z') = \{Int\}$.

Call this system CFA$_{=}$. Definitions of type-consistency, minimality and CF(CFA$_{=}$)(e_0) can be given analogously to those in the previous subsections. Typing and control-flow aspects of CFA$_{=}$ are defined by:

- Let *typable*($\mathrm{CFA}_=$) denote the set of type-consistent terms under $\mathrm{CFA}_=$.
- Let *control-flow*($\mathrm{CFA}_=$) denote the partial mapping from e_0 into $\mathrm{CF}(\mathrm{CFA}_=)$

4.4 Standard CFA without Recursion

As with $\mathrm{CFA}_\subseteq$, the $\mathrm{CFA}_=$ system can reason about recursion. We now define a fourth control-flow analysis system that combines $\mathrm{CFA}_=$ with the recursion restriction. In this new system, an annotated term z is well-annotated if it satisfies the conditions given in Subsection 4.1 and in addition satisfies the recursion restriction given in Subsection 4.2. Call this system $\mathrm{CFA}_{=,\neg Y}$. Definitions of type-consistency, minimality and $\mathrm{CF}(\mathrm{CFA}_{=,\neg Y})(e_0)$ can be given analogously to those in the previous subsections. Typing and control-flow aspects of $\mathrm{CFA}_=$ are defined by:

- Let *typable*($\mathrm{CFA}_{=,\neg Y}$) denote the set of type-consistent terms under $\mathrm{CFA}_{=,\neg Y}$.
- Let *control-flow*($\mathrm{CFA}_{=,\neg Y}$) denote the partial mapping from ϵ_0 into $\mathrm{CF}(\mathrm{CFA}_{=,\neg Y})$

5 Types ≡ Control-Flow

We now establish a series of correspondences between the four type systems and the form control-flow systems that have been presented. In short, we prove the following equivalences:

$\vdash_\lambda \ \equiv\ \mathrm{CFA}_{=,\neg Y}$	$\vdash_{\lambda(\mu)} \ \equiv\ \mathrm{CFA}_=$
$\vdash_{\lambda(\Omega)} \ \equiv\ \mathrm{CFA}_{\supseteq,\neg Y}$	$\vdash_{\lambda(\Omega,\mu)} \ \equiv\ \mathrm{CFA}_\supseteq$

That is, the two subtype systems correspond to the control-flow systems based on $\subseteq$; the two other type systems correspond to the control-flow systems based on $=$. Non-recursive type systems correspond to control-flow systems that exclude control-flow cycles, and recursive type systems correspond to control-flow systems that do not restrict control-flow cycles.

For each pair of type and control-flow system, we establish that the type and control-flow components of the systems correspond. For example, for $\vdash_\lambda$ and $\mathrm{CFA}_{=,\neg Y}$ we prove that (a) *typable*($\vdash_\lambda$) = *typable*($\mathrm{CFA}_{=,\neg Y}$), and (b) *control-flow*($\vdash_\lambda$) = *control-flow*($\mathrm{CFA}_{=,\neg Y}$). These connections are proved by showing that a "correctly annotated" term in one system can be used to reconstruct a closely related "correctly annotated" term in the other system. Specifically, for each pair of systems, we prove the following two theorems:

Theorem 1 *If z_{type} is correctly typed then there exists a well-annotated term z_{cfa} such that $|z_{type}| = |z_{cfa}|$ and* $\mathrm{CF}(z_{type}) \rhd \mathrm{CF}(z_{cfa})$.

Proof Sketch(for $\vdash_{\lambda(\Omega,\mu)}$ and $\mathrm{CFA}_\supseteq$) Suppose that $\vdash_{\lambda(\Omega,\mu)}$ z_{type}. Unfortunately, we cannot proceed to define a well-annotated term z_{cfa} directly using the control-flow information in z_{type} because there is some slack in the definition of correctly typed terms with respect to the specification of control-flow information. In essence, the set of control-flow information associated with a term can often be enlarged using subtyping, without affecting correctness of the type annotations. Due to minor structural differences between type systems and control-flow analysis, such enlargements cannot necessarily be replayed in a control-flow analysis. Hence, we must first trim this slack. In the context of z_{type}, define a function *trim* on types as follows: $trim(\tau)$ is the result of replacing the control-flow component of τ with the set of all labels $l \in \mathrm{CF}(\tau)$ such that if $\lambda^l x.z : \tau_l$ appears in z_{type}, then $\tau_l \leq \tau$. Then, define that $trim(z_{type})$ is the annotated term such that all types τ appearing in z_{type} (and subexpressions thereof that are types) are systematically replaced by $trim(\tau)$. It is easy to verify that $\vdash_{\lambda(\Omega,\mu)} trim(z_{type})$ and that $z_{type} \rhd trim(z_{type})$.

Now, consider systematically replacing the annotations in $trim(z_{type})$ as follows: an annotation τ is replaced by $\mathrm{CF}(\tau) \cup \{Int\}$ if $\langle\{\}, Int\rangle \leq \tau$ and $\mathrm{CF}(\tau)$ otherwise. Call the resulting term z_{cfa}. Clearly $|z_{cfa}| = |z_{type}|$ and $\mathrm{CF}(z_{type}) \rhd \mathrm{CF}(z_{cfa})$. It remains to show that z_{cfa} is well-annotated according to the control-flow definitions. This is mostly straightforward; the most interesting case is application, and here the proof follows the following property of any annotation appearing in $trim(z_{type})$

if $l \in \mathrm{CF}(\tau)$ and $\lambda^l x.z : \tau_l$ appears in $trim(z_{type})$, then $\tau_l \leq \tau$.

This is an immediate consequence of the *trim* construction. $\Box$

The proof for the other three cases of this theorem are modifications of this basic result. For type systems that do not include recursion, the above construction can be modified to yield a well-annotated terms that does not involve control-flow cycles. For type systems that do not involve subtyping, the type systems do not include the subtyping rule, and when translated to well-annotated terms (via the above construction), this means that the control-flow consistency conditions become equalities rather than inclusions.

Theorem 2 *If z_{cfa} is well-annotated then there exists a correctly typed term z_{type} such that $|z_{cfa}| = |z_{type}|$ and $\mathrm{CF}(z_{cfa}) = \mathrm{CF}(z_{type})$.*

Proof Sketch(for $\vdash_{\lambda(\Omega,\mu)}$ and $\mathrm{CFA}_{\supseteq}$) Suppose that z_{cfa} is well-annotated according to $\mathrm{CFA}_{\supseteq}$, and consider the following translation $T(\gamma)$ from the annotations γ appearing in z_{cfa} into types:

$$T(\gamma) = \begin{cases} \bot & \text{if } \gamma = \{\} \\ Int & \text{if } \gamma = \{Int\} \\ \langle \gamma, T(dom(\gamma)) \rightarrow T(ran(\gamma))\rangle & \text{if } \gamma \neq \{\} \text{ and } \gamma \subseteq \text{LABELS} \\ \top & \text{otherwise} \end{cases}$$

where *dom* and *ran* are defined by:

$$\begin{aligned} dom(\gamma) &= \bigcap\{\gamma : \lambda^l x.z \text{ appears in } z_{cfa} \text{ and some free occurrence of } x \text{ in } z \text{ has annotation } \gamma\} \\ ran(\gamma) &= \bigcap\{annot(z) : \lambda^l x.z \text{ appears in } z_{cfa}\} \end{aligned}$$

To explain these, consider an abstraction $\lambda^l x.z$ appearing in z_{cfa}. Annotations on the free occurrences of x in z represent upper bounds on the arguments to which this abstraction can be "safely" applied. Hence, $dom(\gamma)$, the intersection of these annotations over all $l \in \gamma$, is the largest set that satisfies all of these bounds, and this represents an upper bound on possible applications of a term with label γ. Similarly, $ran(\gamma)$ represents a lower bound on the possible results of an application of a term with annotation γ. Observe that the above translation defines an infinite object if z_{cfa} involves control-flow cycles. To deal with this situation, we modify the mapping T as follows. First, we set aside a collection of distinct type variable α_γ for each annotation γ. We then modify the above definition to give a translation $T^{\tilde{\gamma}}$ which carries the sequence of annotations $\tilde{\gamma}$ that have already been considered. The key modification is the part for arrow types: if $\gamma \subseteq$ LABELS then $T^{\tilde{\gamma}}(\gamma)$ is

$$\begin{array}{ll} \alpha_\gamma & \text{if } \gamma \in \tilde{\gamma} \\ \langle \gamma, T^{\tilde{\gamma}.\gamma}(dom(\gamma)) \rightarrow T^{\tilde{\gamma}.\gamma}(ran(\gamma))\rangle & \text{otherwise} \end{array}$$

In other words, the recursion operator μ is used to fold infinite types into finite ones. Now define that $T(\tau)$ is $T^{\emptyset}(\tau)$ where $\emptyset$ denotes the empty sequence. A key property of T is that if $\gamma_1 \subseteq \gamma_2$ then $T(\gamma_1) \subseteq T(\gamma_2)$. Finally, define that z_{types} is the term that results from systematically replacing each annotation γ in z_{cfa} by $T(\gamma)$. Clearly $|z_{cfa}| = |z_{type}|$ and $\text{CF}(z_{cfa}) = \text{CF}(z_{type})$, and it only remains to show that z_{types} is correctly typed. This is essentially a structural induction on z_{types} (although note that contexts have the be constructed), and is fairly straightforward. $\square$

Again the proof for the other three cases of this theorem are modifications of this basic result. For control-flow systems that exclude recursion, note that the original definition of T is sufficient since the ordering $\succ$ on LABELS means that cycles cannot appear in this definition. Hence, it is clear that the image of T consists of non-recursive types. For control-flow systems that do not involve subtyping, the control-flow consistency conditions become equalities rather than inclusions. When translated to types, such annotations correspond to type derivations that do not require subtyping (we remark that when the above construction is applied to a $\text{CFA}_{=,\neg Y}$-well-annotated term, the resulting term can contain $\perp$; by replacing such occurrences by Int, we obtain the desired result).

References

[1] R. Amadio and L. Cardelli, "Subtyping Recursive types", *ACM-TOPLAS*, 15(4):575-631, 1993. (Also in POPL-91, pp. 104–118).

[2] A. Bondorf and J. Jorgensen, "Efficient Analysis for Realistic Off-Line Partial Evaluation", *Journal of Function Programming*, 3(3), pp. 315–346, 1993.

[3] N. Jones, "Flow Analysis of Lambda Expressions", *Symp. on Functional Languages and Computer Architecture*, pp. 66-74, 1981.

[4] N. Jones, "Flow Analysis of Lazy Higher-Order Functional Programs", in *Abstract Interpretation of Declarative Languages*, S. Abramsky and C. Hankin (Eds.), Ellis Horwood, 1987.

[5] D. Kozen, J. Palsberg, M. Schwartzbach, "Efficient Recursive Subtyping" *POPL-93*, pp. 419–428, 1993.

[6] R. Milner, "A Theory of Type Polymorphism in Programming", *Journal of Computer System Science*, 17, pp. 348–375.

[7] J. Palsberg and M. Schwartzbach, "Safety Analysis versus Type Inference for Partial Types" Information Processing Letters, Vol 43, pp. 175–180, North-Holland, September 1992.

[8] J. Palsberg and M. Schwartzbach, "Safety Analysis versus Type Inference" *Information and Computation*, to appear.

[9] J. Palsberg and P. O'Keefe, "A Type System Equivalent to Flow Analysis", to appear POPL-95.

[10] O. Shivers, "Control Flow Analysis in Scheme", *Proc. 1988 ACM Conf. on Programming Language Design and Implementation*, Atlanta, pp. 164–174, 1988.

[11] Y. Tang and P. Jouvelot, "Control-Flow Effects for Escape Analysis", WSA'92, Bordeaux, France. 1992.

[12] Y. Tang and P. Jouvelot, "Separate Abstract Interpretation for Control-Flow Analysis", TACS-94, LNCS 789, pp.224-243, 1994.

[13] S. Thatte, "Type Inference with Partial Types", *ICALP-88*, LNCS 317, pp. 615–629, 1988.

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3890