Termination Analysis and Call Graph Construction for Higher-Order Functional Programs

Damien Sereni
Oxford University
dsereni@comlab.ox.ac.uk

Abstract

The analysis and verification of higher-order programs raises the issue of control-flow analysis for higher-order languages. The problem of constructing an accurate call graph for a higher-order program has been the topic of extensive research, and numerous methods for flow analysis, varying in complexity and precision, have been suggested.

While termination analysis of higher-order programs has been studied, there has been little examination of the impact of call graph construction on the precision of termination checking. We examine the effect of various control-flow analysis techniques on a termination analysis for higher-order functional programs. We present a termination checking framework and instantiate this with three call graph constructions varying in precision and complexity, and illustrate by example the impact of the choice of call graph construction.

Our second aim is to use the resulting analyses to shed light on the relationship between control-flow analyses. We prove precise inclusions between the classes of programs recognised as terminating by the same termination criterion over different call graph analyses, giving one of the first characterisations of expressive power of flow analyses for higher-order programs.

Categories and Subject Descriptors F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—Program Analysis

General Terms Verification, Theory

Keywords Termination, Functional programs, Semantics, Program Analysis

1. Introduction

Termination analysis is a fundamental problem in program verification, and proofs of program correctness require proving termination of a program, often independently from the proof of functional correctness of the program. Furthermore, termination checking has been shown to address real issues in program correctness (Cook et al. 2006a,b; Berdine et al. 2006).

Termination checking is often limited to first-order programs, or first-order term rewriting systems, though a number of extensions to higher-order programs have been made in recent years (Jones and Bohr 2004; Sereni and Jones 2005; Giesl et al. 2005, 2006). Termination analysis for higher-order programs introduces specific problems, in particular that of *control-flow analysis*. The flow of control of a first-order program is apparent from the program text, however this is not the case for higher-order programs. Indeed there has been a great deal of research on control-flow analysis (CFA) of higher-order programs (Shivers 1991; Jones 1987; Shivers 1988; Malacaria and Hankin 1998; Might and Shivers 2006), and several variants of higher-order CFA have been proposed. These present different characteristics, in particular in the aspects of precision and complexity.

However, it has proved difficult to account for the precise relationships between these flow analyses. There is an intuitive sense in which we can view one method for CFA as more precise than another, and experimental evidence can justify such statements, but it remains difficult to establish such properties formally.

The ability to define such statements precisely is nonetheless crucial to our understanding of control-flow analysis. Indeed a precise hierarchy of expressiveness allows one to make a rational choice of one analysis over another.

The aim of this work is to highlight the importance of control-flow analysis in achieving good termination analysis results, and to use the effectiveness of termination analysis to give a precise account of the precision of control-flow analysis.

The termination problem gives us a unique handle on evaluating the precision of control-flow analysis. Unlike, say, the verification of safety properties, termination does not invite other issues such as the logic used to express properties. A method for termination analysis (as long as it is sound) simply partitions the set of terminating programs into two sets: the set of programs accepted by the analysis, and the set of false negatives. It is thus in principle straightforward to compare the expressiveness of termination analyses, as this reduces to comparing the respective sets of accepted programs. We shall use precisely this measure to evaluate the effectiveness of flow analyses: one such analysis is strictly superior (in precision) if it yields a strictly larger set of terminating programs.

Our contributions are:

1. We define a framework for termination analysis of higher-order, purely functional programs that allows the underlying control-flow analysis to be varied at will.
2. We present three control-flow analyses: the known 0CFA analysis (Shivers 1991), a family of more precise analyses known as k-limited CFA, and finally an analysis based on tree automata with different precision characteristics.
3. Finally, we give a precise characterisation of the relationships between the family of analyses thus defined, giving the first systematic study of expressiveness of termination analyses.

ICFP'07, October 1–3, 2007, Freiburg, Germany.

We shall not regard experimental evaluation as an aim of this paper. While it is doubtless important to obtain such validation, the task is made somewhat complicated by a lack of good canonical suites of higher-order purely functional programs. In its place, the relationships between the precision of our analyses serve to validate the use of more complex analyses to improve results. Much detail is elided for space reasons, and the interested reader may find details and proofs in a companion technical report (Sereni 2006).

2. Size-Change Termination for Higher-Order Programs

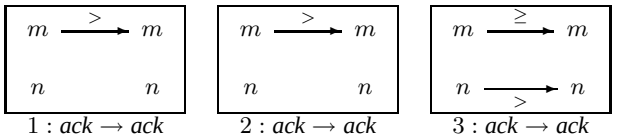
2.1 Size-Change Termination

Our basis for termination analysis is the *size-change termination* (SCT) method (Lee et al. 2001). In this framework, we assume that each datatype in the language carries a natural well-founded order (though this restriction can be lifted (Avery 2006), we keep it for simplicity). The SCT method then attempts to detect as terminating all those programs in which any potential infinite sequence of calls would give rise to an infinitely decreasing value. As all datatypes are well-founded, infinitely decreasing value chains are impossible, and so termination is established.

Let us illustrate the SCT method in more detail by example. Consider the following (somewhat archetypal) program:

```
ack (m,n) = if m=0 then n + 1 else
            if n=0 then 1ack (m−1, 1) else
            2ack (m−1, 3ack (m, n−1))
```

The *ack* function has three recursive calls, labelled in the program text. For each of these calls, a simple analysis can produce a *size-change graph*, shown below:



$1 : ack \to ack$ $2 : ack \to ack$ $3 : ack \to ack$

To wit, in calls 1 and 2 it is detected that the value of m in the callee is strictly less than its value in the calling instance. Likewise, in the third call the value of m is guaranteed not to increase (in fact, m is passed unchanged), while the value of n decreases.

It is not difficult to see by inspection from the above graphs that any infinite sequence of calls would cause either the value of n to decrease infinitely, or the value of m to decrease infinitely. As a result, no infinite sequence of calls is possible, as infinitely decreasing chains of natural numbers are impossible.

The SCT framework formalises and automates the above discussion. First presented for first-order programs (Lee et al. 2001), and later extended for higher-order programs (Jones and Bohr 2004; Sereni and Jones 2005), the SCT framework can be summarised as follows:

1. The *call graph* of the program is computed,
2. Each edge in the call graph is annotated with a *size-change graph* describing dataflow and size changes in the call safely, and
3. The annotated call graph is checked for infinite paths which do not imply infinite decrease in some value.

The third part of this process is decidable (if computationally expensive, as it is a PSPACE-complete problem) (Lee et al. 2001). However, the first two phases of the procedure are not, and it is of particular interest to focus on the issues raised by call graph construction. We shall therefore not describe the algorithm due to Lee et al. (2001) for proving termination given an annotated call graph, but rather focus on the first two points in the remainder of the paper.

2.2 Termination of Higher-Order Programs

The SCT termination check was originally introduced for first-order purely functional languages only. However, all major functional languages support higher-order functions. These make the task of termination analysis substantially more complex, for two essentially independent reasons.

Comparing Functional Values The first issue in the analysis of higher-order programs is the presence of functional values. The SCT framework relies on a well-founded order on values computed in the program's execution. In the first-order case it is straightforward to compare values: natural numbers may be compared using the numerical $<$ order, while structures such as lists allow a range of natural well-founded orders, such as the sublist order. However, the appropriate order for functional values is not as obvious.

We shall follow previous work on this problem (Jones and Bohr 2004; Sereni and Jones 2005), and opt for a *syntactic* order on function values. The crucial observations is that in the execution of a program, a function value is represented as a closure, that is to say a function definition together with an environment assigning values to free variables. As a value may itself be a closure, this gives rise to a tree structure, where the nodes are function names or constants.

The order on such values is just given by the subtree order. As an example, consider the following program:

```
map f xs = match xs with
    [] → []
  | y::ys → f y :: map f ys
compose f g x = f (g x)
add x y = x + y

map (compose (add 1) (add 2)) [0;1]
```

Evaluation of this program will cause the following state, s say, to be evaluated (where $(\!|f : \rho|\!)$ represents a closure of f with environment ρ):

$$(\!|\mathit{map} : \begin{array}{l}\{f \mapsto (\!|\mathit{compose} : \{\ \begin{array}{l} f \mapsto (\!|\mathit{add} : \{x \mapsto 1\}|\!), \\ g \mapsto (\!|\mathit{add} : \{x \mapsto 2\}|\!)\}\end{array}\ |\!), \\ \mathit{xs} \mapsto 0 :: 1 :: []\}\end{array}\ |\!)$$

The nesting structure of this closure gives rise to the order relation, defined as the subtree ordering. For instance,

$$(\!|\mathit{add} : \{x \mapsto 1\}|\!) < s$$

This order is clearly well-founded, as closures are finite trees. Furthermore, it is useful in proving termination of nontrivial programs — we shall return to this point in the next section and further invite the reader to find a more complete discussion in the companion report (Sereni 2006).

Call Graph Construction The second issue in the analysis of higher-order programs, and the main focus of this paper, is *call graph construction*. In the first-order case, it is straightforward to define the call graph of a program: this is a graph with a node for each function in the program, and an arrow $f \to g$ iff a call to g appears (textually) in the body of function f. This call graph, annotated with size-change graphs as shown above, is the structure on which the SCT algorithm is defined. The setting of higher-order programs complicates both the definition and construction of the call graph, and we shall briefly outline the issues in this section.

While a formal definition of the call graph of a program in our more general setting must wait until the next section, let us give the intuition behind the definition. The execution of the program

is represented by a *call* relation $s \Rightarrow s'$ (between states in the execution). The intended meaning of the judgement $s \Rightarrow s'$ is that the evaluation of state s depends on (and therefore triggers) the evaluation of state s'. This is a natural generalisation of the dynamic call relation for first-order programs, but this more general definition extends to arbitrary languages. The crucial property of this relation is that nontermination should be reflected as an infinite call sequence, exactly as in the first-order case.

The *dynamic* call graph of a program is just the graph of the $\Rightarrow$ relation (restricted to reachable states). It is therefore the case that a program is terminating iff its dynamic call graph is finite and acyclic. Accordingly, the aim of our termination analysis will be to prove that infinite paths in the call graph are impossible.

Naturally, the dynamic call graph of a program cannot be computed in general. We are therefore concerned with static approximations of the dynamic call graph. We use the framework of abstract interpretation (Cousot and Cousot 1977) to define this approximation. Informally, the *static* call graph is a finite graph, where each node in the graph (henceforth called an abstract state) represents a set of nodes in the dynamic call graph. Nodes in the static call graph are drawn from a finite set State^α of abstract states.

Furthermore, the static call graph should safely approximate the dynamic call graph, in the sense that each call in the dynamic call graph should be reflected in the static call graph. This condition is sufficient to guarantee that any infinite paths in the dynamic call graph is reflected in the static call graph, and thus that termination may be proved by showing that infinite paths in the static call graph give rise to infinite descent. To make this general framework concrete, however, two aspects must be decided:

1. How the $\Rightarrow$ relation is defined, and
2. What the set State^α and the abstraction function α should be.

In the next section we shall settle the first question. However, no one answer can be given to the second question: many different choices have been proposed, and these differ in the size of the set State^α, as well as the precision they offer.

3. Termination Analysis

In the previous section, we have introduced the main components of higher-order termination analyses in our framework: the SCT algorithm, the size order on values, the dynamic call graph (defined by the $\Rightarrow$ relation) and the static call graph. In this section we make these ideas more precise.

3.1 The language

The language that we shall use throughout is a simple call-by-value language of recursive, curried function definitions, often known as supercombinator form (Peyton-Jones 1987). It is intended to be a small intermediate language, suitable for representing programs written in purely functional variants of ML or similar languages, and as such we shall refer to this language as the "core" language. The λ-calculus can be embedded into this language via λ-lifting (Johnsson 1985). As the language is largely standard, for space reasons we shall not give a full definition, but merely illustrate the language with the example program in Figure 1. As this shows, the core language includes pattern matching and arithmetic operations. The language allows several kinds of values: *algebraic* values such as lists, trees and other user-defined datatypes; function values and primitive constants such as integers and booleans. The set of constants is largely arbitrary but must be well-founded, and we choose:

$$\mathit{Constant} = \mathbb{N} \cup \mathbb{B}$$

Nested function definitions are not permitted in the core language, and we eschew many of the complexities of ML-like languages,

```
map f xs =
  match xs with
    [] → []
  | y::ys → f y :: map f ys

compose f g x = f (g x)

succ n = n + 1

map (compose succ succ) [1;2;3]
```

Figure 1. The Core Language: Example Program

but the purely functional core of ML can be translated into our core language.

The semantics of the language is defined in big-step operational style. The evaluation relation $s \Downarrow v$ denotes that a *state* s evaluates to a *value* v. Each value is a state, but not conversely — values represent states that need no further evaluation.

There are three kinds of values, namely primitive constants, algebraic constants and closures. This is described below in pseudo-BNF form:

$$\begin{array}{lcl} \mathit{Value} & ::= & \mathit{Constant} \\ & | & \mathsf{Constructor}(\mathit{Value}^*) \\ & | & (\!|f : \rho|\!) \\ & & \text{where } f \in \mathit{FunctionName} \wedge \rho \in \mathit{Environment} \\ & & \text{and } \operatorname{dom}\rho \subsetneq \operatorname{params}(f) \end{array}$$

An algebraic constant is of the form $\mathsf{C}(v_1, \ldots, v_n)$, where C is a constructor symbol and the v_i are values. We write list constants using the usual abbreviations $[]$ and $v_1 :: v_2$ (a list with head value v_1 and tail value v_2).

A closure $(\!|f : \rho|\!)$ is a function name f together with an environment ρ, where the domain of ρ is a strict subset of the parameters of f. The environment binds all the parameters of f which have already been applied to values, and hence this closure represents a function with parameters those parameters of f that are *not* in the domain of ρ. As an example, with the usual *map* function, $(\!|\mathit{map} : \emptyset|\!)$ is a function of two arguments, while $(\!|\mathit{map} : \{f \mapsto (\!|\mathit{id} : \emptyset|\!)\}|\!)$ is a function of one argument.

States include all values, and in addition contain: "complete" closures of the form $(\!|f : \rho|\!)$, where $\operatorname{dom}\rho$ is the set of parameters of f, and expression states of the form $e : \rho$:

$$\begin{array}{lcl} \mathit{State} & ::= & \mathit{Value} \\ & | & (\!|f : \rho|\!) \\ & & \text{where } f \in \mathit{FunctionName} \wedge \rho \in \mathit{Environment} \\ & & \text{and } \operatorname{dom}\rho = \operatorname{params}(f) \\ & | & \mathit{Expression} : \mathit{Environment} \end{array}$$

The first case represents a fully applied function, which must be evaluated, while the second represents an intermediate state in the evaluation of a function body. For instance,

$$(\!|\mathit{map} : \{f \mapsto (\!|\mathit{id} : \emptyset|\!), \mathit{xs} \mapsto []\}|\!)$$

is a state representing an application of *map* that requires evaluation. While complete closures use the same notation as proper closures (values), the domain of the environment can be used to separate the two cases.

The semantics is presented in Figure 2. We shall not comment on this further as it is largely standard, in the style of the semantics of ML (Milner et al. 1997). For completeness we note that the given

Values, Variables and Constants

$$\frac{}{v \Downarrow v}\ v \text{ is a value} \qquad \frac{}{x : \rho \Downarrow \rho(x)} \qquad \frac{}{c : \rho \Downarrow c}$$

Primitive Operators

$$\frac{(\forall i) e_i : \rho \Downarrow c_i \quad \overline{op}(c_1, \ldots, c_n) = c}{op(e_1, \ldots, e_n) : \rho \Downarrow c}\ (\forall i) c_i \text{ is a constant}$$

Constructors

$$\frac{(\forall i) e_i : \rho \Downarrow v_i}{\mathsf{C}(e_1, \ldots, e_n) : \rho \Downarrow \mathsf{C}(v_1, \ldots, v_n)}$$

Function Application

$$\frac{e_1 : \rho \Downarrow (\!|f : \mu|\!) \quad e_2 : \rho \Downarrow w \quad (\!|f : \mu|\!) \oplus w \Downarrow v}{e_1 e_2 : \rho \Downarrow v}$$

Function References and Closures

$$\frac{}{f : \rho \Downarrow (\!|f : \emptyset|\!)} \qquad \frac{\text{body}(f) : \rho \Downarrow v}{(\!|f : \rho|\!) \Downarrow v}\ \text{dom } \rho = \text{params}(f)$$

Conditionals

$$\frac{e_g : \rho \Downarrow \textbf{true} \quad e_t : \rho \Downarrow v}{\textbf{if } e_g \textbf{ then } e_t \textbf{ else } e_f : \rho \Downarrow v} \qquad \frac{e_g : \rho \Downarrow \textbf{false} \quad e_f : \rho \Downarrow v}{\textbf{if } e_g \textbf{ then } e_t \textbf{ else } e_f : \rho \Downarrow v}$$

Pattern Matching

$$\frac{e : \rho \Downarrow \mathsf{C}_l(v_1, \ldots, v_{n_l}) \quad e_l : \rho \oplus \{x_j^l \mapsto v_j\}_{j=1}^{n_l} \Downarrow v}{\textbf{match } e \textbf{ with } \left\langle \mathsf{C}_i(x_1^i, \ldots, x_{n_i}^i) \to e_i \right\rangle_{i=1}^{k} : \rho \Downarrow v}$$

Notation:

- body(f) is the body of function f, and params(f) is the set of its parameters
- $\overline{op}$ is the evaluation function for an operator *op* (so $\overline{+}$ is the addition function)
- $(\!|f : \mu|\!) \oplus v = (\!|f : \mu \oplus \{x \mapsto v\}|\!)$, where x is the first parameter of f not bound in μ.

Figure 2. Semantics of the Core Language

semantics excludes rules for evaluating "error" states (for instance, pattern matching failure), but that these are straightforward.

3.2 Dynamic and Static Call Graphs

The semantics that we have presented previously is useful as a concise definition of the language, but it is not directly possible to use the SCT analysis on this semantics. This is because nontermination is not explicitly represented in the big-step style. In this semantics (augmented with error rules), evaluation of a state s is nonterminating iff it is not the case that $s \Downarrow v$ for some v. This implicit negative characterisation is inconvenient, and gives no handle on an infinite series of events for nonterminating evaluations. As the SCT method operates by proving that such infinite execution sequences are impossible, this is a serious obstacle.

To remedy this issue, we introduce the dynamic call relation $\Rightarrow$. This is a relation between states, and the intended meaning of the judgement $s \Rightarrow s'$ is that evaluation of state s necessarily depends on evaluation of state s'. This does not imply that $\Rightarrow$ is a small-step reduction relation, as we shall see below, as $\Downarrow$ is not related to the transitive closure of $\Rightarrow$.

The definition of the $\Rightarrow$ relation is given in Figure 3. Informally, $s \Rightarrow s'$ whenever the evaluation of s' appears as a premiss of the rule that applies to the evaluation of s. For instance, the function application rules shown in Figure 3 reveal that a function application state $e_1 e_2 : \rho$ first calls the operator state $e_1 : \rho$, and provided this evaluates to a closure, calls the operand state $e_2 : \rho$. Finally, if both evaluations terminate without errors, there is a final call to the callee function. This last call is the only proper function call, while the first two calls reflect control dependencies. Furthermore, the calls to operator and operand states are not reduction steps, as they do not preserve the values of states.

The $\Rightarrow$ relation can be systematically defined for any big-step operational semantics. We assume that two properties of the semantics hold: it should be *deterministic*, so that only one rule may apply to a state, and *total*, so that some rule applies to each state. Note that totality requires the addition of error rules to avoid stuck states, for instance for ill-typed expressions. Given these two rules, the following holds:

> $s \Downarrow v$ for some v iff there is no infinite sequence of calls $s \Rightarrow s_1 \Rightarrow s_2 \Rightarrow \cdots$

This relation is thus sufficient to prove termination by the SCT principle: if we show that infinite sequences of calls are impossible, then each state evaluates to a result (possibly an error value). The converse implication guarantees that no precision is lost by restricting our attention to the $\Rightarrow$ relation when proving termination.

The Static Call Graph The dynamic call graph, defined by the $\Rightarrow$ relation is of course not computable, as otherwise the termination problem would be decidable. We must hence approximate the dynamic call graph, leading to the *static* call graph. The nature of this approximation is not fixed (unlike in Jones and Bohr (2004)), and the choice of approximation has an important impact on the precision of the analysis, so we shall defer its definition to the next section. However, we give the requirements for call graph constructions here.

We require a finite set State^α of abstract states, where each abstract state represents a set of concrete states. The correspondence between abstract and concrete states is defined by an abstraction function:

$$\alpha : \mathit{State} \Rightarrow \mathit{State}^\alpha$$

In addition, the set State^α carries a partial order $\sqsubseteq$, such that $s \sqsubset s'$ whenever s is more defined than s', that is represents a strict subset of the states represented by s'. For any state s, $\alpha(s)$ should be the most precise (least in the $\sqsubseteq$ order) state representing s. As a result, the set of states represented by an abstract state can be deduced, and defines the concretisation function

$$\gamma(s) = \{t \in \mathit{State} \mid \alpha(t) \sqsubseteq s\}$$

The static call graph approximates the dynamic call graph, and is defined by the abstract equivalent of the $\Rightarrow$ relation, following the usual abstract interpretation definition of safety. Namely, the call graph is sound provided

Primitive Operators

$$\frac{(\forall j \le i)\ e_j : \rho \Downarrow c_j}{op(e_1, \ldots, e_n) : \rho \Rightarrow e_{i+1} : \rho}\ {}^{i<n}_{(\forall j \le i) c_j \text{ is a constant}}$$

Closures

$$\frac{}{(\!|f : \rho|\!) \Rightarrow \text{body}(f) : \rho}\ {}^{\text{dom } \rho = \text{params}(f)}$$

Constructors and Pattern Matching

$$\frac{(\forall j \le i)\ e_j : \rho \Downarrow v_j}{\mathsf{C}(e_1, \ldots, e_n) : \rho \Rightarrow e_{i+1} : \rho}\ {}^{i<n} \qquad \frac{}{\textbf{match } e \textbf{ with } \left\langle \mathsf{C}_i(x_1^i, \ldots, x_{n_i}^i) \to e_i \right\rangle_{i=1}^{k} : \rho \Rightarrow e : \rho}$$

$$\frac{e : \rho \Downarrow \mathsf{C}_l(v_1, \ldots, v_{n_l})}{\textbf{match } e \textbf{ with } \left\langle \mathsf{C}_i(x_1^i, \ldots, x_{n_i}^i) \to e_i \right\rangle_{i=1}^{k} : \rho \Rightarrow e_l : \rho \oplus \{x_j^l \mapsto v_j\}_{j=1}^{n_l}}$$

Conditionals

$$\frac{}{\textbf{if } e_g \textbf{ then } e_t \textbf{ else } e_f : \rho \Rightarrow e_g : \rho} \qquad \frac{e_g : \rho \Downarrow \textbf{true}}{\textbf{if } e_g \textbf{ then } e_t \textbf{ else } e_f : \rho \Rightarrow e_t : \rho} \qquad \frac{e_g : \rho \Downarrow \textbf{false}}{\textbf{if } e_g \textbf{ then } e_t \textbf{ else } e_f : \rho \Rightarrow e_f : \rho}$$

Function Application

$$\frac{}{e_1 e_2 : \rho \Rightarrow e_1 : \rho} \qquad \frac{e_1 : \rho \Downarrow (\!|f : \mu|\!)}{e_1 e_2 : \rho \Rightarrow e_2 : \rho} \qquad \frac{e_1 : \rho \Downarrow (\!|f : \mu|\!) \quad e_2 : \rho \Downarrow w}{e_1 e_2 : \rho \Rightarrow (\!|f : \mu|\!) \oplus w}$$

Figure 3. Semantics: Dynamic Call Graph

> Whenever $s \Rightarrow s'$ in the dynamic call graph, then if $t \sqsupseteq \alpha(s)$, there exists $t' \sqsupseteq \alpha(s')$ such that $t \Rightarrow^{\alpha} t'$ in the static call graph.

It is convenient to further require that the abstract interpretation define an approximation $\Downarrow^{\alpha}$ of the evaluation judgement, obeying the equivalent axiom.

In Section 4, we shall give three different instantiations of the set $State^{\alpha}$, varying in complexity and precision, before evaluating the resulting analyses in Section 5.

3.3 Tracking Dataflow

The static call graph alone is not sufficient to prove termination in the SCT framework. As we have described previously it is further necessary to annotate each edge in the static call graph with a *size-change graph* describing the dataflow in a call, as well as the changes in sizes of values between the caller and the callee. In this section we define the order on values (more generally on arbitrary states), and define a general procedure for computing size-change graphs independently of the static analysis used.

The order on values is constructed from two sources: a given order $<^c$ say on primitive constants, and a structural order on all other values. More precisely, we define the order as follows, by recursion on the right-hand side value or state:

$$\begin{aligned} c < c' &\iff c <^c c' \quad \text{for constants } c, c' \\ w < \mathsf{C}(v_1, \ldots, v_n) &\iff (\exists i)\ w \le v_i \\ w < (\!|f : \rho|\!) &\iff (\exists x)\ w \le \rho(x) \\ w < e : \rho &\iff (\exists x)\ w \le \rho(x) \end{aligned}$$

This definition treats algebraic values and closures in the same way, which certainly has the advantage of simplicity. We shall later see how the order on closures turns out to be useful in proving termination.

Environment Paths In the first-order case, size-change graphs track dataflow and size relationships between function parameters. It is possible to restrict oneself to the same abstraction in the higher-order case also, but this proves limiting. Indeed, in the presence of higher-order functions environments are no longer flat, and values stored in the environment may themselves be functions with parameters[1]. We are therefore moved to give a more general definition of the subparts of the state, which we represent as *environment paths*. An environment path is a string of variable names representing a path from the root of the tree representing a state s to a substate of s. Formally,

Definition 1. The set of *environment paths* of a state s, called the *graph basis* of s, is defined as follows:

$$\begin{aligned} \text{gb}(c) &= \{\epsilon\} \quad \text{for a constant } c \\ \text{gb}(\mathsf{C}(v_1, \ldots, v_n)) &= \{\epsilon\} \cup \bigcup_{i=1}^{n} \{x_i p \mid p \in \text{gb}(v_i)\} \\ \text{gb}((\!|f : \rho|\!)) &= \{\epsilon\} \cup \bigcup_{x \in \text{dom } \rho} \{xp \mid p \in \text{gb}(\rho(x))\} \\ \text{gb}(e : \rho) &= \{\epsilon\} \cup \bigcup_{x \in \text{dom } \rho} \{xp \mid p \in \text{gb}(\rho(x))\} \end{aligned}$$

where concatenation of strings is denoted by juxtaposition, so that xp is the path p preceded by the variable x. In the constructor case, dummy variable names $x_1, \ldots, x_n$ are used to identify constructor parameters.

For a complete closure state $(\!|f : \rho|\!)$, the parameters of f can be found as strings of length one, so this is a (strict) generalisation of the use of function parameters in the first-order case.

Definition 2. The substate of a state s at environment path p is denoted by $s \nabla p$, and is the subtree of s at path p from the root:

$$\begin{aligned} s\nabla\epsilon &= s \\ \mathsf{C}(v_1, \ldots, v_n)\nabla x_i p &= v_i \nabla p \\ (\!|f : \rho|\!)\nabla xp &= \rho(x)\nabla p \\ e : \rho \nabla xp &= \rho(x)\nabla p \end{aligned}$$

[1] Environments need not be flat in the first order case in the presence of algebraic values, and this limitation also applies in the first-order case, but was not previously noted.

Dataflow in the Dynamic Call Graph As a useful stepping stone to defining size-change graphs for the static call graph, we define an instrumentation of the dynamic call graph to record dataflow information alongside evaluations and calls. We shall not be concerned with size changes yet, but merely to account for the effect of each evaluation step on data. We shall then use this to define size-change graphs in the static call graph. Note that while we are chiefly concerned with dataflow along calls (the $\Rightarrow$ relation), it is convenient to define dataflow instrumentations for both calls and evaluations (the $\Downarrow$ relation).

Definition 3. Let s and s' be states. A *dataflow graph* G for (s, s') is a relation $G \subseteq \text{gb}(s) \times \text{gb}(s')$. The graph G is *safe* for (s, s') if whenever $(p, q) \in G$, $s \nabla p = s' \nabla q$.

Whenever $(p, q) \in G$, we write $p \xrightarrow{G} q$. The instrumentation of the dynamic call graph defines new judgements $s \Rightarrow s', G$ and $s \Downarrow v, G$. The intended meaning is that G is safe, and thus accurately describes dataflow, for (s, s') (resp. (s, v)).

Inspection of the semantics in Figures 2 and 3 reveals that there are four rules giving rise to nontrivial dataflow: variable lookup, function application, pattern matching and constructor application. Furthermore, the last two operations are direct equivalents of the first two, as closures and algebraic values share the same structure. It thus suffices to describe dataflow for variable lookup and function application.

To illustrate these cases, we shall use the following running example. We consider the *apply* function:

```
apply f x = f x
```

and describe two stages of its evaluation in the following environment:

$$\sigma = \{f \mapsto (\!|\textit{apply} : \{f \mapsto (\!|\textit{id} : \emptyset|\!)\}|\!), x \mapsto 0\}$$

The first step of its evaluation is to find the value of f in the environment, that is to say to evaluate the state $s_f = f : \sigma$. Evaluation of s_f will illustrate variable lookup. To illustrate function application, we consider evaluation of the state $s_{\text{app}} = fx : \sigma$, which occurs after both the operator f and the operand x have been evaluated.

- Variable Lookup

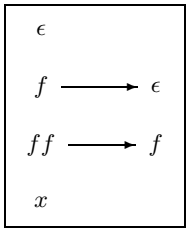
Consider a state $s = x : \rho$. Then $s \Downarrow v = \rho(x)$. The dataflow in this state transition is straightforward: the value bound to x (that is, at environment path x) in s is just v. Therefore:

Lemma 4. *The graph* $\{x : p \to p \mid p \in gb(v)\}$ *is safe for* (s, v).

For instance, consider the state $s_f = f : \sigma$ from our running example. Then

$$s_f \Downarrow v_{\text{fun}} = \sigma(f) = (\!|\textit{apply} : \{f \mapsto (\!|\textit{id} : \emptyset|\!)\}|\!)$$

and the the dataflow graph for this evaluation judgement is shown below:



- Function Application

Let s be a state of the form $s = e_1 e_2 : \rho$, and let $s_1 = e_1 : \rho$ and $s_2 = e_2 : \rho$. Suppose further that $s_1 \Downarrow s_b$, where $s_b = (\!|f : \mu|\!)$ with G_1 safe for (s_1, s_b); and that $s_2 \Downarrow w$ with G_2 safe for (s_2, w).

Evaluation of s then requires evaluation of the state $s' = (\!|f : \mu \oplus \{x \mapsto w\}|\!)$, where x is the first parameter of f not in $\text{dom}\,\mu$. The dataflow in the state transition (s, s') is given by the graph defined below.

Lemma 5. *The graph:*

$$\{p \to q \in G_1 \mid p, q \neq \epsilon\} \cup \{p \to x : q \mid p \xrightarrow{G_2} q \wedge q \neq \epsilon\}$$

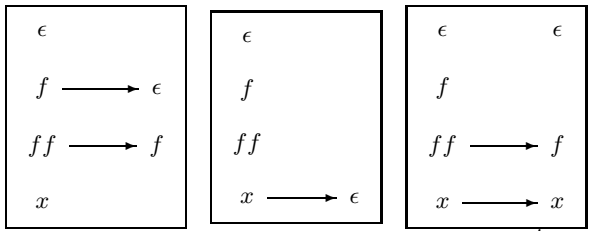
is safe for (s, s').

The interpretation of this graph is as follows: the first component reflects dataflow to parameters of the callee that have already been bound, while the second component describes dataflow to the newly bound value of x. The exclusion of the ϵ path reflects the fact that G_1 and G_2 are dataflow graphs from states s_1 and s_2 (resp.), and that while the environments of these states are the same as that of s, the s_1 and s_2 are not equal to s.

Let us illustrate the function application case with our running example, this time considering the state $s_{\text{app}} = fx : \sigma$. Evaluation of this state proceeds as follows. The operand is evaluated first, which is just the judgement $s_f \Downarrow v_{\text{fun}}$ presented in the variable lookup case. Then, the operand $s_x = x : \sigma$ is evaluated to 1. Finally, $s_{\text{app}} \Rightarrow s'$, where s' is the state for the callee:

$$s' = (\!|\textit{apply} : \{f \mapsto (\!|\textit{id} : \emptyset|\!), x \mapsto 1\}|\!)$$

The dataflow graphs for these three operations (operator evaluation, operand evaluation and transition to the callee function) are shown below:



$s_f \Downarrow s_{\text{fun}}$ $\quad$ $s_x \Downarrow 1$ $\quad$ $s_{\text{app}} \Rightarrow s'$

Abstract Interpretation The dataflow graphs provide all the necessary information to define size-change graphs in the static call graph, and thus apply the SCT method to prove termination. The one exception is the construction of size-change graphs for primitive operators, such as operators on numbers, which may handled as in first-order SCT analysis (Lee et al. 2001; Frederiksen 2001). In this section we define size-change graphs in the static call graph, and show how these may be deduced from dataflow graphs.

We first observe that as an abstract state can represent several concrete states, the graph bases of these states may vary and therefore may not be known. We must therefore approximate the graph basis of an abstract state:

Definition 6. Let t be an abstract state. The *graph basis* $\text{gb}^\alpha(t)$ of t is a set of environment paths such that $\text{gb}^\alpha(t) \subseteq \text{gb}(s)$ whenever $s \in \gamma(t)$.

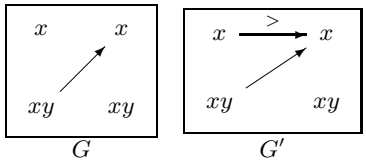
A size-change graph is then defined as annotating a pair of *abstract* states, giving both dataflow and size-change information:

Definition 7. A *size-change graph* for abstract states (t, t') is a labelled relation $G \subseteq \text{gb}^\alpha(t) \times \{\geq, >\} \times \text{gb}^\alpha(t')$. This is *safe* for (t, t') iff whenever $p \xrightarrow{>} q$ (resp. $p \xrightarrow{\geq} q$) in G, then for any states $s \in \gamma(t)$, $s' \in \gamma(t')$, $s \nabla p > s' \nabla q$ (resp. $s \nabla p \geq s' \nabla q$).

In other words, the information encoded in the size-change graph should accurately describe any states that the abstract states might represent. Apart from the move to abstract states, the crucial difference with dataflow graphs is the presence of size change information (edges labelled $>$). However, this is not as substantial

a difference as might appear at first sight, as these may be deduced from dataflow graphs. The key is the observation that dataflow between nested environment paths interacts with the subtree order for values.

For instance, consider the graph basis $A = \{x, xy\}$ and let G be the graph on A with the single arrow $(xy, \geq, x)$. This graph contains no decrease labels. However, consider any (s, s') for which G is safe. Then $s'\nabla x \leq s\nabla xy$. Now $v = s\nabla xy = (s\nabla x)\nabla y$, so v is a strict substate of $w = s\nabla x$. As our order on states is the subtree order, we deduce that $v < w$. Putting all of this together, $s'\nabla x \leq s\nabla xy = v < w = s\nabla x$. We may conclude that the graph G' is also safe for (s, s'), where G and G' are shown below:



Generalising this, we may complete a dataflow graph with size-change information as follows:

Definition 8. Let G be a size-change graph. Then the *completion* of G is:

$$\begin{aligned}\overline{G} \quad = \quad & G \cup \{(p, >, q) \mid (\exists l)\ (pr, l, q) \in G \wedge r \neq \epsilon\} \\ & \cup \{(p, >, qr) \mid r \neq \epsilon \wedge qr \in B \wedge (\exists l)\ (p, l, q) \in G\}\end{aligned}$$

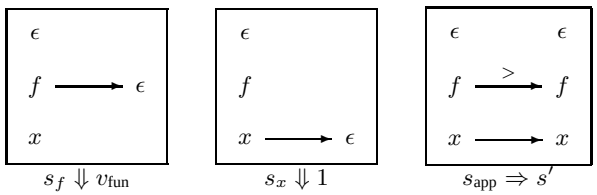
where as before pq denotes the concatenation of p and q.

The completion operator on graphs allows us to recover information about sizes from dataflow graphs. To complete the picture, we must account for the fact that the full graph basis of an abstract state is not known (and is not well-defined, as an abstract state encodes several distinct concrete states). To attack this problem, we assume that for each abstract domain there is a *depth* k such that whenever $s \in \gamma(t)$, then $\text{gb}^\alpha(t)$ is just the set of environment paths of length at most k in $\text{gb}(s)$. In particular, all states represented by t have the same graph basis up to a certain depth. A natural choice is $k = 1$, giving states with the same function parameters, but we shall give analyses with arbitrarily large values of k. Given this assumption, we define:

Definition 9. Let G be a dataflow or size-change graph. The *k-restriction* of G is the set of tuples in G such that both environment paths have length at most k.

It is now straightforward to define the size-change graph we wish to construct for each edge in the static call graph. Each edge in the static call graph is constructed with an abstract equivalent of one of the rules in the concrete semantics (for instance, the function application rule). For each such abstract rule, the graph annotating the edge is obtained by the *same* construction as in the exact semantics, but applying both completion to recover size-change information, and k-restriction to ensure soundness.

To illustrate k-restricted size-change graphs, let us return to the running example illustrating dataflow graphs shown in the previous section. We have shown the size-change graphs generated in the evaluation of the state $s_{\text{app}} = fx : \sigma$ in the exact semantics. Below, we give the equivalent 1-limited graphs, after completion:



The key difference with the exact graphs is the arrow $f \xrightarrow{\geq} f$ in the last graph. This is obtained by completion from the previous arrow $ff \rightarrow f$: the value of f in the callee state is a proper substate of the caller's value for f, and hence is smaller under the subtree order. This example further illustrates the point of the subtree order for closures: it registers decreases in value when self-loops are generated by the application of a function parameter, as in this case. Indeed, in the call from *apply* to itself, the depth of the closure passed to *apply* decreases, so even though this is a recursive call, it is guaranteed to terminate.

With this our overview of the generation of size-change graphs is complete. It is important to note that as size-change graphs are constructed (by rules such as the application rule illustrated above) from k-restricted graphs, some precision is lost with respect to the annotated dynamic semantics. This is natural in the context of static analysis, however, and is necessary to maintain a computable approximation.

4. Control-flow Analysis

The choice of static analysis (essentially determined by the set State^α of abstract states) is a crucial factor in the precision of the resulting termination analysis. In this section, we define three call graph constructions, and show how they fit in the higher-order SCT method: the first is the well-known 0CFA analysis (Shivers 1991), the second is a family of analyses which we refer to as k-limited CFA (distinct from k-CFA (Shivers 1991)) and finally we introduce a new method for call graph construction based on the use of tree automata.

4.1 Common Framework

The three analyses we describe share many common aspects, and making these aspects explicit clarifies the issues underlying control-flow analysis in our setting. Each analysis aims to restrict the set of abstract states to a finite set. In order to understand the means of achieving this, one must note that the set of concrete states is infinite for two essentially distinct reasons: first, states may contain constants, and there are infinitely many of these; second, the environments of states may have unbounded depth. In keeping with our focus on higher-order issues we shall concentrate on the second problem. As such, the problem of approximating states reduces to approximating *environments* by a finite set. For this reason, in every abstract interpretation each abstract state will be represented as a pair p/ρ of a *program point* and an *abstract environment*:

$$\mathit{State}^\alpha = \mathit{ProgramPoint} \times \mathit{Environment}^\alpha$$

A program point is one of:

1. an expression e,
2. a constructor C, or
3. a closure representation $(\!|f : S|\!)$, where S is a subset of the parameters of f (the variables that are already bound).

In closure program points $(\!|f : S|\!)$, the set of bound variables is included to distinguish partial function applications: $(\!|\mathit{map} : \emptyset|\!)$ is a function of two arguments, while $(\!|\mathit{map} : \{f\}|\!)$ is a function of one argument. Note that thanks to this choice, for each program point any abstract state with this program point is unambiguously a value or a proper state: e is always a state, C always a value, and $(\!|f : S|\!)$ is a value iff $S \subsetneq \text{params}(f)$. This goes further, and in fact the 1-limited graph basis of a state (for a function, this is just the set of its parameters) is entirely fixed by its program point.

There are only two operations on environments, given below:

$$\mathit{lookup} : \mathit{Environment} \to \mathit{Variable} \to \mathit{Value}$$
$$\mathit{lookup}\ \rho\ x = \rho(x)$$

$$\mathit{bind} : \mathit{Environment} \to \mathit{Variable} \to \mathit{Value} \to \mathit{Environment}$$
$$\mathit{bind}\ \rho\ x\ v = \rho \oplus \{x \mapsto v\}$$

These two operations encode all fundamental manipulations of environments: adding values to the environment and retrieving them. It now suffices to define the abstract equivalent of each of these to define the abstract interpretation (call graph construction) for any choice of abstract domain. The types of the abstract operations are as follows:

$$\begin{aligned} \mathit{lookup}^\alpha &: \mathit{Environment}^\alpha \to \mathit{Variable} \to \mathbb{P}(\mathit{Value}^\alpha) \\ \mathit{bind}^\alpha &: \mathit{Environment}^\alpha \to \mathit{Value}^\alpha \to \mathit{Environment}^\alpha \end{aligned}$$

We will also use an operation derived from bind^α to simultaneously bind several variables without comment.

The types of these operations reflect the fact that each abstract value has a single program point, so that only the environment may be partially unknown. The lookup^α operator may therefore return a set of values with distinct program points. Formally, the abstract interpretation is defined on the Hoare powerdomain of State^α.

Given these operations only, it is possible to define the abstract interpretation. This is shown in Figure 4, where we use the notation $\Downarrow$ rather than $\Downarrow^\alpha$ for the abstract evaluation relation. The call relation is left to the companion report, as its derivation from the exact call relation of Figure 3 is similar to that of the abstract evaluation relation.

With this framework in place, it now suffices to define our instantiations for $\mathit{Environment}^\alpha$, to define the analyses that we discuss. As shown previously, the size-change graph constructions can be derived from dataflow annotations of the dynamic call graph, and so these are not included in the analyses.

4.2 0CFA

0CFA (Shivers 1991) is the simplest possible instantiation of our framework: no information at all is stored about environments, so the set $\mathit{Environment}^\alpha$ is a dummy one-point set $\{\bullet\}$. 0CFA is thus a context-insensitive analysis: each program point (such as the body of a function) is given only one node in the static call graph, and no distinction is made between different occurrences of the same program point. The abstract graph basis of a state is just its 1-limited graph basis, which is given by the program point.

The definition of the bind^α function is trivial, as there is only one abstract environment. Defining the lookup^α function is more involved, however, as no information is available from the environment to give the value of a variable. This is solved using *closure analysis*, which relies on the assumption that the whole program is being analysed. Given this assumption, a value v may only appear bound to a variable x in an environment if there is some appropriate binding site in the program P. Suppose that x is a parameter of function f, and that the set of parameters of f that appear before x is S. Then the only binding sites for x are those of the form e_1e_2, where e_1 evaluates (in 0CFA) to a value with program point $(\!|f : S|\!)$. We therefore define:

$$\mathit{lookup}^\alpha\ \rho\ x = \{w \mid (\exists e_1e_2 \in P)\ \ e_1/\bullet \Downarrow^\alpha (\!|f : S|\!)/\bullet\ \wedge\ e_2/\bullet \Downarrow^\alpha w\}$$

0CFA is well-documented, and was used in the first higher-order SCT analysis (Jones and Bohr 2004). As such, we shall not describe it further, and in particular do not present the standard proof of soundness of closure analysis, but rather give an example showing that it is frequently insufficient. Consider the following program:

```
id x = x
k us = map id (1 :: us)
h vs = map k vs
h *
```

where $*$ represents any arbitrary value. This artificial example is a terminating program, and only involves uses of the *map* function, so should be straightforwardly size-change terminating. However, this is not the case in 0CFA. For, the k function calls *map*; in turn *map* calls k (as k is a parameter to *map*). The call graph is shown in Figure 5. However, there is no size-change information in the $k \Rightarrow k$ call (this is the purpose of calling *map* with argument $1 :: \mathit{us}$ in k, as this prevents what is otherwise a decrease in the list value). The imprecision of the 0CFA call graph thus prevents a successful termination proof for even a very simple program, due to the confusion between unrelated instances of *map*. While it took a contrived example to show this in a tiny program, as programs grow larger, precision drops through the many uses of functions such as *map*. This is the motivation for introducing the more precise analyses that follow.

4.3 k-limited CFA

In 0CFA, no information at all was kept about environments, leading to the above problem with uses of the *map* function. In the example we have shown, however, it is easy to separate the two uses of the map function, by the value bound to the f parameter of *map* — one call binds f to k, the other to *id*. This leads to the idea of k-limited CFA: keep a fixed, limited amount of information about environments in the call graph.

In k-limited CFA, we keep the environment (a tree) up to a certain depth k. In particular, in 1-limited CFA each variable is mapped to the program point of its value, but no information is kept about the environments of these values. In general, this leads to a family of abstract values and environments (one set of environments for each k):

$$\begin{aligned} \mathit{Value}_k^\alpha &= \mathit{ProgramPoint} \times \mathit{Environment}_k^\alpha \\ \mathit{Environment}_0^\alpha &= \{\bullet\} \\ \mathit{Environment}_{k+1}^\alpha &= \mathit{Variable} \to \mathit{Value}_k^\alpha \end{aligned}$$

This makes Value_k^α into a set of trees of depth at most k, where some of the leaf nodes may have environment $\bullet$. This special environment (the 0CFA environment) denotes that some information is lost in this state. The abstraction map $\alpha_k : \mathit{State} \to \mathit{State}_k^\alpha$ is readily defined, and as an example, if $s = (\!|\mathit{map} : \{f \mapsto (\!|\mathit{apply} : \{f \mapsto (\!|\mathit{id} : \emptyset|\!)\}|\!), \mathit{xs} \mapsto []\}|\!)$, then

$$\begin{aligned} \alpha_0(s) &= (\!|\mathit{map} : \{f, \mathit{xs}\}|\!)/\bullet \\ \alpha_1(s) &= (\!|\mathit{map} : \{f, \mathit{xs}\}|\!)/\{f \mapsto (\!|\mathit{apply} : \{f\}|\!)/\bullet, \mathit{xs} \mapsto []\} \\ \alpha_2(s) &= s \end{aligned}$$

The k-limited CFA has a useful characteristic in our setting: a k-limited state defines the environment up to depth k, so the graph basis of a state is known for the first k levels of the environment. Furthermore, a leaf node (for instance, $(\!|f : S|\!)$) defines the set of parameters in the environment for this node (in this case, this set is S). The graph basis of a k-limited state is therefore known up to depth $(k + 1)$, and the abstract graph basis function for k-limited CFA is the $(k + 1)$-limited graph basis function. This analysis therefore allows us to track dataflow between values that occur nested in the environment, increasing precision.

The bind_k^α operation (for depth k) is once more easy to define: $\mathit{bind}^\alpha\ \rho\ x\ v$ simply binds v to x in the environment, and applies the above α_k function to restrict the depth of the resulting tree to k.

The lookup function uses ideas similar to 0CFA. Given an abstract k-limited environment ρ, the value $\rho(x)$ of a variable x in ρ is a $(k-1)$-limited value and as such may be less precise than is required. The lookup_k^α therefore proceeds in two steps:

Values, Variables and Constants

$$\frac{}{v \Downarrow v}\ v \text{ is a value} \qquad \frac{v \in \mathit{lookup}^{\alpha}\ \rho\ x}{x/\rho \Downarrow v} \qquad \frac{}{c/\rho \Downarrow c}$$

Function References and Closures

$$\frac{}{f/\rho \Downarrow (\!|f : \emptyset|\!)/\emptyset} \qquad \frac{\mathrm{body}(f)/\rho \Downarrow v}{(\!|f : S|\!)/\rho \Downarrow v}\ S = \mathrm{params}(f)$$

Primitive Operators

$$\frac{(\forall i) e_i/\rho \Downarrow c_i \quad \overline{op}^{\alpha}(c_1, \ldots, c_n) = c}{op(e_1, \ldots, e_n)/\rho \Downarrow c}$$

Conditionals

$$\frac{e_g/\rho \Downarrow c \sqsupseteq \textbf{true} \quad e_t/\rho \Downarrow v}{\textbf{if } e_g \textbf{ then } e_t \textbf{ else } e_f/\rho \Downarrow v} \qquad \frac{e_g/\rho \Downarrow c \sqsupseteq \textbf{false} \quad e_f/\rho \Downarrow v}{\textbf{if } e_g \textbf{ then } e_t \textbf{ else } e_f/\rho \Downarrow v}$$

Constructors

$$\frac{(\forall i) e_i/\rho \Downarrow v_i}{\mathsf{C}(e_1, \ldots, e_n)/\rho \Downarrow \mathsf{C}/\mathit{bind}^{\alpha}\ \emptyset\ \{x_i \mapsto v_1, \ldots, x_n \mapsto v_n\}}$$

Pattern Matching

$$\frac{e/\rho \Downarrow \mathsf{C}_l/\mu \quad (\forall j)\ v_j \in \mathit{lookup}^{\alpha}\ \mu\ x_j \quad e_l/\mathit{bind}^{\alpha}\ \rho\ \{x_j^l \mapsto v_j\}_{j=1}^{n_l} \Downarrow v}{\textbf{match } e \textbf{ with } \left\langle \mathsf{C}_i(x_1^i, \ldots, x_{n_i}^i) \to e_i \right\rangle_{i=1}^{k}/\rho \Downarrow v}$$

Function Application

$$\frac{e_1/\rho \Downarrow (\!|f : S|\!)/\mu \quad e_2/\rho \Downarrow w \quad (\!|f : S \cup \{x\}|\!)/\mathit{bind}^{\alpha}\ \mu\ x\ w \Downarrow v}{e_1 e_2/\rho \Downarrow v}\ x \text{ is the first parameter of } f \text{ not in } S$$

Figure 4. Abstract Interpretation

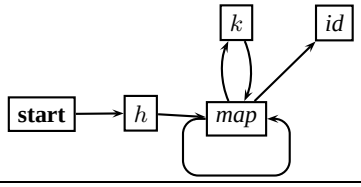


Figure 5. 0CFA Counterexample Program: Call Graph

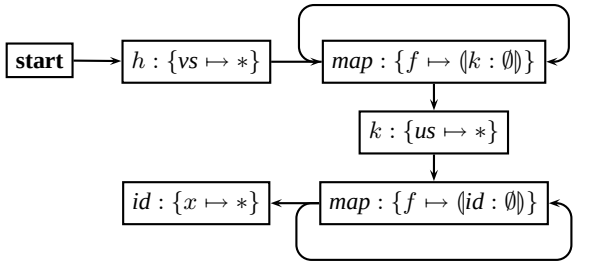


Figure 6. Example Program: 1-limited CFA Call Graph

1. Look up a a value in the environment, yielding a $(k-1)$-limited value, then
2. Find the set of k-limited values that are compatible with this $(k-1)$-limited value.

This second step once more uses closure analysis: closure analysis is applied exactly as in the 0CFA case to the *leaves* of the tree, converting each leaf into a set of possible 1-limited trees. Each possible way of substituting these trees for leaves gives rise to a potential k-limited abstract value. To illustrate this, consider the following 1-limited state:

$$(\!|f : \{h, k\}|\!)/\{h \mapsto (\!|h_1 : \{g_1\}|\!)/\bullet, k \mapsto (\!|h_2 : \{g_2\}|\!)/\bullet\}$$

Suppose further that by closure analysis, there are two potential values for the parameter g_1 and three potential values for g_2. Then there are in total *six* 2-limited trees that are compatible with the above 1-limited tree. Each of these trees is obtained by replacing the environments of h_1 and h_2 with each possible pair of values of g_1 and g_2.

The k-limited CFA analysis is considerably more expensive than 0CFA, as the number of abstract states increases (doubly) exponentially with the limit k. However, this increased precision is indeed useful in distinguishing between otherwise equivalent nodes in the call graph. As an example, let us consider the program given as a failure of 0CFA. The 1-limited call graph for this program is given in Figure 6. The difference between this call graph and the 0CFA call graph (Figure 5) lies in the fact that the 1-limited call graph ends up with *two* nodes for the *map* function, due to the different values of the f parameter. This transforms the call graph into one that is size-change terminating: the only self-loops left are those around the *map* function, which are trivially size-change terminating as *map* is defined by primitive recursion.

4.4 Tree Automata

The k-limited CFA analysis seeks to resolve the problem posed by uses of functions such as *map* and *fold*, where a single higher-order function is called with numerous parameter values, and is a natural way of controlling precision in this context. However, k-limited CFA is not a natural way of increasing precision for programs that create *unbounded* environments. As an example, consider the following function:

```
h n f g = if n = 0 then g
          else compose f (h (n−1) f g)
```

An application $h\ n\ f\ g$ evaluates to the composition $f^n \circ g$, represented in the semantics as a closure of n nested applications of the *compose* function, with g appearing as the second argument of the innermost occurrence of *compose*. The value of g therefore appears at depth n in the environment. As n is in general unknown in static analysis, the k-limited CFA analysis is incapable of distinguishing between two values $h\ n\ f\ g_1$ and $h\ n\ f\ g_2$ whenever the value of n is not fully known. Increasing the value of k does not in this case improve precision, and a program similar to the 0CFA counterexample can be constructed to take advantage of this.

It should be noted that while the above example is contrived, the phenomenon of arbitrarily deep closure values is not rare, in particular in the analysis of lazy programs — lazy programs can involve arbitrarily long chains of suspended computations, represented as unbounded closures.

To remedy this issue with k-limited CFA, we introduce a new analysis, based on the idea of using a restricted set of tree automata to represent environments. The idea is to represent a set of trees as

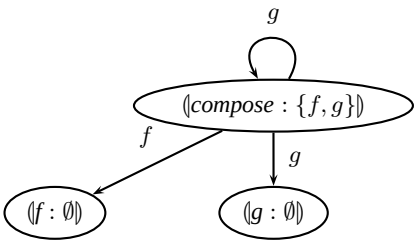


Figure 7. Tree Automata Analysis: Example Value

a possibly cyclic graph with a root by identifying all nodes with the same label. In our context, the label of a node is its program point, so this amounts to identifying all nodes in an environment tree with the same program point. For instance, Figure 7 represents the representation of the value of $h\ n\ f\ g$, for $n > 0$, showing the fact that the value of g is recorded in the abstract state.

More generally, a value in this analysis is a labelled relation *ProgramPoint* $\times$ *Variable* $\times$ *ProgramPoint*, together with a distinguished root node. This expands to a set of trees by following all paths from the root node. The abstraction function is readily defined by collapsing a tree to the minimal directed graph representing it. The graph basis function just gives the 1-limited graph basis of a state, as no definite information is known beyond the first level in the environment.

The definition of the *bind*$^\alpha$ and *lookup*$^\alpha$ operations is of more interest. Let us first consider variable lookup. The possible values of a variable x may be found by finding all the successors of the root node of a state along an edge x:

$$\mathit{lookup}^\alpha\ \rho\ x \quad = \quad \{\rho[v/\text{root}] \mid \text{root}(\rho) \xrightarrow{x} v \in \rho\}$$

where $\rho[v/\text{root}]$ is the graph obtained from ρ by setting node v to be the root of the graph. For efficiency it is desirable to remove unreachable nodes from the result, but this is not necessary for soundness.

To compute the value $\mathit{bind}^\alpha\ \rho\ x\ v$, three steps must be taken:

1. Add a new root to the graph ρ, to record the fact that x is bound. If the previous root was $⦇f : S⦈$, the new root is $⦇f : S \cup \{x\}⦈$.
2. Take the union of the two graphs ρ and v. This merges corresponding nodes, and so naturally can cause a loss of precision.
3. Add an edge labelled x from the root of the new graph to the root of v, to record the binding of x.

The soundness proof of these operations is straightforward but lengthy, and the interested reader is referred to the companion report (Sereni 2006) for details. It is however interesting to compare the k-limited CFA approaches and this graph approach. In k-limited CFA, the environment is kept exactly (up to constants) for a certain depth, but no contextual information is stored beyond this depth. In contrast, this analysis stores information at arbitrary depth, but no information is stored exactly as any two nodes with the same label are merged. While we have argued that k-limited CFA is more natural for programs with bounded environments and that the tree automata analysis is better suited for unbounded environments, this further suggests that these analyses may be incomparable, and that each may offer better precision in certain contexts. The purpose of the next section is to present these claims in a more precise light.

5. Precision of the Analysis

We have in the previous sections introduced the framework for higher-order SCT analysis, and given three instantiations of this framework with control-flow analyses, namely the inexpensive context-insensitive 0CFA analysis, the bounded context k-limited analyses, and the unbounded depth tree automata analysis. These vary wildly in cost: the 0CFA is invariably cheapest, the worst-case cost of k-limited analyses increases doubly exponentially with k (but for fixed k is polynomial in the size of the program), while the tree automata analysis has an exponential worst-case complexity, rarely attained. It is thus crucial to evaluate what is achieved by more precise analyses.

Our aim in this section is to give exact relationships between the sets of programs accepted by each analysis. For each CFA, this is the set of programs such that the constructed call graph is size-change terminating. This set is always a subset of the set $\mathcal{T}$ of terminating programs, but can never equal it by undecidability. Any terminating program not in this set represents a failure of the analysis, and so the aim is that this set should be as large as possible. In this section, we compare the sets 0CFA, kCFA and TA of programs accepted by the 0CFA, k-limited CFA and tree automata analyses (resp.). The final result is the precise hierarchy shown in Figure 8.

5.1 The k-limited CFA Hierarchy

We have argued that k-limited CFA increases precision as k increases. This is indeed the case, and the first part of this result is the following k-limited CFA hierarchy theorem:

$$0\text{CFA} \subseteq 1\text{CFA} \subseteq \cdots \subseteq k\text{CFA} \subseteq (k+1)\text{CFA} \subseteq \cdots$$

The key to establish this result is to show that the $(k+1)$-limited call graph is more precise than the k-limited call graph, in the sense defined below:

Definition 10. Let $\mathcal{A}$ and $\mathcal{B}$ be static call graphs for a program P, defining relations $\Rightarrow^A$, $\Downarrow^A$ and $\Rightarrow^B$, $\Downarrow^B$ respectively, and with abstract state sets A and B.

A map $f : A \to B$ is a *simulation* if whenever $s \Downarrow^A v$ (resp. $s \Rightarrow^A s'$) and $t \sqsupseteq f(s)$, then there exists $w \sqsupseteq f(v)$ (resp. $t' \sqsupseteq f(s')$) such that $t \Downarrow^B w$ (resp. $t \Rightarrow^B t'$). In addition the size-change graph of the edge in $\mathcal{B}$ is a subset of that of the edge in $\mathcal{A}$.

Informally, such a map f takes an edge $s \Rightarrow s'$ in $\mathcal{A}$ to an edge $f(s) \Rightarrow f(s')$ in $\mathcal{B}$, but as some precision may be lost in this embedding it is permitted for the target edge $t \Rightarrow t'$ to be less precise (in the $\sqsupseteq$ order) than $f(s) \Rightarrow f(s')$.

If there is a simulation from $\mathcal{A}$ to $\mathcal{B}$, then each edge in $\mathcal{A}$ is reflected in $\mathcal{B}$, and so $\mathcal{B}$ cannot be more precise than $\mathcal{A}$. In particular, we have the following:

Lemma 11. *Suppose that f is a simulation from $\mathcal{A}$ to $\mathcal{B}$, and that $\mathcal{B}$ is size-change terminating. Then $\mathcal{A}$ is size-change terminating.*

Proof sketch. Suppose that $\mathcal{A}$ is not size-change terminating. Then there must exist a cycle $s_0 \Rightarrow s_1 \Rightarrow \cdots \Rightarrow s_n = s_0$ in $\mathcal{A}$ such that the sequence of size-change graphs γ_i annotating each edge does not give rise to infinite descent. The simulation f allows us to construct a corresponding sequence $t_0 \Rightarrow \cdots \Rightarrow t_n$ in $\mathcal{B}$, where $t_i \sqsubseteq f(s_i)$. Furthermore, each size-change graph in this sequence is a subset of the corresponding graph in $\mathcal{A}$, so this sequence does not give rise to infinite descent and $\mathcal{B}$ is nonterminating. □

This result allows us to compare static analyses for the SCT framework.

Corollary 12. *Suppose that Γ and Γ' are control-flow analyses, and that for each P there is a simulation from the Γ-call graph P^Γ of P to the $P^{\Gamma'}$. Then any program accepted by Γ' is accepted by Γ.*

Proof. If P is accepted by Γ', then $P^{\Gamma'}$ is size-change terminating. As there is a simulation from P^{Γ} to $P^{\Gamma'}$, so is P^{Γ}. Hence Γ accepts P. □

We are now able to show that $(k+1)$CFA $\supseteq$ kCFA. By the corollary it suffices to show that there is a simulation from the $(k+1)$-limited call graph to the k-limited call graph.

Proposition 13. *The map α_k (restricting a state to a tree of depth k) is a simulation from a $(k+1)$-limited call graph to a k-limited call graph.*

The proof of this result is essentially the same as the proof of soundness of k-limited CFA, and is omitted. As a result, we have that kCFA $\subseteq$ $(k+1)$CFA for each k, as required.

This result shows that $(k+1)$-limited CFA is at least as precise as k-limited CFA. Naturally, we want more — we can certainly hope that is the case that $(k+1)$-limited CFA is strictly more expressive. We shall show this in the next section, via a detour through the simply-typed λ-calculus.

5.2 Simple Types and k-limited CFA

The simply-typed λ-calculus represents the best-known set of terminating functional programs. By strong normalisation, each simply-typed λ-expression terminates under call-by-value. It is therefore natural to ask whether the SCT analysis, in one of its forms, can recognise all simply-typed λ-expressions as terminating. Indeed, it was conjectured by Jones (Jones and Bohr 2004) that this was the case under 0CFA. As we shall show, this is not the case, even under k-limited CFA.

Consider first the following (simply-typable) λ-expression:

$$\Lambda_0 = \big(\lambda a.a(\lambda b.a(\lambda cd.d))\big)\big(\lambda e.e(\lambda f.f)\big)$$

We claim that Λ_0 lies in 1CFA but not 0CFA, so that it is not 0CFA-terminating. To justify this, observe that evaluation of Λ_0 proceeds as follows (where we use an equivalent formulation of our semantics for λ-terms, which may be translated into our own framework by λ-lifting (Johnsson 1985)):

	start	(1)
$\Rightarrow$	$(\!\lvert \lambda a : \{a \mapsto (\!\lvert \lambda e : \emptyset \rvert\!)\}\rvert\!)$	(2)
$\Rightarrow$	$(\!\lvert \lambda e : \{e \mapsto (\!\lvert \lambda b : \{a \mapsto (\!\lvert \lambda e : \emptyset \rvert\!)\}\rvert\!)\}\rvert\!)$	(3)
$\Rightarrow$	$(\!\lvert \lambda b : \{a \mapsto (\!\lvert \lambda e : \emptyset \rvert\!), b \mapsto (\!\lvert \lambda f : \emptyset \rvert\!)\}\rvert\!)$	(4)
$\Rightarrow$	$(\!\lvert \lambda e : \{e \mapsto (\!\lvert \lambda c : \emptyset \rvert\!)\}\rvert\!)$	(5)
$\Rightarrow$	$(\!\lvert \lambda c : \{c \mapsto (\!\lvert \lambda f : \emptyset \rvert\!)\}\rvert\!)$	(6)
$\Rightarrow$	$(\!\lvert \lambda d : \emptyset \rvert\!)$	(7)

We note that states (3) and (5) are identical under 0CFA, as they are both closures of the λe expression, that is if s_i is the i^{th} state in this sequence, $\alpha_0(s_3) = \alpha_0(s_5)$. Furthermore, the only safe 1-limited size-change graph for (s_3, s_5) is empty, so no size-change information can be obtained. As a result, the 0CFA call graph for Λ_0 contains a cycle around $\alpha_0(s_3) = \alpha_0(s_5)$, which cannot be proved to terminate.

However, the 1CFA call graph for Λ_0 is acyclic, as the 1-limited abstraction of each state in the above call sequence is distinct. As a result, Λ_0 lies in 1CFA $\setminus$ 0CFA. Note that this is not an artefact of our formulation of the 0CFA analysis — the above argument shows that Λ_0 cannot be recognised as size-change terminating under any CFA with the same abstract domain as our 0CFA.

This result extends beyond 1CFA, and indeed for each k there is a λ-expression in $(k+1)$CFA $\setminus$ kCFA. These can be derived from Λ_0 by noting a general technique for manufacturing programs that defeat k-limited CFA. Define the application function $\mathsf{apply} = \lambda xy.xy$. Then given any application e_1e_2 in a program, transforming this to $\mathsf{apply}\ e_1\ e_2$ has the effect of binding the value of e_1 in the environment of apply before applying it. This increases the depth of intermediate states by one, as the value of e_1 would otherwise have been found at the top level of the environment. By nesting applications of apply it is therefore possible to increase the depth of intermediate states arbitrarily, which may be used to confuse k-limited CFA. The result of applying this technique to our example is given below:

$$\Lambda_k = \big(\lambda ga.a(g^k(\lambda b.a(g^k(\lambda cd.d))))\big)\mathsf{apply}\big(\lambda e.e(\lambda f.f)\big)$$

where $e_1^k e_2$ is syntactically defined by $e_1^1 e_2 = e_1 e_2$ and $e_1^{n+1} e_2 = e_1(e_1^n e_2)$. Once more $\Lambda_k \notin k$CFA, as the nested uses of apply add k levels to the environment, but $\Lambda_k \in (k+1)$CFA.

Before recapitulating our results, it is interesting to consider the entire k-limited CFA hierarchy. Define

$$\exists k\text{CFA} = \bigcup_k k\text{CFA}$$

to be the set of programs recognised by *some* k-limited CFA analysis. This is not obviously computable, but we may wonder whether some analysis might subsume all k-limited CFA analyses. In fact, this is not the case, as a consequence of this result:

Proposition 14. *For each $k \geq 0$, kCFA $\subsetneq$ $(k+1)$CFA. Furthermore, no kCFA contains the simply-typed λ-calculus. Finally, the set $\exists k$CFA contains all terminating closed λ-expressions.*

In particular, each terminating λ-expression lies in some kCFA, and the set $\exists k$CFA is not decidable.

Proof. We have outlined the proofs of much of the result. To show that any terminating λ-expression lies in $\exists k$CFA, note that such an expression has a finite call graph. There is hence a bound d to the depth of states in its call graph. Analysis of this expression in $(d+1)$-limited constructs the exact call graph, which is acyclic as the expression terminates, and so is trivially size-change terminating. □

5.3 Tree Automata and k-limited CFA

We shall conclude our description of the relationships between our analyses by outlining the results relating the tree automata analysis to k-limited CFA analyses. It is not entirely unexpected that this analysis performs at least as well as the context-insensitive 0CFA:

Lemma 15. *TA $\supseteq$ 0CFA.*

Proof outline. The map taking each environment to the dummy environment $\bullet$ is a simulation from a tree automata call graph to a 0CFA call graph, so we may appeal to Corollary 12. □

However, as we have informally argued before, the tree automata analysis does not keep precise environment information at any depth, and so should not be expected to be strictly more expressive than any k-limited CFA for $k > 0$:

Lemma 16. *TA $\not\supseteq$ 1CFA.*

Proof outline. The program:

```
id x = x
omega x = omega x
f x y z = y z

f (f id omega) id 1
```

is terminating under 1-limited CFA, but not the tree automata analysis. The key is that in the state resulting from evaluating *f (f id omega) id*, the second argument of *f* appears bound to *omega* and *id*, and thus as these two uses are not distinguished, *f* is found to potentially apply *omega*. □

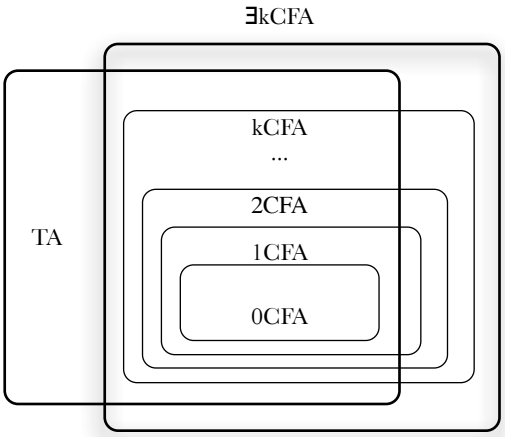


Figure 8. Precision of Static Analyses

Thus all k-limited analyses for $k \geq 1$ accept programs that the tree automata analysis does not. The two analyses are essentially orthogonal, however, and the tree automata analysis accepts programs that require arbitrarily large depth values in k-limited CFA:

Lemma 17. *For each k, TA $\cap$ $((k+1)$CFA $\setminus$ kCFA$)$ is nonempty.*

Proof outline. The expression Λ_k is accepted by the tree automata analysis for each k (proof omitted), whence the result follows. □

The tree automata analysis therefore intersects the k-limited CFA hierarchy at each level, but does not perform better than any fixed k-limited CFA for $k > 0$. To conclude this section, we describe the relationship between the tree automata analysis and the cumulative $\exists k$CFA criterion. As we have shown previously, TA cannot be a superset of $\exists k$CFA. It is in fact incomparable with $\exists k$CFA, though it intersects each kCFA nontrivially:

Lemma 18. *TA $\not\subseteq$ $\exists k$CFA.*

The tree automata analysis thus accepts programs that cannot be handled by any k-limited CFA analysis. The proof of this result is omitted as the corresponding program is lengthy, but may be found in a companion report (Sereni 2006). This completes our discussion of relationships between static analyses, the results of which are summarised in Figure 8.

6. Related Work

The size-change termination criterion was introduced by Lee, Jones, and Ben-Amram (2001), for first-order purely functional (strict) programs, and focusing on the algorithm for deciding termination rather than program analysis. Several extensions have been proposed, for instance to handle non-well founded datatypes (Avery 2006). The extension to higher-order functions was first studied in the context of the pure untyped λ-calculus (Jones and Bohr 2004), implicitly using 0CFA. Sereni and Jones (2005) extend this to a larger higher-order language and introduce the use of k-limited CFA. Our framework is a substantial generalisation of this work, and we further introduce the tree automata CFA for programs with unbounded environments.

Many other termination provers have been proposed, with a strong emphasis on logic programming (for instance, Termilog (Lindenstrauss and Sagiv 1996, 1997) which appears to be closely related to the size-change termination analysis) and term rewriting systems (one of the most recent tools being the AProVE system (Giesl et al. 2005, 2004)). While these settings naturally emphasize first-order programs, and indeed most TRS termination provers are restricted to first-order systems, there have been a number of studies of termination in a higher-order context. In particular, Giesl et al. (2006) show that termination tools for term rewriting systems, and in particular the AProVE system, can fruitfully be applied to termination of (lazy) functional programs, in particular handling higher-order functions. While it is difficult to compare such tools to our framework, in particular because of their reliance on theorem proving, it appears likely that techniques for improving call graph precision would be applicable, and would improve results, in such settings. A different approach to termination analysis is the *transition predicate abstraction* (Podelski and Rybalchenko 2005) technique used by the Terminator tool (Cook et al. 2006a,b). While this has only been used for first-order (imperative) programs, the termination criterion used by Terminator, based on disjunctive well-founded relations, appears similar to size-change termination, suggesting that our results may be applicable.

Control-flow analysis for functional programs has been extensively studied, starting with 0CFA (Shivers 1991, 1988). The kCFA analyses, also due to Shivers, are different from our k-limited CFA analyses in that they use approximations of the call stack, rather than environments, to improve precision. Approximating environments has the benefit of allowing better dataflow information to be recorded, improving precision at the cost of increased complexity. Many other settings for control-flow analysis have been proposed, from the use of game semantics (Malacaria and Hankin 1998) to the approximation of function values, rather than closures, in an abstract interpretation context (Cousot and Cousot 1994, 1991). Our tree automata analysis is related to the flow analysis of Jones (Jones 1987), in which regular tree grammars are used to describe reachable program states in lazy higher-order functional programs. ΓCFA (Might and Shivers 2006) is a promising new technique for improving flow analyses of higher-order programs, but we shall leave the study of this technique in our framework to future work.

Finally, there has been little investigation into the relationship between static analysis and typing. A notable exception is the work of Palsberg and Schwartzbach (1995) on safety analysis of the λ-calculus, where it is shown that 0CFA safety analysis, surprisingly, is strictly more expressive than simple types: any simply-typed term will be accepted by the 0CFA safety analysis. In constrast, no k-limited CFA analysis achieves this for termination in the SCT framework.

7. Conclusion

We have presented a framework for size-change termination analysis of higher-order functional programs, building on previous work adapting this framework to the higher-order setting. This analysis proceeds in two steps, as the call graph of the program is computed first, together with dataflow and size change information, before an existing algorithm can be applied to prove termination.

We set out requirements for the static call graph construction, independently of the gathering of termination information, and have shown three distinct instantiations of this procedure with control-flow analyses: the (known) context-insensitive 0CFA analysis, the family of k-limited CFA analyses, and an analysis based on the use of simple tree automata to represent environments.

The precision of the call graph constructed in the first phase of the analysis emerges as a crucial factor influencing the expressiveness of the resulting termination criterion. In particular, proving termination of programs using the inexpensive 0CFA analysis is likely to fail due to the imprecision of the call graph whenever common functions such as *map* are used repeatedly.

We proposed a characterisation of precision of control-flow analyses, through the study of the expressiveness of the size-change termination analysis obtained with each control-flow analysis. This gives precise relationships between the sets of programs that each analysis accepts. Settling a previous conjecture, we have further clarified the relationship between the simply-typed λ-calculus and SCT analyses. While the use of termination analysis results to evaluate call graph precision may appear limiting, it seems that this corresponds closely to our intuitive notion of precision, mainly through the simplicity of the SCT criterion. It is hence likely that this measure will be a good indicator of precision in other settings.

Avenues of future work include first the study of recently proposed methods for CFA such as Γ-CFA (Might and Shivers 2006) in our framework. This techniques are promising ways to improve precision of CFA, and the question of their relationship to our hierarchy is unknown. A separate area of future work is a more systematic study of the relationship between the simply-typed λ-calculus and SCT or related termination analyses, and in particular determining whether some control-flow analysis can recognise all simply-typed λ-expressions as size-change terminating.

References

James Avery. Size-change termination and bound analysis. In *Proceedings of FLOPS '06*, volume 3945 of *LNCS*, pages 192–207. Springer, 2006.

Josh Berdine, Byron Cook, Dino Distefano, and Peter W. O'Hearn. Automatic termination proofs for programs with shape-shifting heaps. In *Proceedings of CAV '06*, volume 4144 of *LNCS*, pages 386–400. Springer, 2006.

Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Termination proofs for systems code. In *Proceedings of PLDI '06*, pages 415–426. ACM Press, 2006a.

Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Terminator: Beyond safety. In *Proceedings of CAV '06*, volume 4144 of *LNCS*, pages 415–418. Springer, 2006b.

Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of POPL '77*, pages 238–252. ACM Press, 1977.

Patrick Cousot and Radhia Cousot. Relational abstract interpretation of higher-order functional programs. In *Actes JTASPEFL '91*, volume 74 of *Bigre*, 1991.

Patrick Cousot and Radhia Cousot. Higher-order abstract interpretation (and application to comportment analysis generalizing strictness, termination, projection and PER analysis of functional languages), invited paper. In *Proceedings of the IEEE International Conference on Computer Languages (ICLL '94)*, pages 95–112. IEEE Computer Society Press, 1994.

Carl Christian Frederiksen. A simple implementation of the size-change termination principle. Working paper (DIKU, D-442). Available online at `http://www.diku.dk/topps/bibliography/2001.html`, 2001.

Jürgen Giesl, René Thiemann, Peter Schneider-Kamp, and Stephan Falke. Automated termination proofs with AProVE. In *Proceedings of RTA '04*, volume 3091 of *LNCS*, pages 210–220. Springer, 2004.

Jürgen Giesl, René Thiemann, and Peter Schneider-Kamp. Proving and disproving termination of higher-order functions. In *Proceedings of the 5th International Workshop on Frontiers of Combining Systems (FroCoS '05)*, volume 3717 of *LNAI*, pages 216–231. Springer, 2005.

Jürgen Giesl, Stephan Swiderski, Peter Schneider-Kamp, and René Thiemann. Automated termination analysis for haskell: From term rewriting to programming languages. In *Proceedings of RTA '06*, volume 4098 of *LNCS*, pages 297–312. Springer, 2006.

Thomas Johnsson. Lambda lifting: Transforming programs to recursive equations. In *Proceedings of FPCA '85*, volume 201 of *LNCS*, pages 190–203. Springer, 1985.

Neil D. Jones. Flow analysis of lazy higher-order functional programs. In Samson Abramsky and Chris Hankin, editors, *Abstract Interpretation of Declarative Languages*, pages 103–122. Ellis Horwood, 1987.

Neil D. Jones and Nina Bohr. Termination analysis of the untyped λ-calculus. In *Proceedings of RTA '04*, volume 3091 of *LNCS*. Springer, 2004.

Chin Soon Lee, Neil D. Jones, and Amir M. Ben-Amram. The size-change principle for program termination. In *Proceedings of POPL '01*, pages 81–92. ACM Press, 2001.

Naomi Lindenstrauss and Yehoshua Sagiv. Checking termination of queries to logic programs. Available online at `http://www.cs.huji.ac.il/~naomil/`, 1996.

Naomi Lindenstrauss and Yehoshua Sagiv. Automatic termination analysis of logic programs. In *Proceedings of ICLP '97*, pages 63–77. MIT Press, 1997.

Pasquale Malacaria and Chris Hankin. A new approach to control flow analysis. In *Proceedings of CC '98*, volume 1383 of *LNCS*, pages 95–108. Springer, 1998.

Matthew Might and Olin Shivers. Improving flow analyses via ΓCFA: Abstract garbage collection and counting. In *Proceedings of ICFP '06*, pages 13–25, Portland, Oregon, September 2006.

Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, May 1997.

Jens Palsberg and Michael I. Schwartzbach. Safety analysis versus type inference. *Information and Computation*, 118(1):128–141, 1995.

Simon Peyton-Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, 1987. Out of print. Online version available at `http://research.microsoft.com/users/simonpj/papers/slpj-book-1987/`.

Andreas Podelski and Andrey Rybalchenko. Transition predicate abstraction and fair termination. In *Proceedings of POPL '05*, pages 132–144. ACM Press, 2005.

Damien Sereni. *Termination Analysis of Higher-Order Functional Programs*. PhD thesis, Oxford University, 2006. Online version at `http://metacomp.comlab.ox.ac.uk/Members/damien/publications/thesis.pdf`.

Damien Sereni and Neil D. Jones. Termination analysis of higher-order functional programs. In *Proceedings of APLAS '05*, volume 3780 of *LNCS*, pages 281–297. Springer, 2005.

Olin Shivers. Control-flow analysis in Scheme. In *Proceedings of PLDI '88*, pages 164–174. ACM Press, June 1988.

Olin Shivers. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, Carnegie Mellon University, May 1991.