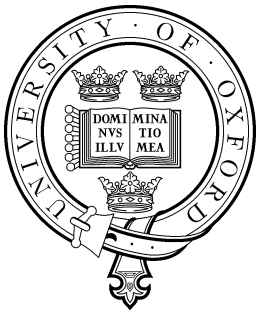
Termination Analysis of Higher-Order Functional Programs

Damien Sereni
Magdalen College

Trinity Term 2006

Submitted in partial fulfilment of the requirements for the degree of Doctor of Philosophy



Oxford University Computing Laboratory
Programming Research Group

Termination Analysis of Higher-Order Functional Programs

Damien Sereni
Magdalen College

D.Phil. Thesis
Trinity Term 2006

Abstract

This thesis concerns fully automatic termination analysis for higher-order purely functional programs, both strict and lazy. We build on existing work on *size-change termination*, in which a program is deemed to terminate if any potential infinite sequence of calls would result in infinite descent in a well-founded data value. This was proposed for strict first-order programs, and a termination analysis of the pure untyped λ-calculus was subsequently obtained in this framework.

We present a generalisation of this work, to handle realistic purely functional programming languages. From our general semantic framework, instances of the termination criterion are derived for both strict (call-by-value) and lazy (call-by-need) and proved sound. It is shown that nontrivial higher-order and lazy programs can be proved to terminate. It is further shown that the analysis of lazy programs requires techniques beyond previous work on size-change termination.

Our analysis proceeds by extracting the call graph of a higher-order program, together with dataflow annotations; termination is then proved by showing that infinite paths in the call graph cannot occur. Precision of the call graph extraction emerges as a crucial factor in the success or failure of termination checking, and we offer several analyses to adjust the tradeoff between precision and complexity. It is shown that existing flow analyses may be applied, and we propose a novel analysis, based on tree automata to represent environments, to improve precision.

As any termination analysis must be incomplete, the issue of determining the precision of the analysis is paramount. The relationship between the sets of programs accepted by the static analyses we present is evaluated, and we further compare size-change termination to known classes of terminating functional programs, in particular the simply-typed λ-calculus.

Acknowledgements

I would first like to express my gratitude to my supervisor Oege de Moor for his support, encouragement and guidance over the years. Since my undergraduate days at Magdalen, Oege has helped me shape my research interests, while giving me every opportunity to explore the field. I could not have asked for a better supervisor, and cannot thank Oege enough for his continuing support after my DPhil.

I am also extremely grateful to Neil Jones for his interest in my work, since his visit to Oxford in 2004. Neil first got me interested in this problem and his continuing involvement with this work has been extremely fruitful. Over the past few years, I have enjoyed discussing my work with Laurie Hendren, whose energy has been a source of inspiration, though I suspect Laurie always thought I should be working in a different area.

Luke Ong and Chris Hankin kindly agreed to be my examiners, and provided many insightful comments during the viva which helped improve this thesis.

The programming tools group in Oxford was an extremely pleasant setting for this work, and thanks in particular to the following for many enjoyable discussions: Yorck Hünke, Rani Ettinger, Ganesh Sittampalam, Duncan Coutts and Mathieu Verbaere. Thanks also to Sam Sanjabi, William Blum, Jolie de Miranda, Richard Bird and Luke Ong for their interest and comments. Many special thanks to Matthew Hague for kindly agreeing to proofread this thesis. Any remaining errors are of course entirely my own. I would also like to express my gratitude to all my fellow basement-dwellers in 002 for helping me enjoy working in the lab for the last three years.

Finally, thanks to everyone who has supported me through the process and put up with me when I was writing up. I could not have completed this DPhil without my family and parents' support and encouragement, and in particular my sister Constance has managed to keep me more or less sane throughout, and has always been there for me through the difficult times.

I cannot do justice to my friends in Oxford in such a short space, but very special thanks to Vicky Kemp, Hélène Pantelli, Fiona Woollard, Jolie de Miranda and Maki Koyama.

I eventually completed this thesis, despite Joe Taylor's best attempts. Thanks.

Contents

Notation

Sets We use standard notation for sets and set operations. In addition, we shall write $\{x_i\}_{i=1}^n$ in lieu of $\{x_i \mid 1 \leq i \leq n\}$. The powerset of a set S is the set of subsets of S, written $\mathbb{P}(S)$.

Sequences We will write the sequence with elements 1, 2 and 3 as $\langle 1, 2, 3\rangle$. The empty sequence is denoted by ε. The concatenation of two sequences s and t is $s +\!\!+ t$. The sequence with first element x and tail s is written $x : s$. We further write hd s for the first element of a (nonempty) sequence. If s is a sequence and S is a set, we write $s \setminus S$ for the sequence obtained by removing all elements of S from s, so that $\mathrm{hd}(s \setminus S)$ is the first element of s not in S. When convenient we shall feel free to regard a sequence as a set, and write for instance $1 \in \langle 1, 2, 3\rangle$.

Functions The set of functions from A to B is $A \to B$, while the set of partial functions from A to B is $A \nrightarrow B$. The set of (partial) functions from A to B with finite domain is written $A \nrightarrow^{\mathrm{fin}} B$.

The domain of a function $f : A \nrightarrow B$ is $\mathrm{dom}\, f \subseteq A$, while its range is $\mathrm{ran}\, f \subseteq B$.

Given a function $f : A \nrightarrow B$ and $x \in A$, $y \in B$, we let $f \oplus \{x \mapsto y\}$ denote the function h defined on $\mathrm{dom}\, f \cup \{x\}$ such that $h(z) = f(z)$ for all $z \in A$, $z \neq x$, and $h(x) = y$.

Common Functions We shall assume familiarity with standard functional programming idioms, defined informally below:

$$
\begin{array}{rcl}
\mathit{apply}\ f\ x & = & f\ x \\
\mathit{compose}\ f\ g\ x & = & f\ (g\ x) \\
\mathit{map}\ f\ \langle x_1, \ldots, x_n\rangle & = & \langle f\ x_1, \ldots, f\ x_n\rangle \\
\mathit{foldr}\ (\oplus)\ e\ \langle x_1, \ldots, x_n\rangle & = & x_1 \oplus (x_2 \oplus \cdots (x_{n-1} \oplus (x_n \oplus e))) \\
\mathit{foldl}\ (\oplus)\ e\ \langle x_1, \ldots, x_n\rangle & = & (((e \oplus x_1) \oplus x_2) \oplus \cdots x_{n-1}) \oplus x_n \\
\mathit{take}\ n\ \langle x_1, \ldots, x_m\rangle & = & \langle x_1, \ldots, x_n\rangle \\
\mathit{zip}\ \langle x_1, \ldots, x_n\rangle\ \langle y_1, \ldots, y_m\rangle & = & \langle (x_1, y_1), \ldots, (x_k, y_k)\rangle \\
\mathit{zipWith}\ f\ \langle x_1, \ldots, x_n\rangle\ \langle y_1, \ldots, y_m\rangle & = & \langle f\ x_1\ y_1, \ldots, f\ x_k\ y_k\rangle \\
 & & \textbf{where}\ k = \min\{n, m\}
\end{array}
$$

These functions extend to infinite lists (in the context of lazy evaluation).

Chapter 1

Introduction

1.1 Program Verification and Termination Analysis

Perhaps one of the most fundamental questions in computer science is: given a program, how can we prove that it is correct? It is well-known that writing correct programs is a difficult task (even simple algorithms can prove tricky — while the first binary search algorithm appeared in 1946, the first correct binary search algorithm was published in 1962). In practice errors in programs are detected by *testing* — the program is run on a variety of inputs, and its actual behaviour is compared to the expected behaviour. While this is an effective way of finding common errors, it is unlikely that infrequently occurring problems can be found in reasonable time by testing. Furthermore, testing a program can never give any assurance that *all* problems have been found (and so that the program is correct). In many cases (safety-critical software, software that would be difficult or impossible to update) it is essential that much stronger assurances be obtained.

1.1.1 Program Verification

This thesis concerns *automatic termination analysis*, which is a special case of the program verification problem. The verification problem may be stated in full generality as follows: we aim to find an automatic procedure that, given any program, verifies whether or not a given property holds of the program (where the property may be fixed or given to the verifier as an input). Properties of interest in verification include:

- Input / output properties: a relationship between the input to the program and its output. For example, $output = input + 1$ is a trivial such property.
- Temporal safety properties: in any execution of the program, the sequence of events must satisfy a given property. An example is the property: any file must be open before it is read from.

- Liveness properties: these are dual to safety properties, and assert that certain events *must* eventually happen in any execution. For instance, a liveness property is: any request for a resource is eventually granted.
- Termination properties. Termination asserts that execution of a program (or part of a program) is necessarily finite. That is, a program terminates if infinite sequences of events are impossible.

In this thesis we shall focus on verifying termination properties.

1.1.2 Termination

The termination analysis problem is perhaps the oldest problem in computer science. It is also the archetypal example of an undecidable problem. Indeed, Turing proves undecidability of the halting (termination) problem in his 1936 paper introducing Turing machines [Tur36]. That is to say, there is *no* program that can take a program as input, and decide (in finite time) whether or not that program terminates on any input, or indeed on a given input.

While undecidability represents an *a priori* restriction on what we may achieve, it does not imply that weaker forms of termination analysis are impossible. In particular, we are forced to restrict our requirements, and to allow a termination analyser, on a given program P, to return a "don't know" answer: for some programs, the analyser produces no information (other perhaps than possible counterexamples which can be manually checked). In this thesis, we shall consider analysers which *either* return a positive termination result *or* fail to produce a definite result[1]. The need to allow the analyser to fail on some programs is not unique to termination analysis — most problems in verification (and indeed more generally program analysis) are undecidable.

Termination Arguments

How may termination of a program be proved? As a prelude to describing automatic analysis, let us a give a couple of examples of manual termination arguments. A simple case is provided by the *factorial* function:

```
fac n = if n = 0 then 1 else n × fac (n−1)
```

It is straightforward to argue that *fac* n terminates for any $n \geq 0$: in any recursive call *fac* $n \rightarrow$ *fac* n', where $n \geq 0$, we necessarily have: $n' < n$ (in fact, $n' = n - 1$), and $n' \geq 0$. Thus the number of possible recursive calls to *fac* is bounded by n, and in particular is finite. Note however that *fac* fails to terminate for inputs $n < 0$, and so the observation that $n' \geq 0$ is crucial.

Similar reasoning extends to imperative programs. For instance, consider the following simple binary search function (searching for a value in an ordered array $a[0..N)$):

[1] It is further possible to consider analysers that may return a definite nontermination result, but this seems less useful.

```
function search(value : α)
var left, right, mid : int;
begin
  left, right ← 0, N;
  while left ≠ right do
  begin
    mid ← (left + right) div 2;
    if a[mid] < value then
      left ← mid + 1
    else
      right ← mid
  end;
  return left
end
```

This function terminates, and if the array is sorted and contains *value* returns an index i such that $a[i] = value$. To prove termination we argue as follows: it is an invariant of the **while** loop that $left \leq right$. Hence (using the guard), in any execution of the loop body, $left < right$. We shall argue that at each iteration of the loop, the value $right - left$ decreases. It suffices to note that the value of *mid* computed in the body satisfies $left \leq mid < right$. Then if the **then** branch is taken, we have: $right' - left' = right - mid - 1 \leq right - left - 1$ (using the convention that primed variables denote the value of a variable at the end of the loop body). Likewise, if the **else** branch is taken, $right' - left' = mid - left < right - left$.

In both cases, the value $right - left$ strictly decreases at each execution of the loop body. However, this value is an integer bounded below by zero (by the invariant of the loop), and so only finitely many decreases are possible — thus the loop must terminate.

These examples reflect the nature of essentially all termination arguments: we attempt to find a *measure* (sometimes known as a variant or bound function) such that:

1. Any infinite execution of the program causes the measure to decrease infinitely, and
2. Infinite decrease in the measure is impossible. That is, the order on the measure is well-founded.

Together, these conditions show that infinite executions are impossible. Note that in some cases, the measure may be apparent in the program (the parameter n in the factorial example), while it may need to be constructed in other cases (such as binary search). Also, as in the case of binary search, termination proofs may rely on knowledge about the program's behaviour (in this case, the loop invariant).

Applications

Applications of termination analysis are of course numerous. First, we can note that termination checking is an integral part of any program verification effort. For, proving that a program or program fragment (for instance, an algorithm to sort a list) is correct naturally splits into two tasks:

1. Proving *partial correctness*: a relationship between the input and output of the program, *if* the program terminates on this input. As an example, in a sorting algorithm the partial correctness requirement is that any output produced should be a sorted permutation of the input

2. Proving termination on any (appropriate) input.

These two tasks are not necessarily independent (in the case of binary search, we use the invariant in the termination argument as well as in the partial correctness proof), but it is usually the case that partial correctness does not rely on termination arguments.

It is important to note at this stage that we may reasonably take the view that *all* programs should terminate, and thus that nontermination is always the result of a programming error. It seems at first sight that such an assertion is invalidated by most real programs — an interactive program is expected not to exit, but rather to wait for user input and respond by performing an action, and thus such programs never terminate. However, such a (reactive) program can invariably be split up into two parts: the infinite "event loop", which waits for user input and calls appropriate event handlers, and the rest of the program encoding its functionality. Then the event loop should be the only nonterminating part of the program: whenever the user asks the program to perform an action, this action should be undertaken in finite time, and control returned to the user.

Beyond Program Verification While program verification is the most direct application of termination analysis, there are many other applications of this technique. An example of such uses of termination analysis is provided by *automated theorem proving*. In any nontrivial theorem proving effort it is necessary to introduce functions (or rather function symbols) into the logic, either because such functions are used in the statement of a theorem, or because they are used at some stage in the proof.

The issues related to function definitions are similar in all theorem provers, but are particularly striking in higher-order logics (for instance, the HOL system [GM93]). In this formalism, based on Church's theory of simple types [Chu40], functions and formulas are uniformly represented as λ-terms, and a logical formula is merely a function returning a boolean value. We may wish to make the following definition:

$$FAC = \lambda n.\mathbf{if}\ n = 0\ \mathbf{then}\ 1\ \mathbf{else}\ n \times FAC\,(n-1)$$

In any reasonable system is should be possible to derive the following theorem:

$$\vdash \forall n.\ FAC\, n = \textbf{if } n = 0 \textbf{ then } 1 \textbf{ else } n \times FAC\,(n-1) \tag{1.1}$$

Consider, however, the "definition"[2] $f = \lambda x.f\ x + 1$. Then we may derive the formula:

$$\vdash \forall x.\ f\ x = f\ x + 1 \tag{1.2}$$

This is unfortunate: we may deduce $f\ 0 = f\ 0 + 1$, whence by left-cancellation, $0 = 1$. Any statement can then be entailed[3].

What has gone wrong? In fact, it is only valid to derive equations (1.1) and (1.2) from the definitions of FAC and f *if* there exist total functions satisfying these definitions. While this is the case of FAC (at least if n is restricted to natural numbers), it is certainly not true of f.

We can therefore see that the definition of functions in theorem provers requires existence proofs for total functions with given properties. As these are expressed in what amounts to a simple functional programming language, this requires termination proofs for functional programs[4].

A related application of termination analysis is found in the program-as-proofs interpretation of type systems, used in languages such as Omega [She05] to derive verifiable assertions about programs. By the Curry-Howard isomorphism we may view types as logical formulae, and programs then represent proofs. That is, whenever a term t has type T, a proof of the formula represented by T can be constructed from t. This has been used in functional languages such as Omega to express properties of programs which may be checked mechanically (in fact, by the typechecker).

The same problem arises, however, as in theorem proving. For instance, consider the definition $f\ x = f\ x$. Under any natural (polymorphic) type system, the function f will be given type $\forall \alpha, \beta.\alpha \to \beta$. This leads *via* the Curry-Howard isomorphism to the worrying assertion that $\forall \varphi, \psi.\varphi \to \psi$. Again, soundness of the desired logic is only obtained *provided* all functions terminate.

Finally, a very different application can be found in *program specialisation*. In program specialisation, a given (general) program is provided, together with annotations marking certain inputs as *static* (available at compile-time). The goal is to obtain a specialised program which computes the same value as the original program, for the given values of static inputs.

In general it is difficult to predict which program variables assume boundedly many values in an execution of the program, and which variables may take arbitrarily many values

[2] This example is taken from the introduction of Konrad Slind's PhD thesis [Sli99].

[3] *"Assume that* $0 = 1$. *Adding one on both sides,* $1 = 2$. *The pope and I are clearly two, so the pope and I are one. Therefore I am the pope."* Alternatively credited to G.H. Hardy or Bertrand Russell.

[4] Termination may not be sufficient in languages with partial operations such as partial pattern matching, but it is certainly necessary.

(depending on dynamic parameters). However, this information is crucial — it is only safe to specialise on bounded parameters. In practice this results in partial evaluators which *either* are needlessly conservative in determining which parameters can safely be used in specialisation, *or* are sometimes nonterminating. Work on termination analysis has thus also been driven by its use in partial evaluation [JG02, Lee01, GJ05], together with the closely related problem of *boundedness analysis*.

1.2 Size-Change Termination

1.2.1 Size-Change Termination for Functional Programs

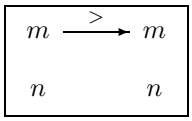
In this thesis we shall be focusing on termination analysis within the framework of *size-change termination*, introduced by Lee, Jones and Ben-Amram [LJBA01] for first-order purely functional programs. The size-change termination principle relies on a simplifying assumption: we assume that all datatypes in the language are ordered by a well-founded partial order. In practice, for functional languages this includes all available datatypes (natural numbers, booleans, user-defined datatypes) *except* integers[5].

Size-change termination (SCT) can then be stated as follows: *if any infinite sequence of calls following the control flow of the program would result in infinite descent in some value, then the program terminates.* This may be seen as a special case of our generic termination argument, where the measure used for termination is restricted to a specific class of functions.

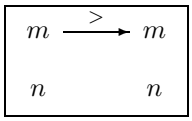
An algorithm is given for SCT analysis of first-order programs. As a somewhat archetypal example, consider the Ackermann function:

```
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))
```

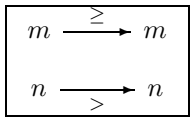
The *ack* function has three recursive calls, labelled in the program text. For each of these calls, a simple analysis can produce a *size-change graph*, shown below:



$1 : ack \to ack$



$2 : ack \to ack$



$3 : ack \to ack$

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.

Given this information, we may argue that the function terminates. Suppose that *ack* were nonterminating on some input. Then for this input there is an infinite sequence of calls

[5] Of course, *any* datatype can be given a well-founded order, but this may not be the natural order. This can fruitfully be used for datatypes that are we do not wish to include in termination proofs — for instance, floating-point numbers might be given a flat (trivial) well-founded order.

in the execution of *ack*, which we may represent as an (infinite) word w in the language $(1 + 2 + 3)^\omega$ of sequences of call labels. Suppose first that w only contains finitely many occurrences of 1 and 2, and so is of the form $w = u3^w$ for some u. Then there is an infinite sequence of calls labelled 3 — but as n decreases in place at each of these calls, n decreases infinitely, which is impossible since n is a natural number. Hence 1 or 2 appears infinitely often in w. Thus $w \in ((1 + 2 + 3)^*(1 + 2))^\omega$. But in any sequence of calls of the form $(1 + 2 + 3)^*(1 + 2)$, the value of m strictly decreases — again a contradiction.

In fact, given any control flow graph annotated with size-change graphs, an algorithm [LJBA01] exists that determines whether any infinite path leads to an infinitely decreasing dataflow path (an infinite sequence of arrows in size-change graphs, infinitely many of which are labelled $>$). We describe this algorithm in Chapter 3.

Note that the argument relies crucially on the fact that both m and n are *natural numbers*, rather than integers. In fact, *ack* is nonterminating for negative inputs.

1.2.2 Discussion

Size-change termination essentially represents a simplifying assumption for termination analysis — namely, that it suffices to consider decreases in the values of program variables to prove termination. This restricts the class of programs for which termination may be inferred, but it appears that the SCT criterion is nonetheless highly effective [MV06]. The simplicity of SCT has a number of useful consequences:

Fully Automatic The SCT analysis is a useful basis for an *purely automatic* termination analysis, that does not rely on user annotations or user-supplied measure functions. The program analysis used need only produce a control flow graph of the program, annotated with size-change graphs as above.

Test-Insensitive Termination in the SCT framework does not rely on the nature of guards appearing in **if** expressions in the program text (beyond what may be used to construct a precise call graph). For, termination is shown by proving that infinite sequences of calls are impossible, *without* reference to guards — in effect, any **if** expression is treated as a nondeterministic choice. This makes the analysis substantially simpler, and eliminates any need for theorem proving, at some cost in precision [MV06].

Completeness The termination problem is naturally undecidable. However, the SCT analysis can be split up into two phases: first, extract the call graph of the program annotated with size-change graphs; then decide whether any infinite path leads to infinite descent. While the first phase is necessarily imprecise, the second phase admits a sound and complete decision procedure. Thus it is guaranteed that the analysis is unaffected by use of:

1. Mutually recursive functions,

2. Permuted function parameters, and

3. Duplicated function parameters.

This improves the robustness of the analysis — we refer the reader to accounts of first-order size-change termination [LJBA01, Lee01] for further examples.

However, the SCT framework implies certain limitations, which we shall consider to be beyond the scope of this text. We outline some of these here, together with some work addressing these issues.

Integers and Increasing Values The most problematic feature of the SCT framework is its reliance on well-founded datatypes. This precludes the use of integers — it is assumed that natural numbers are used instead, so that evaluation of $0 - 1$ fails (and aborts) immediately. This is unrealistic, and dangerously unsound (certainly no implementation of a functional language behaves in this way).

However, a solution to this problem is clear: an integer variable whose value is guaranteed to be *bounded below* is not problematic — the standard order on integers is still well-founded for sets of values that are bounded below. Dually, we may deal with integer values bounded *above*, as in the following program:

```
fromTo a b = if a > b then [] else a : fromTo (a+1) b
```

by recording *increases* in size-change graphs (this is a straightforward extension).

In fact, this leads to a more general solution [Ave06]: the analysis of Cousot and Halbwachs [CH78] may be used to approximate the set of reachable program states (tuples of integer variables) as convex polyhedra. A convex polyhedron is represented as a set of linear inequalities: $0 \leq c + \sum_i a_i x_i$. The positive integral linear expression in each of these can be used as a measure function, and the termination criterion can be restated as: the program terminates provided any infinite path would cause infinite decrease in a linear expression representing the polyhedron.

This extension is independent of the program analysis used to obtain size-change graphs, but is crucial to extend the applicability of size-change termination. Of course, no sound order can be given for values that are not found to be bounded.

Non-monotonic Decrease In the SCT analysis, size-change graphs encode information about dataflow and size changes. Size-change graphs encode information such as $x \xrightarrow{>} y$ (a decrease in value) and $x \xrightarrow{\geq} y$ (non-increasing value). More general labels encoding equality or increases can be used [Fre01], but in our setting, an increase in a parameter value is not directly represented in size-change graphs. Consider however the following (artificial) program:

$$\begin{aligned} f\ x &= \textbf{if } x = 0 \textbf{ then } 1 \textbf{ else } g\ (x+1) \\ g\ x &= f\ (x-2) \end{aligned}$$

Then any recursive call from f to itself (or indeed from g to itself) causes the value of its parameter to decrease — however this is only obtained through an increase followed by a larger decrease.

This type of non-monotonic variation is not naturally handled by the SCT framework, even in extended size-change graphs representing parameter increases, as sound reasoning in the above relies on observing that the decrease in x is larger than the preceding increase. This may be addressed for bounded increase and decrease [Fre01], but the general case is beyond the scope of this framework (and indeed requires arithmetic inequalities in size-change graphs, removing decidability properties).

Abnormal Termination As we have stated previously, the SCT analysis is not sensitive to the nature of **if** guards in the program. As an immediate consequence it is impossible to guarantee that a program terminates *successfully* with the SCT method. Indeed we shall consider that undefined operations (such as $0 - 1$ on natural numbers, division by zero, incomplete partial matching such as *hd* [] or even ill-typed operations) result in *abnormal*, or exceptional, termination: the execution of the program is halted, and thus the program is terminating. We shall consider the problem of showing that a program may only terminate successfully as a separate issue (in fact, this is essentially a safety property).

Size Analysis An aspect of SCT analyses that will not concern us in this text is the use of *global size analysis* to improve results. For instance, consider the following simple QuickSort program:

```
let partition a xs = (List.filter (<= a) xs, List.filter (> a) xs)

let rec quicksort =
  function
    [] → []
  | x :: xs →
    let (ys, zs) = partition x xs in
      quicksort ys @ [x] @ quicksort zs
```

This program *is* size-change terminating, as the argument to *quicksort* in each recursive call is strictly smaller than the list $x :: xs$. However, this is not *locally* observable — it is necessary to show that the lists returned by *partition* are no larger than its input, *i.e.* that *filter* returns a sublist of its input. This can be achieved by size analysis [Fre02], and is essentially independent of the SCT framework.

1.2.3 Towards Termination Analysis of Higher-Order Programs

The SCT analysis was originally formulated for first-order purely functional programs. This has since been extended to other frameworks, such as term rewriting systems [TG03] and

Martin-Löf type theory [Wah04]. However, this stopped short of providing a termination tool for real functional languages. For, such languages typically allow the use of *higher-order* functions. Extending the SCT analysis to higher-order functions requires many conceptual changes, the most salient of which are:

1. Constructing an appropriate call graph for higher-order programs, and
2. Extending the notion of size of a value to functional (higher-order) values.

An important step towards the resolution of these issues was taken by Jones and Bohr [JB04], who presented an application of size-change termination to the pure, call-by-value (untyped) λ-calculus. While the language considered is tiny, and indeed does not contain constants, the novel termination criterion used allows termination of many λ-expressions to be established. In particular, termination of λ-expressions involving Church numerals, or even the Y fixpoint combinator, can be established.

We shall not describe the analysis of Jones and Bohr in more detail in this introduction, as further details can be found in Chapter 3.

1.3 Aims and Contributions

In this text, we shall be concerned with termination analysis of higher-order purely functional languages, based on the size-change termination framework. This is applicable to both strict languages such as the purely functional core of ML [MTHM97] and lazy languages such as Haskell [PJ03].

We present a list of the contributions of this thesis below.

- Work on the SCT analysis [LJBA01] has largely been focused on first-order languages, though small extensions to higher-order functions have been considered [Lee01]. Dually, the termination analysis for the pure λ-calculus of Jones and Bohr [JB04] deliberately ignores many features found in functional programming languages, such as constants, user-defined datatypes and recursion.

  We present the first SCT analysis of ML-like languages. This unifies the first-order SCT analysis and the termination analysis for the pure λ-calculus. A termination analysis handling higher-order functions accurately is essential for successful termination proofs for most real functional programs.

- Any termination checker based on static analysis relies on the specific analysis chosen to construct the call graph. This is particularly evident in the higher-order case — however the analysis of Jones and Bohr [JB04] uses a specific abstraction, making it *a priori* impossible to evaluate the importance of the abstraction.

  We generalise the size-change termination criterion to address these issues. This requires more general notions of size-change graphs (beyond immmediate function parameters), as well as stating the termination criterion as depending on an arbitrary

safe control- and data-flow graph. Furthermore, we will show that the construction of the analysis may be parametrised on the semantics of the language being analysed, broadening its applicability.

- The framework defined for size-change termination allows several methods for constructing the call graph of the program prior to proving termination. This may be used to vary the tradeoff between precision of the call graph construction and resulting termination analysis and the time and space complexity of the procedure.

  We present several instantiations of this framework, using different static analyses. We first show how the analysis of Jones and Bohr [JB04] can be expressed in our framework (this is equivalent to 0CFA [Shi91]). We then define the k-bounded CFA family of analyses, and show that these may be used to improve the precision of the analysis by disambiguating uses of a function in different contexts.

  Finally, we introduce a new static call graph construction for higher-order languages, based on *tree automata* to represent environments. This can increase the precision of the analysis of some programs, in particular in the analysis of programs whose execution creates environments of unbounded depth.

- We present the first SCT analysis of *lazy* functional programs. Lazy programs may of course be analysed simply as strict programs, but this is needlessly coarse. In fact, termination analysis of lazy programs presents several challenges — such programs may manipulate infinite data structures in intermediate steps of the computation, but return a result in finite time.

  Our SCT framework may be applied to lazy languages as well as strict ones. However, we show that this is not in itself sufficient to obtain good results, and give a refinement of the size order on values used for strict languages to address this. Nontrivial lazy programs, even if nonterminating under call-by-value, can then be proved terminating using the size-change termination method.

- Finally, we evaluate the different analysis approaches presented in the text, and account for the relationships between classes of terminating programs accepted by the different instances of our termination test: the k-bounded CFA analyses, as well as our tree automata call graph construction. This allows the precision gains offered by various static analyses to be evaluated, and is complementary to experimental evaluation of the effectiveness of the analyses on real programs. While we will present the analysis of example programs throughout, we regard further experimental assessment of the analysis as beyond the scope of this text.

1.4 Notes to the Reader

This thesis is structured linearly. We present the semantics of the language used throughout the text in Chapter 2, while Appendix A describes a transformation on the semantics used to represent nontermination explicitly, used in Chapters 2 and 5. We then introduce our general size-change termination framework for higher-order programs in Chapter 3. Chapter 4 then presents the different static analyses used for call graph construction (0CFA, k-bounded CFA and our novel tree automata-based analysis). The application of higher-order size-change termination to lazy languages is presented in Chapter 5. Finally, we account for the expressiveness of these analyses in Chapter 6, relating classes of programs accepted by each analysis.

Related work is discussed together with its relationship to our own contributions at the end of each chapter for ease of reference — the reader is encouraged to turn to the end of the chapter for such reviews. A brief overview of previous work in termination analysis is presented in Chapter 7, as other approaches to termination analysis are not immediately connected to any of the results in technical chapters.

Part of this work has been published in a joint paper with Neil Jones [SJ05]. This includes earlier versions of material from Chapters 2 (semantics of the language), 3 (size-change termination framework) and 4 (static analysis). In this paper the k-bounded CFA size-change termination analyses are presented, but not our tree automata call graph construction.

Chapter 2

Semantics and Termination

2.1 The Language

In this chapter, we define the syntax and semantics of the language that we shall consider for termination analysis throughout the text. This is a simple, minimal purely functional language, but is sufficient to serve as a convenient intermediate representation for input languages such as ML. The semantics of the language is defined in big-step operational style. This is subsequently refined to obtain a more explicit representation of nonterminating evaluation.

2.1.1 The Core Language

We aim to analyse programs written in any purely functional, call-by-value language, such as the functional core of ML. For convenience, we will use OCaml [LDG+04] syntax for examples throughout. However, it is desirable to abstract away from the specificities of the ML language, both for presentation purposes and to make the analysis as general as possible. Note in particular that we shall not use the type system of ML in our analysis, and in fact untypable programs can be handled[1].

We shall therefore restrict ourselves to analysing programs written in a restricted *core* language, serving as an intermediate representation. Programs input in (say) OCaml will be translated into the core language prior to analysis, and thus it is desirable this translation be natural and straightforward. However, to simplify the description of the analysis, the core language should be kept minimal.

In light of these requirements, we shall define the core language as a language of *recursive equations*. A program in the core language consists of a series of (mutually recursive) top-

[1]The use of type information in program analysis has the potential to increase precision, and is therefore desirable for the analysis of typed languages. We shall however only concern ourselves with the untyped case in this text. Our analyses are thus applicable to any appropriate language, but could be improved for typed 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 2.1: The Core Language: Example Program

level curried function definitions, together with an expression to be evaluated in the context of these definitions. An example of such a program is shown in Figure 2.1. The main features of OCaml expressions that we exclude are λ-abstractions, as well as local definitions (**let** and **let rec**). In addition, we exclude various forms of syntactic sugar provided by OCaml, as well as orthogonal aspects of the language such as the module language and (in keeping with our aim to analyse purely functional programs) imperative features.

Constants

We first define the constants that may appear in core programs, and the operations on these. It is useful to distinguish between two types of constants: *primitive* constant types, such as natural numbers and characters, and *algebraic* types, including all user-defined ML datatypes.

Primitive Constants The core language includes a set of constant types, such as natural numbers, together with operations on values of these types. The precise set of primitive constants is not relevant to the analysis, and thus we shall leave it unspecified here. We shall merely state the properties that will be required in the sequel.

We assume defined a set *PrimConst* of primitive constants. This must include the type of booleans:

Property 2.1. $\{\textbf{\textit{true}}, \textbf{\textit{false}}\} = \mathbb{B} \subseteq \mathit{PrimConst}$

The crucial property that we shall require in order to apply the size-change termination method is that this set should carry a well-founded order:

Property 2.2. $\prec \subseteq \mathit{PrimConst} \times \mathit{PrimConst}$ *is a well-founded order.*

An example of a suitable set of primitive constants is the (disjoint) union of $\mathbb{B}$, $\mathbb{N}$, the set of characters and the set of strings. An appropriate choice of order $\prec$ might be: for $n, m \in \mathbb{N}$, $n \prec m$ iff $n < m$, with no other constants comparable (more refined orders are of course possible).

Furthermore, we assume defined a set *PrimOp* of primitive operators. This is intended to include operations on primitive constants such as $+$, $-$ and $\wedge$. Each operator $\oplus \in PrimOp$ has an *arity* $\sharp\oplus \in \mathbb{N}_{>0}$. The semantics of primitive operators is defined by the function *Apply*:

Property 2.3. *For each* $\oplus \in PrimOp$, $Apply \oplus : PrimConst^{\sharp\oplus} \to PrimConst \cup \{PrimopErr\}$, *where PrimopErr is a special error value assumed disjoint from all constants.*

The intention is that a primitive operator can only be applied to primitive constants, and only returns primitive constants. However these operators may be partial, which is modelled by the special *PrimopErr* value. For example, we expect that $Apply + (1, 2) = 3$ but $Apply - (0, 1) = PrimopErr$. Likewise, $Apply + (\mathbf{true}, \mathbf{false}) = PrimopErr$.

Algebraic Datatypes ML-like languages allow user-defined datatypes through *algebraic datatype declarations* such as the type of binary trees:

type α btree = Null | Node **of** α btree $\times$ α $\times$ α btree

We shall model these user-defined types in our core language. As we do not require input programs to be typable we regard such definitions as introducing constructors (such as Null or Node) only. It therefore suffices to assume a (finite) set *Constr* of constructors. We extend the map $\sharp$ to constructors, the arity of a constructor being defined as the number of type parameters in its signature. For instance, $\sharp\mathsf{Null} = 0$, while $\sharp\mathsf{Node} = 3$. Note that unlike primitive operators, constructors may have zero arity.

We will represent n-tuples as algebraic datatypes — this is straightforward, defining a constructor tuple_i of arity i for each $i > 1$. Note that though there are infinitely many possible tuple constructors, the arity of a tuple application is known statically, and thus there are finitely many possible arities appearing in the program. There is therefore no contradiction with our requirement that the set of constructors be finite.

Syntax

We shall now define the syntax of core language programs, in BNF notation. A program is a list of top-level function definitions, together with a single expression to be evaluated in the context of these definitions:

$$Program ::= FunctionDef^* \; Expr$$

A function definition defines a function of $n > 0$ arguments:

$$FunctionDef ::= FunctionName \; VarName^+ = Expr$$

We shall assume that each function name is defined only once, and that function names and variable (parameter) names are distinct. Furthermore, we shall assume that all parameters

of all functions have distinct names. This clearly entails no loss of generality.

Furthermore, for each function definition $f\ x_1\ \cdots\ x_n = e$ we define the arity of f to be $\sharp f := n > 0$, and the *body* of f to be $Body(f) := e$.

Finally, expressions include constants and operators, application, conditionals and (simple) pattern matching:

$e \in Expr$	::=	*VarName*	$\ni x$
	|	*FunctionName*	$\ni f$
	|	*PrimConst*	$\ni c$
	|	*PrimOp* $(Expr^+)$	(primitive operator application)
	|	*Constr* $(Expr^*)$	(constructor application)
	|	*Expr Expr*	(function application)
	|	**if** *Expr* **then** *Expr* **else** *Expr*	(conditionals)
	|	**match** *Expr* **with** $Case^+$	(pattern matching)
$Case$	::=	*Constr* $(VarName^*) \rightarrow Expr$	

Program Points We shall need to identify separate occurrences of the same expression in a program for the sequel. We shall therefore assume that each subexpression of the program is uniquely labelled by a *program point*, and write *PP* for the set of program points. Each program point identifies a unique expression, but an expression e may occur multiple times in the program with distinct program points. When we need to specify the program point l at which an expression e occurs, we will write e^l in place of e, but this notation will mostly be elided for clarity.

Any variable x together with its program point l identifies a unique binding construct for x, namely the closest textually enclosing binder for x. This may be a top-level function definition, or a pattern-matching expression. We write $\text{Binder}(x^l)$ for this binding construct[2].

Furthermore, any program point has a well-defined set of variables in scope — the set of variables bound by some binding construct enclosing the given program point. We write $\text{Scope}(l)$ (or $\text{Scope}(e^l)$ to make the expression explicit) for this set.

We have defined primitive operators as *uncurried* functions, that is any occurrence of a primitive operator is of the form $\oplus(x_1, \ldots, x_n)$. This excludes *operator sections* such as $(+\,2)$, as found in Haskell for example. However, it is standard to define operator sections as syntactic sugar for defined functions, for instance $(+\,2)$ is syntactic sugar for $\lambda x.(x + 2)$. We write $a \oplus b$ instead of $\oplus(a, b)$ throughout for clarity.

There are a number of (standard) well-formedness criteria for core programs:

- Whenever a variable x appears at program point l, $x \in \text{Scope}(l)$. In particular, the expression to be evaluated at the end of the program contains no free variables.

[2] As we assume that all parameter names are distinct, the program point l is not strictly required here

- Any constructor application $\mathsf{C}(e_1, \ldots, e_n)$ must satisfy $n = \sharp\mathsf{C}$. If $\sharp\mathsf{C} = 0$ we write simply C in lieu of $\mathsf{C}()$.
- Any primitive operator application $\oplus(e_1, \ldots, e_n)$ must likewise satisfy $n = \sharp\oplus$.
- In a pattern-matching expression, we assume that the top-level constructor of each pattern is distinct.

2.1.2 Analysing ML Programs

The translation from ML (or OCaml) programs to the core language is reasonably straightforward, and we shall not include full details here. The correctness criterion for this translation is that it should be nontermination-preserving — if an OCaml program is nonterminating, its core equivalent should also be nonterminating. This then guarantees that if the core program is found to be terminating by the analysis, the OCaml program is necessarily terminating. In practice however it is essential that the correspondence between the OCaml program and the core program be direct, to assist with reporting any potential counterexamples to termination. Furthermore, the transformation should preserve termination exactly (that is, a terminating program should be translated to a terminating program), for precision.

We will not attempt to deal with the module language of OCaml. Note however that it is possible to eliminate all module definitions from an OCaml program, so that no generality is lost here. The main difference between the functional core of OCaml and our core language is the lack of local and anonymous function declarations in the core language. These (**let**, **let rec** and λ-abstractions) can be eliminated by the *λ-lifting* and *block floating* transformations [Joh85, PJ87, Dan02]. This transforms a program with locally scoped definitions to a program in which all function definitions occur at the top level. For example, the program

```
let f x =
  let g y = x + y in
    (fun z → z) (g 2)
```

might be translated to:

```
fun1 z = z
g x y = x + y
f x = fun1 (g x 2)
```

The transformation is semantics-preserving, and in particular nontermination-preserving, as required.

In addition to local definitions, Ocaml allows a much more general form of pattern matching than the core language. Again, the translation of general pattern matching to simple constructor selection statements is well-known [PJ87] and semantics-preserving, so we omit details.

2.2 Semantics

We shall now proceed to define the semantics of our core language, in big-step operational style. Our semantics will be *closure*-based [Lan64], using environments for variable bindings rather than explicit substitution.

2.2.1 States and Values

States and values represent the configurations that may appear in the evaluation of an expression. Values represent final configurations only (the result of the evaluation) and require no further evaluation, while states represent arbitrary configurations.

We use an environment-based semantics, so that states corresponding to expressions consist of an expression together with an environment, that is a mapping from (a superset of) the free variables of the expression to values. However, we will also require states and values for constants (primitive constants and algebraic constants), as well as primitive operators.

We will now proceed to define states, values, environments and closures more precisely.

Constant Values As we have previously remarked, we will distinguish between primitive constants and algebraic constants. The primitive constant values are just all members of *PrimConst*, such as 42 and **false**. In contrast, algebraic constants can contain other values, which can be arbitrary — for instance, a list may contain primitive constants, other algebraic constants, or even functional values. We therefore define:

Definition 2.4. The set of *constant values* is defined by:

$$\mathit{ConstValue} = \mathit{PrimConst} \cup \{\mathsf{C}(v_1, \ldots, v_n) \mid \mathsf{C} \in \mathit{Constr} \wedge n = \sharp\mathsf{C} \wedge (\forall i) v_i \in \mathit{Value}\}$$

The union in the above is assumed disjoint.

For instance, the expression $(1, 2)$ should reduce to $\mathsf{tuple}_2(1, 2)$.

Closures Functional values are represented by closures. In our setting, a functional value necessarily corresponds to the partial application of a top-level function. Thus the expressions *map* and *map f* (where f is the name of some defined function) should reduce to functional values, described by closures.

A closure consists of a function name together with an *environment*. The environment maps function parameters to values, and the domain of the environment is the set of parameters that have already been applied. In particular, the results of evaluating *map* and *map f* are both closures with function name *map*, and can be distinguished by their environments (empty in the case of *map* and consisting of a binding for the first parameter of *map* in *map f*).

We shall write $\mathit{Params}(f)$ for the list (sequence) of parameters of a function f. By extension we shall assume that for a constructor C of arity n, $\mathit{Params}(\mathsf{C}) = \langle x_1, \ldots, x_n \rangle$.

Definition 2.5. An *environment* is a finite partial map from variable names to values. Thus $Env = VarName \pfun^{\text{fin}} Value$.

Definition 2.6. The set $Closure'$ of *proper closures* is defined by:

$$Closure' = \{(\!|f : \rho|\!) \mid f \in FunctionName \wedge \rho \in Env \wedge \operatorname{dom}\rho \subsetneq Params(f)\}$$

For instance, the expression $(map, (1, 2))$ should reduce to $\mathsf{tuple}_2((\!|map : \emptyset|\!), \mathsf{tuple}_2(1, 2))$.

In addition to the *proper closures* defined above, which represent functional values, it will be convenient to consider a more general notion of closures, representing functions that may be partially or fully applied. Note that while a proper closure is a value (a partially applied function requires no further evaluation), a closure representing a fully applied function is not.

Definition 2.7. The set $Closure$ of *general closures* is defined by:

$$Closure = \{(\!|f : \rho|\!) \mid f \in FunctionName \wedge \rho \in Env \wedge \operatorname{dom}\rho \subseteq Params(f)\}$$

The only difference with Definition 2.6 is that the domain of the environment ρ need not be a strict subset of the parameters of f.

The fundamental operation on a proper closure $(\!|f : \rho|\!)$ is binding a further parameter in the environment ρ. To simplify notation in the sequel we define:

Definition 2.8. For $(\!|f : \rho|\!) \in Closure'$ and $v \in Value$,

$$Bind\ (\!|f : \rho|\!)\ v = (\!|f : \rho \oplus \{x \mapsto v\}|\!) \quad \text{where } x = \operatorname{hd}(Params(f) \setminus \operatorname{dom}\rho)$$

i.e. x is the first parameter of f not in $\operatorname{dom}\rho$.

Thus $Bind\ (\!|map : \emptyset|\!)\ (\!|id : \emptyset|\!) = (\!|map : \{f \mapsto (\!|id : \emptyset|\!)\}|\!)$.

Values A value can now be defined as either a constant value or a proper closure.

Definition 2.9. The set $Value$ of values is the (disjoint) union of $ConstValue$ and $Closure'$.

Note that the definitions of values, closures and environments are mutually recursive. The intent is naturally that the defined sets be the least solution of these recursive definitions.

States Unlike values, states represent configurations that may require further evaluation. Every value is a state, but the converse does not hold. We shall define states to consist of all values, together with states of the form $e : \rho$, where e is an expression and ρ an environment. Such states represent intermediate steps in the execution of the program.

Definition 2.10. A *state* is either a value, a general closure or a pair $e : \rho$, where e is an expression and ρ an environment assigning values to the free variables in e:

$$State = Value \cup Closure \cup \{e : \rho \mid e \in Expr \wedge \rho \in Env \wedge \operatorname{dom}\rho \supseteq \operatorname{fv}(e)\}$$

Values, Variables and Constants

$$\frac{}{v \Downarrow v}\ {\scriptstyle v \in Value} \qquad \frac{}{x : \rho \Downarrow \rho(x)} \qquad \frac{}{c : \rho \Downarrow c}$$

Function References and Closures

$$\frac{}{f : \rho \Downarrow (\!|f : \emptyset|\!)} \qquad \frac{Body(f) : \rho \Downarrow v}{(\!|f : \rho|\!) \Downarrow v}\ {\scriptstyle \mathrm{dom}\,\rho = Params(f)}$$

Primitive Operators

$$\frac{(\forall i) e_i : \rho \Downarrow c_i \quad Apply\ op\ \langle c_1, \ldots, c_n \rangle = c}{op(e_1, \ldots, e_n) : \rho \Downarrow c}\ {\scriptstyle (\forall i) c_i \in PrimConst \atop c \notin Error}$$

Constructors and Pattern Matching

$$\frac{(\forall i) e_i : \rho \Downarrow v_i}{\mathsf{C}(e_1, \ldots, e_n) : \rho \Downarrow \mathsf{C}(v_1, \ldots, v_n)} \qquad \frac{e : \rho \Downarrow \mathsf{C}_l(v_1, \ldots, v_{n_l}) \quad e_l : \rho \oplus \{x_j^l \mapsto v_j\}_{j=1}^{n_l} \Downarrow v}{\textbf{match}\ e\ \textbf{with}\ \left\langle \mathsf{C}_i(x_1^i, \ldots, x_{n_i}^i) \to e_i \right\rangle_{i=1}^{k} : \rho \Downarrow v}$$

Conditionals

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

Function Application

$$\frac{e_1 : \rho \Downarrow (\!|f : \mu|\!) \quad e_2 : \rho \Downarrow w \quad Bind\ (\!|f : \mu|\!)\ w \Downarrow v}{e_1 e_2 : \rho \Downarrow v}\ {\scriptstyle (\!|f:\mu|\!) \in Closure' \atop w \in Value}$$

Figure 2.2: Semantics of the Core Language

2.2.2 Semantics of Evaluation

We have defined the values and states that may appear in the execution of a program, and can now detail the semantics of the core language. This is a straightforward call-by-value big-step operational semantics, and may be found in Figure 2.2 as an inductive definition giving the *evaluation relation* $s \Downarrow v$ (s evaluates to v), where s is a state and v a value.

An important property of this semantics is that it is *deterministic*, defined precisely in Appendix A. Informally, determinism is the property that for any state s, there is *at most one* proof tree with conclusion $s \Downarrow v$, where v is a value.

A straightforward corollary is:

Lemma 2.11. *If $s \in State$ and $s \Downarrow v$, $s \Downarrow v'$, then $v = v'$.*

2.2.3 Error Values

The semantics defined in the previous section represents nontermination as the absence of a valid reduction: if s is a state whose evaluation is nonterminating, then $\neg(s \Downarrow v)$ for any value v (which we will henceforth write $s \not\Downarrow$). However, the converse does not hold: for some

states s, $s \not\Downarrow$, but evaluation of s is terminating. This occurs when the evaluation of s is stuck and therefore s reduces to no proper value. Examples of states s whose evaluation gets stuck include:

1. The expression $0 - 1$ (in any environment): the operator $-$ is partial, and $0 - 1$ should evaluate to *PrimopErr*. We have specifically excluded this eventuality in Figure 2.2 as *PrimopErr* is not a value, so $s \Downarrow \mathit{PrimopErr}$ may not arise.

2. The expression $0 + \mathbf{true}$: type error in the application of $+$. Similarly, the expression 0 1 (0 applied to 1) and other ill-typed expressions fail to reduce.

3. The expression $\mathbf{match}\ [1]\ \mathbf{with}\ [] \to e$: the pattern matching is not exhaustive, and [1] is not matched by the only case.

In all the above cases, evaluation is terminating, as the execution of the program aborts when an invalid operation is encountered. We must represent these cases in the semantics, so that we may characterize nontermination of the evaluation of s precisely as $s \not\Downarrow$.

We can achieve this by generalizing the evaluation judgement $s \Downarrow v$. Where we previously specified that v must be a *Value*, we shall now let v range over values and errors:

Definition 2.12. The set *Error* of error values is defined as $\mathit{Error} = \{\mathit{PrimopErr}, \mathit{TypeErr}, \mathit{PatMatchErr}\}$. We let $\mathit{Value}_{\mathit{Err}} = \mathit{Value} \cup \mathit{Error}$, with the union assumed disjoint.

We may now add rules for producing error values in the semantics, letting $\Downarrow^{\mathrm{err}}$ denote the extended evaluation judgement. The additional rules defining $\Downarrow^{\mathrm{err}}$ are shown in Figure 2.3. These are straightforward, and just involve adding the cases not covered in the previous semantics. In addition, we must add *error propagation* rules — if an error occurs in the evaluation of a subexpression, the error propagates to the top level and aborts the execution. These are again straightforward and relegated to Appendix A.

The $\Downarrow^{\mathrm{err}}$ relation is a superset of $\Downarrow$, and is conservative in the following sense:

Lemma 2.13. *If $s \in \mathit{State}$ and $v \in \mathit{Value}$, then $s \Downarrow v$ iff $s \Downarrow^{\mathrm{err}} v$. In addition, if $s \Downarrow^{\mathrm{err}} e$ for some $e \in \mathit{Error}$, then $s \not\Downarrow$. As a corollary, $\Downarrow^{\mathrm{err}}$ is deterministic.*

It is easily checked that the extended semantics preserves the determinism property. In addition, we have gained a property defined in Appendix A, namely *totality*. Informally, totality states that evaluation of a state never gets stuck. It is thus the case that if $s \not\Downarrow$, then this is because the (unique) proof tree for showing $s \Downarrow$ is infinite (that is, the evaluation of s is nonterminating), not because s fails to match any rule.

As the $\Downarrow^{\mathrm{err}}$ relation is equal to $\Downarrow$ on its domain and merely extends the domain, we shall not need the original $\Downarrow$ relation in the sequel and simply write $\Downarrow$ for the extended relation.

Primitive Operators

$$\frac{(\forall j < i) e_j : \rho \Downarrow c_j \in \mathit{PrimConst} \quad e_i : \rho \Downarrow v_i \notin \mathit{PrimConst}}{op(e_1, \ldots, e_n) : \rho \Downarrow^{\mathrm{err}} \mathit{PrimopErr}}\, i \leq n$$

$$\frac{(\forall i) e_i : \rho \Downarrow c_i \in \mathit{PrimConst} \quad \mathit{Apply\ op}\ \langle c_1, \ldots, c_n \rangle = \mathit{PrimopErr}}{op(e_1, \ldots, e_n) : \rho \Downarrow^{\mathrm{err}} \mathit{PrimopErr}}$$

Pattern Matching

$$\frac{e : \rho \Downarrow^{\mathrm{err}} v \in \mathit{Value} \quad (\nexists i, v_j) v = \mathsf{C}_i(v_1, \ldots, v_{n_i})}{\mathbf{match}\ e\ \mathbf{with}\ \left\langle \mathsf{C}_i(x_1^i, \ldots, x_{n_i}^i) \to e_i \right\rangle_{i=1}^{k} : \rho \Downarrow^{\mathrm{err}} \mathit{PatMatchErr}}$$

Conditionals

$$\frac{e_g : \rho \Downarrow^{\mathrm{err}} v \in \mathit{Value} \setminus \{\mathbf{true}, \mathbf{false}\}}{\mathbf{if}\ e_g\ \mathbf{then}\ e_t\ \mathbf{else}\ e_f : \rho \Downarrow^{\mathrm{err}} \mathit{TypeErr}}$$

Function Application

$$\frac{e_1 : \rho \Downarrow^{\mathrm{err}} v \in \mathit{Value} \setminus \mathit{Closure}'}{e_1 e_2 : \rho \Downarrow^{\mathrm{err}} \mathit{TypeErr}}$$

Figure 2.3: Semantics: Error Values

2.3 Sequentialising Nontermination

In Section 2.2, we have defined the evaluation judgement $s \Downarrow v$ for a state s and a value v, and extended this to represent abnormal termination (abortion) explicitly. In this semantics, nontermination is represented precisely as evaluation failure: for any state s, $s \not\Downarrow$ occurs iff the evaluation of s is nonterminating.

However, this is not yet sufficient to apply the size-change termination method. For, we wish to prove that nontermination is impossible by showing that an infinite sequence of calls is impossible, as in the first-order case. In the SCT framework we may prove that an infinite sequence of calls cannot occur by showing that some well-founded value would decrease infinitely. We must therefore obtain a semantics that represents nontermination in this way, and move away from a big-step semantics.

We define a relation $s \to s'$ (s calls s'), where s and s' are states, with the following property:

Property 2.14. *For any state s, $s \not\Downarrow$ iff there exists an infinite sequence of calls $s = s_0 \to s_1 \to s_2 \to \cdots$.*

The intention is that the $\to$ relation should represent a *traversal* of the unique proof tree with conclusion $s \Downarrow v$. If this proof tree is finite (*i.e.* $s \Downarrow v$ and s terminates), then the set of states reachable in this traversal is finite, thus all call sequences starting at s are likewise finite. Conversely, if this proof tree is infinite (so that $s \not\Downarrow$ and s is nonterminating), then there is an infinite path through the proof tree and so an infinite call sequence.

Property 2.14 holds if the $\to$ relation is defined in the following way:

$$s \to s' \text{ iff evaluation of } s \text{ requires the evaluation of } s'.$$

As an example, consider the rule for evaluation function applications, repeated below:

$$\frac{e_1 : \rho \Downarrow (\!|f : \mu|\!) \quad e_2 : \rho \Downarrow w \quad \mathit{Bind}\ (\!|f : \mu|\!)\ w \Downarrow v}{e_1 e_2 : \rho \Downarrow v}\ (\!|f:\mu|\!) \in \mathit{Closure}'$$

It is clear from this that evaluation of $e_1 e_2 : \rho$ requires evaluation of $e_1 : \rho$ and $e_2 : \rho$, so that $e_1 e_2 : \rho \to e_1 : \rho$ and $e_1 e_2 : \rho \to e_2 : \rho$. Furthermore, provided e_1 and e_2 both reduce, and e_1 reduces to a proper closure (as per the conditions of the above rule), $e_1 e_2 : \rho \to \mathit{Bind}\ (\!|f : \mu|\!)\ w$.

The $\to$ relation should not be mistaken for a small-step operational semantics. In particular, if $\rightsquigarrow$ defines a small-step operational semantics, then whenever $s \rightsquigarrow s'$ and $s \Downarrow v$, then $s' \Downarrow v$ also. That is, in a small-step semantics, a step in the evaluation of a state preserves the value of that state. This is clearly *not* the case with the $\to$ relation we have defined: the value of a state $e_1 e_2 : \rho$ is unlikely to be the same as the value of $e_1 : \rho$.

The definition of the $\to$ relation is shown in Figure 2.4. This definition depends on the existing $\Downarrow$ relation, and should be seen as a form of instrumentation of the inference system defining $\Downarrow$.

The $\to$ relation is presented here with an *ad hoc* definition, but in fact this relation may be deduced from the semantics in Figures 2.2 and 2.3. The details are given in Appendix A. It is further shown that for appropriate semantics, Property 2.14 always holds of the call relation.

2.4 The Dynamic Call Graph

The $s \to s'$ relation defines the *dynamic call graph* of a program. The dynamic call graph represents nontermination explicitly, by Property 2.14 — a program is nonterminating iff there is an infinite path in its dynamic call graph.

Definition 2.15. Let P be a program, with main expression e. The set of *reachable states* of P is $S = \{s \mid e : \emptyset \to^* s\}$. The *dynamic call graph* DCG of P is defined to be $\mathrm{DCG} = \to \cap\, S \times S$.

The definition of the dynamic call graph is thus the graph of the $\to$ relation, restricted to the set of reachable states of the program. It is necessary to restrict the domain of $\to$, as nontermination in the evaluation of *unreachable* states does not affect termination of the program.

As stated above, the dynamic call graph characterises nontermination:

Lemma 2.16. *Let P be a program. Then P is nonterminating iff there is an infinite path in the dynamic call graph DCG of P.*

Proof. Let **start** be the start state of P. Suppose first that P is nonterminating. Then **start** $\not\Downarrow$. Hence by Property 2.14 there is an infinite sequence of calls **start** $= s_0 \to s_1 \to$

Primitive Operators

$$\frac{e_1 : \rho \Downarrow c_1 \quad \cdots \quad e_i : \rho \Downarrow c_i}{op(e_1, \ldots, e_n) : \rho \to e_{i+1} : \rho}\ {\scriptstyle i<n \atop (\forall j \le i) c_j \in PrimConst}$$

Closures

$$\frac{}{(\!|f : \rho|\!) \to Body(f) : \rho}\ {\scriptstyle \mathrm{dom}\,\rho = Params(f)}$$

Constructors and Pattern Matching

$$\frac{e_1 : \rho \Downarrow v_1 \quad \cdots \quad e_i : \rho \Downarrow v_i}{\mathsf{C}(e_1, \ldots, e_n) : \rho \to e_{i+1} : \rho}\ {\scriptstyle i<n}$$

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

Conditionals

$$\frac{\dfrac{}{\textbf{if } e_g \textbf{ then } e_t \textbf{ else } e_f : \rho \to e_g : \rho}}{\textbf{if } e_g \textbf{ then } e_t \textbf{ else } e_f : \rho \to e_t : \rho}\ {\scriptstyle e_g : \rho \Downarrow \textbf{true}}$$

$$\frac{\dfrac{}{\textbf{if } e_g \textbf{ then } e_t \textbf{ else } e_f : \rho \to e_g : \rho}}{\textbf{if } e_g \textbf{ then } e_t \textbf{ else } e_f : \rho \to e_f : \rho}\ {\scriptstyle e_g : \rho \Downarrow \textbf{false}}$$

Function Application

$$\frac{\dfrac{}{e_1 e_2 : \rho \to e_1 : \rho}}{e_1 e_2 : \rho \to e_2 : \rho}\ {\scriptstyle e_1 : \rho \Downarrow (\!|f : \mu|\!) \atop (\!|f:\mu|\!) \in Closure'}$$

$$\frac{e_1 : \rho \Downarrow (\!|f : \mu|\!) \quad e_2 : \rho \Downarrow w}{e_1 e_2 : \rho \to Bind\ (\!|f : \mu|\!)\ w}\ {\scriptstyle (\!|f:\mu|\!) \in Closure'}$$

Figure 2.4: Semantics: Sequentialised Form

$s_2 \to \cdots$. But for each i, **start** $\to^* s_i$, so that s_i is reachable. Hence $s_i \to s_{i+1}$ lies in DCG, and we have exhibited an infinite path in DCG.

Conversely, suppose that there exists an infinite path in DCG, say $t_0 \to t_1 \to t_2 \to \cdots$. Then as t_0 is a node in DCG, t_0 is reachable. Thus **start** $\to^* t_0 \to t_1 \to \cdots$, and we have exhibited an infinite call sequence starting at **start**. By Property 2.14, **start** $\not\Downarrow$, and thus P is nonterminating. □

The SCT analysis attempts to prove that a program terminates by showing that infinite paths in the dynamic call graph can never arise. The above result therefore gives two properties of the analysis:

- *Soundness*: If the SCT analysis indicates that a program terminates, then this is indeed the case
- *Precision*: No precision is lost by considering infinite paths in the dynamic call graph.

Of course, undecidability guarantees that precision must be lost at some other stage of the analysis, and indeed the dynamic call graph of a program is not computable. The remainder of the development of the analysis will be focused on approximating the dynamic call graph, giving the *static call graph*. The key property is that the static call graph should be a conservative (over-) approximation to the dynamic call graph, so that soundness is preserved.

Program Templates

As it stands, however, our semantic framework is not sufficient for the range of termination proofs that we wish to achieve. For, we have only thus far considered termination of individual programs, where termination of a program is just termination of the evaluation of its start state **start**.

However, consider the following situation. We define the *reverse* function:

```
rev xs ys =
  match xs with
    [] → ys
  | z::zs → rev zs (z::ys)

reverse xs = rev xs []
```

How may we prove that for *any* list *xs*, *reverse xs* terminates within the size-change termination framework? To define the dynamic call graph of this program we must define the expression to evaluate; however it is clear that no choice of expression will yield the property that the dynamic call graph contains an infinite path iff *reverse xs* is nonterminating for *some* list *xs*.

In particular, it should be observed that it is not enough to prove termination of evaluation of the expression *reverse*. This will terminate (and return a function), whatever the termination properties of the *reverse* function.

To remedy this situation, we introduce *program templates*. This amounts to allowing free variables to appear in programs, so that we may consider several instantiations of free variables. For instance, we may consider every instantiation of a program in which a free variable is given a list value.

Definition 2.17. A *program template* is a program in which free variables may appear. Given a program template P with free variables $X_1, \ldots, X_n$, a *valuation* $\mathfrak{V}$ for P is a map $\mathfrak{V} : \{X_1, \ldots, X_n\} \to \mathit{Value}$ assigning a value to each free variable.

The semantics of a program now depends on the valuation used to assign meaning to its free variables. This requires the following additional case to define the reduction relation $\Downarrow$:

$$\frac{}{X : \rho \Downarrow \mathfrak{V}(X)}\ \text{with valuation } \mathfrak{V}$$

The notions of evaluation, termination and the dynamic call graph are now unchanged, once a valuation is fixed. To conclude our earlier example, we may now use a program template to consider termination of the *reverse* function on arbitrary inputs. The program template is:

```
rev xs ys =
  match xs with
    [] → ys
  | z::zs → rev zs (z::ys)

reverse xs = rev xs []

reverse <X>
```

We shall use the angle-bracket syntax for free variables throughout to avoid confusion with other program variables.

This particular program template terminates for *any* valuation, showing that the *reverse* function terminates on any input (on non-list inputs, this is exceptional, rather than successful, termination). In general we shall be able to analyse program templates for certain sets of valuations — for instance, assigning a variable X to nonzero natural numbers. The representation of such sets of valuations is a matter for static analysis, and we defer it to Chapters 3 and 4.

2.5 References

The semantics that we have presented in this chapter are standard, see for instance the semantics of ML [MTHM97]. The representation of partially applied and higher-order functions as closures is due to Landin [Lan64]. The restricted language that we consider is similar to that used in some implementations of (in particular lazy) functional languages [PJ87], and compilation from more full-featured languages to such core languages by λ-lifting [Joh85] and pattern matching compilation is well-known.

Jones and Bohr [JB04] observe the need for a closure-based semantics to provide a sensible abstract domain, and point out the property that all expressions appearing in reachable states are subexpressions of the program. As Jones and Bohr consider the pure λ-calculus only all operations are total — thus the treatment of partially defined operations, introduction of error values and the totality requirement for semantics are our own.

The sequentialising transformation (described in Appendix A) was used by Jones and Bohr in an *ad-hoc* fashion, devised manually for their semantics for the call-by-value λ-calculus. The idea of defining this transformation for general languages was suggested to this author by Neil Jones (private communication). The author is not aware of an existing similar definition.

The use of a big-step environment semantics is necessary in our approach to obtain the property that expressions appearing in intermediate states are subexpressions of the program, a property that the straightforward small-step substitution semantics does not possess. It is not clear that a natural small-step semantics adequate for our purposes could be defined. Other approaches are possible, for instance using explicit expression labels and making a distinction between expressions in the programs and *intermediate expressions* arising during the computation rather than requiring that all expressions that appear be subexpressions of the programs [NNH99].

Chapter 3

Size-Change Termination

In Chapter 2, we have defined the operational semantics of our core language, and defined the call relation $s \to s'$. This call relation defines the dynamic call graph of the program, and we have shown that nontermination is precisely characterised as the absence of infinite paths in the dynamic call graph.

The aim of this chapter is now to present the size-change termination framework for higher-order functional programs. This is described in several stages. We first define the well-founded order used to compare arbitrary values (both constants and higher-order values). This order, based on the order used by Jones and Bohr [JB04], will be used to prove that infinite call sequences are impossible.

We then show that the semantics can be annotated with dataflow information, forming the basis for the generation of size-change graphs in static analyses. The groundwork for static approximations to the dynamic call graph is laid in this chapter, but we do not fix a static analysis yet, as this can be varied for precision. Finally, we give a brief account of the algorithmic realisation of the size-change termination, due to Lee, Jones and Ben-Amram [LJBA01].

The definitions and results in this chapter are generalisations of existing work on size-change termination, to adapt the SCT framework to our setting and permit extensions that may improve precision. We shall throughout attempt to highlight results that are included for completeness but are not new to this development.

3.1 Comparing Values

3.1.1 Sizes of Higher-Order Values

In this section we shall be concerned with defining a well-founded order on values. This may then be used within the size-change termination framework to prove program termination. It is not in fact necessary for the relation thus defined to be a partial order, and well-foundedness

is the only crucial property. We refer the reader to Appendix B for definitions and results on well-founded relations.

While we wish to compare values, it is convenient to define a slightly more general order on arbitrary states. This is certainly sufficient as every value is a state (though the converse does not hold). We recall that a state can be one of the following:

1. A primitive constant $c \in \mathit{PrimConst}$
2. An algebraic constant $\mathsf{C}(v_1, \ldots, v_n)$ for some values v_i
3. A closure $(\!|f : \rho|\!)$, where f is a function name and ρ is an environment
4. A state $e : \rho$, where e is an expression and ρ an environment

Comparing primitive constant states presents no difficulty — we have assumed in Property 2.2 (p. 14) a well-founded order $\prec$ on the set of primitive constants.

The problem of comparing arbitrary values is of more interest. In particular, how can two closure states representing functional values be compared? A number of well-founded orders on functional values could be defined — for instance, the pointwise order on functions with well-founded codomain is well-founded. However, it is unclear how such an order could help prove termination of functional programs.

To define our order on higher-order values (for this section, this should be taken to include algebraic constants, which may or may not contain functions), we observe that there is a natural tree structure to these values. For instance, consider the 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 to be evaluated:

$$(\!|\mathit{map} : \begin{array}{l}\{f \mapsto (\!|\mathit{compose} : \{f \mapsto (\!|\mathit{add} : \{x \mapsto 1\}|\!), g \mapsto (\!|\mathit{add} : \{x \mapsto 2\}|\!)\}|\!),\\ \mathit{xs} \mapsto (::)(0, (::)(1, []))\}\end{array} |\!) \tag{3.1}$$

The tree structure of this state is made explicit in the graphical representation in Figure 3.1.

The observation that states (and therefore values) have a natural representation as trees justifies the definition of our order: we shall use the *subtree* order to compare higher-order values. We define this below:

Definition 3.1. Define a relation $<$ on the set *State* as follows:

1. $s < e : \rho$ iff $s \leq \rho(x)$ for some x

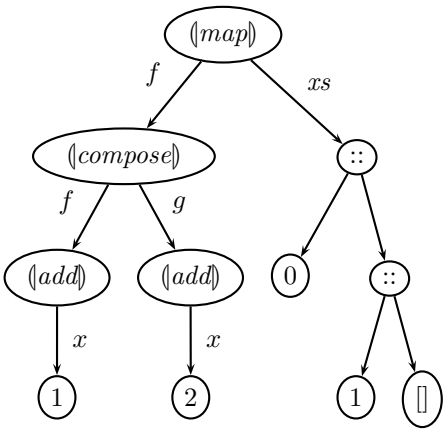


Figure 3.1: The Tree Structure of States

2. $s < (\!|f : \rho|\!)$ iff $s \leq \rho(x)$ for some x

3. $s < \mathsf{C}(v_1, \ldots, v_n)$ iff $s \leq v_i$ for some i

4. $s < c$ iff s is a constant c', and $c' \prec c$

where $s \leq s'$ iff $s < s'$ or $s = s'$.

Let us verify that this does define a well-founded relation on *State*:

Lemma 3.2. *The $<$ relation is a well-founded relation on State.*

Proof. We define a *size function* $|\bullet|$ such that $|s|$ is an ordinal for each state s, as follows:

1. $|e : \rho| = |(\!|f : \rho|\!)| = 1 + \bigcup_{x \in \mathrm{dom}\,\rho} |\rho(x)|$

2. $|\mathsf{C}(v_1, \ldots, v_n)| = 1 + \bigcup_i |v_i|$

3. $|c|$ (for a constant c) can be defined from the well-founded relation $\prec$ by Lemma B.5.

Now $|s|$ is well-defined for all s as states are finite trees. Furthermore it is trivial that $s < s' \Rightarrow |s| < |s'|$. Hence by Lemma B.5, $<$ is a well-founded relation. □

We shall not make the distinction between well-founded orders and more general well-founded relations in the sequel, but neither shall we use any properties of partial orders (in particular, transitivity).

3.1.2 Justification

We have defined the $<$ order on states as the subtree order on higher-order states, together with the $\prec$ order on primitive constants. As remarked previously there are many possible choices of orders on higher-order values, and the question therefore arises of the usefulness

of this particular order. In this section, we shall describe the intuitive appeal of the subtree order on higher-order values. Later chapters will show its use in practice for establishing termination of real programs.

Algebraic Constants It is easiest to see the appeal of this order for algebraic constants. The key observation is that the size-change method relies on observing *local* size changes. It is thus crucial that the natural operations on values create locally observable decreases when appropriate.

In the case of algebraic constants, the operations are: creating a new value by constructor application, and taking a value apart by pattern matching. In the latter case, in a statement

$$\textbf{match } e \textbf{ with } \left\langle \mathsf{C}_i(x_1^i, \ldots, x_{n_i}^i) \to e_i \right\rangle_{i=1}^{k}$$

the values bound to the variables x_j^i of the matching case are all subtrees of the value of e. In particular, all of these values are less than (in the $<$ order) the value of e, which fits with our intuition. For instance, in the case of a list $x :: xs$, the head x and tails xs of the list are deemed to be smaller than it.

Functional Values The case of functional values is of more interest. The only operation on functional values is application — given a function passed as a parameter, it is either ignored or applied. As an example, consider a higher-order function f which applies its argument g:

$$f\ g\ x_1\ \cdots\ x_n = \cdots\ g\ e\ \cdots$$

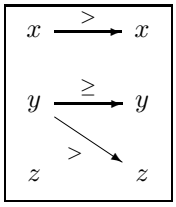
Then in any state s representing an instance of f, in environment ρ, the value of $v = \rho(g)$ of g is a substate of s, so that $v < s$. In the function call $g\ e$, this value v forms the basis of the callee state, as the parameter g is applied. This guarantees that the size of the callee state, excluding the newly bound value of e, is less than that of the caller state s. We therefore expect to be able to prove termination when loops arise through a function calling one of its parameters. This intuition, which shall be made more precise in the sequel, justifies the choice of order on functional values.

3.2 Dataflow and Size Changes

We have seen how an appropriate well-founded order on states and values can be defined in the higher-order case. In addition to this, we will require some machinery for tracking the size of individual values in a computation, in the form of *size-change graphs*.

3.2.1 Graph Bases

In the first-order case, the size-change termination method annotates function calls with size-change graphs that track the dataflow and size-change relationships between the parameters of the calling and callee functions. For instance, a call $f(x, y, z) \to g(x - 1, y, y - 1)$ might be annotated with the following size-change graph:



In the first-order case, environments are flat, so that tracking dataflow at the level of function parameters is clearly appropriate. In the more general higher-order setting, however, environments are trees of arbitrary depth. To reflect this we define the notion of *graph basis* of a state. This is a generalisation of the set of parameters of a function to take into account the parameters of functions themselves bound in the environment.

Definition 3.3. The *graph basis* $\mathrm{gb}(s)$ of a state s is defined by:

1. $\mathrm{gb}(\llparenthesis f : \rho \rrparenthesis) = \mathrm{gb}(e : \rho) = \{\varepsilon\} \cup \{x : p \mid x \in \mathrm{dom}\,\rho \wedge p \in \mathrm{gb}(\rho(x))\}$
2. $\mathrm{gb}(\mathsf{C}(v_1, \ldots, v_n)) = \{\varepsilon\} \cup \{x_i : p \mid 1 \leq i \leq n \wedge p \in \mathrm{gb}(v_i)\}$
3. $\mathrm{gb}(c) = \{\varepsilon\}$ for constants c.

The elements of $\mathrm{gb}(s)$ are the *environment paths* of s. Each environment path p denotes a substate $s\nabla p$ of s, defined by the following:

Definition 3.4. The substate of a state s at environment path p is $s\nabla p$, defined by:

$$\begin{aligned} s\nabla\varepsilon &= s \\ \llparenthesis f : \rho \rrparenthesis \nabla x : p &= \rho(x)\nabla p \\ e : \rho \nabla x : p &= \rho(x)\nabla p \\ \mathsf{C}(v_1, \ldots, v_n)\nabla x_i : p &= v_i \nabla p \end{aligned}$$

Consider first the simple case of a first-order function, with primitive constant arguments. Such a function state (of the form $\llparenthesis f : \rho \rrparenthesis$) has graph basis $\{\varepsilon\} \cup \{\langle x \rangle \mid x \in \mathrm{dom}\,\rho\}$. The set of values reachable by nonempty environment paths are therefore just the values bound to parameters of f, as expected. As an example in the higher-order case, consider the state s shown in Equation (3.1) (p. 29). The graph basis of this state is $\{\varepsilon, \langle f \rangle, \langle f, f \rangle, \langle f, f, x \rangle, \langle f, g \rangle, \langle f, g, x \rangle, \langle xs \rangle, \langle xs, x_1 \rangle, \langle xs, x_2 \rangle, \langle xs, x_2, x_1 \rangle, \langle xs, x_2, x_2 \rangle\}$, corresponding to the set of paths from the root of the tree given in Figure 3.1. Also, $s\nabla\langle f \rangle$ is the substate rooted at *compose*, while $s\nabla\langle xs, x_2, x_1 \rangle = 1$.

A few properties of the environment path lookup operator are given below:

Lemma 3.5. *Let s be a state. Then for any p, $s\nabla p \leq s$, and if $p \neq \varepsilon$, $s\nabla p < s$. Also, for any p, q, $s\nabla(p +\!\!+ q) = (s\nabla p)\nabla q$*

Proof. The proof that $s\nabla p \leq s$ by induction on $|p|$ is straightforward: if $p = \varepsilon$ then $s\nabla p = s$, and for the inductive step $s\nabla x : p = s'\nabla p \leq s' < s$ where s' is a substate of s: $s' = \rho(x)$ if $s = e : \rho$ or $s = (\!|f : \rho|\!)$, and if $s = \mathsf{C}(v_1, \ldots, v_n)$ then $s' = v_i$ for some i. This also gives $s\nabla p < s$ if $p \neq \varepsilon$.

Similarly we may prove $s\nabla(p +\!\!+ q) = (s\nabla p)\nabla q$ by induction on $|q|$; the details are left to the reader. □

3.2.2 Size-Change Graphs

Size-Change Graphs and Arrows

Size-change graphs record information about dataflow and size relationships. A size-change graph (typically written γ) annotates a pair of states (s, s'), where it will usually be the case that $s \to s'$, and describes these relationships between (some of) the substates of s and s'. More formally,

Definition 3.6. A *size-change graph* γ is a triple (A, Γ, B), where A and B are set of environment paths, and $\Gamma \subseteq A \times \{>, \geq\} \times B$. A is the *in-basis* of γ, B is its *out-basis* and Γ its set of *arrows*.

The intention is that a triple $(p, >, q)$ (resp. $(p, \geq, q)$) indicates that the substate rooted at p in the in state is greater (resp. no smaller than) the substate rooted at q in the out state.

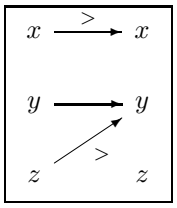
As an alternative formulation, we may regard a size-change graph $\gamma = (A, \Gamma, B)$ as a map from pairs $(p, q) \in A \times B$ to the set of arrow labels $\{\bot, \geq, >\}$, such that

$$\gamma(p,q) = \begin{cases} > & \text{if } (p, >, q) \in \Gamma \\ \geq & \text{if } (p, \geq, q) \in \Gamma \\ \bot & \text{otherwise} \end{cases}$$

The set of labels has a natural order $\bot \prec \geq \prec >$: $\bot$ indicates that no information is known, $\geq$ that the value does not increase, and $>$ that the value definitely decreases. The order on arrows lifts to an order on size-change graphs, which generalizes the subset order on arrow sets:

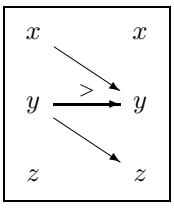
Definition 3.7. Let $\gamma = (A, \Gamma, B)$ and $\delta = (A, \Delta, B)$ be size-change graphs. Then $\gamma \subseteq \delta$ iff for all $(p, q) \in A \times B$, $\gamma(p, q) \preccurlyeq \delta(p, q)$.

Example 3.8. We will often represent size-change graphs graphically, as we have done informally before. For instance, the size-change graphs γ and δ from $A = \{\langle x \rangle, \langle y \rangle, \langle z \rangle\}$ to A are shown below with their arrow sets:



γ

$\{(x, >, x), (y, \geq, y), (z, >, y)\}$



δ

$\{(x, \geq, y), (y, >, y), (y, \geq, z)\}$

As in the above example we shall omit $\geq$ labels from size-change graphs when represented in this way.

Consider two size-change arrows (p, l, q) and (q, l', r). What can be said about the relationship between p and r, given that the information in the two given arrows is correct? If one of the labels is l then no information can be deduced, and otherwise it is readily checked that the maximum of the two labels is a safe description. This is used to define the *composition* of two size-change graphs:

Definition 3.9. Let l and l' be arrow labels. The *composition* of l and l' is:

$$l \cdot l' = \begin{cases} \bot & \text{if } l \wedge l' = \bot \\ l \vee l' & \text{otherwise} \end{cases}$$

Definition 3.10. Let $\gamma = (A, \Gamma, B)$ and $\delta = (B, \Delta, C)$ be size-change graphs. The *composition* of γ and δ is the size-change graph $\gamma; \delta$ from A to C such that

$$(\gamma; \delta)(p, r) = \bigvee_{q \in B} \gamma(p, q) \cdot \delta(q, r)$$

for all $(p, r) \in A \times C$.

The composition operator is just relational composition, where edges in the relation are labelled. In addition, let us define the operation of *union* on size-change graphs:

Definition 3.11. Let $\gamma = (A, \Gamma, B)$ and $\delta = (A, \Delta, B)$ be size-change graphs. The *union* of γ and δ is the size-change graph $\gamma \cup \delta$ from A to B such that

$$(\gamma \cup \delta)(p, q) = \gamma(p, q) \vee \delta(p, q)$$

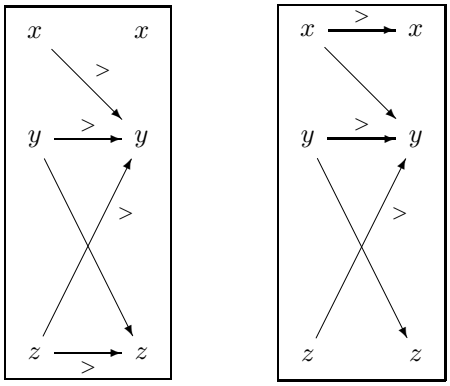
for all $(p, q) \in A \times B$.

Composition and union are monotonic:

Lemma 3.12. *If γ and δ are size-change graphs from A to B and B to C (resp. both from A to B), and $\gamma \subseteq \gamma'$, $\delta \subseteq \delta'$, then $\gamma; \delta \subseteq \gamma'; \delta'$ (resp. $\gamma \cup \delta \subseteq \gamma' \cup \delta'$).*

The proof is immediate from the above definitions.

Example 3.13 (Example 3.8 continued)**.** The composition and union of γ and δ are:



Safety

We have thus far defined size-change graphs without reference to actual states. The connection between size-change graphs and state transitions is given by the notion of *safety* which we shall now define. A size-change graph is safe for a pair of states (s, s') if the information that it encodes about size relationships between substates of s and s' is accurate.

Definition 3.14. A size-change graph $\gamma = (A, \Gamma, B)$ is *safe* for the pair of states (s, s') if:

1. $A \subseteq \mathrm{gb}(s)$ and $B \subseteq \mathrm{gb}(s')$, and
2. For each $(p, q) \in A \times B$, if $v = s\nabla p$ and $w = s'\nabla q$: if $\gamma(p, q) = >$ then $v > w$, and if $\gamma(p, q) = \geq$ then $v \geq w$.

Composition and union naturally preserve safety:

Lemma 3.15. *Let γ be safe for (s, t) and δ be safe for (t, r) (resp. (s, t)). Then $\gamma;\delta$ is safe for (s, r) (resp. $\gamma \cup \delta$ is safe for (s, t)).*

There may be many graphs that are safe for a pair of states. The maximal safe size-change graph is guaranteed to be unique, however.

In the definition of safety of a graph γ for states (s, s'), it is merely required that any information encoded by an edge from x to y in γ be correct with respect to states s and s'. This may hold whether or not there exists a flow of data from x to y in an actual transition from state s to s'. However, in practice the size-change graphs that we shall produce truly reflect dataflow information. In particular, in the next section we shall see how dataflow may be tracked using size-change graphs.

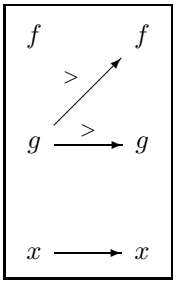
Example 3.16. Consider the state shown below:

$$s = (\!|\mathit{compose} : \begin{array}{l} \{f \mapsto (\!|\mathit{times} : \{x \mapsto 1\}|\!), \\ g \mapsto (\!|\mathit{compose} : \{f \mapsto (\!|\mathit{minus} : \{x \mapsto 2\}|\!), g \mapsto (\!|\mathit{plus} : \{x \mapsto 3\}|\!)\}|\!), \\ x \mapsto 4\} \end{array} |\!)$$

Then this state (indirectly) calls the state

$$s' = (\!|compose : \{f \mapsto (\!|minus : \{x \mapsto 2\}|\!), g \mapsto (\!|plus : \{x \mapsto 3\}|\!), x \mapsto 4\}|\!)$$

The following graph is safe for (s, s'):



This graph is not maximal, though it is the maximal graph with given in- and out- bases. This example illustrates the behaviour of our order on functional values under function application: as the function g is applied in state s, yielding state s', the value of g in s is greater than the values of parameters in s'. This yields the edge $g \xrightarrow{>} g$, which may be useful in proving termination of the program.

3.3 Dataflow in the Dynamic Call Graph

In this section, we shall use the machinery of size-change graphs to define an annotated semantics for our core language that allows *dataflow* to be tracked in the evaluation of a program. To wit, we shall define analogues of the evaluation judgement $s \Downarrow v$ and the call judgement $s \to s'$ that produce size-change graphs to describe the dataflow in a reduction or call. These judgements will be written $s \Downarrow v, \gamma$ and $s \to s', \gamma$; where the intention is not merely that γ is safe for (s, v) (resp. (s, s')) but also that the edges in γ reflect the dataflow in the reduction or call.

3.3.1 Operations on Size-Change Graphs

We have so far introduced the elementary operations of union and composition on size-change graphs. In this section we define additional useful operations on graphs that are required in the definition of the annotated semantics.

Identity and Empty Graphs Let S be a set of environment paths. We may define the *identity graph* on S to be the graph 1_S with in- and out-bases S and arrow set $\{(p, \geq, p) \mid p \in S\}$. By extension given sets S, T of environment paths we define $1_{S,T}$ to be the graph with in-basis S, out-basis T and arrow set that of $1_{S \cap T}$. As a special case, $1_S = 1_{S,S}$. Furthermore, if s is a state, by extension we define $1_s = 1_{\mathrm{gb}(s)}$.

Furthermore, given S, T as before we may define the *empty graph* with in-basis S and out-basis T to be the graph $0_{S,T}$ with empty arrow set.

Lemma 3.17. *Let s, t be states with $gb(s) \supseteq S$ and $gb(t) \supseteq T$. Then $0_{S,T}$ is safe for (s,t). Furthermore, if s and t are equal for environment paths in $S \cap T$ ($s\nabla p = t\nabla p$ for all $p \in S \cap T$) then $1_{S,T}$ is safe for (s,t). In particular, 1_S is safe for (s,s).*

Restriction The next operation that we shall require is *restriction* of a size-change graph. Suppose that γ is a size-change graph with in-basis S and out-basis T, and suppose that $X \subseteq S$, $Y \subseteq T$. The *restriction* of γ to X and Y is defined as expected (the restriction of a relation to a given domain and codomain).

Definition 3.18. Let $\gamma = (S, \Gamma, T)$ be a size-change graph, and let $X \subseteq S$, $Y \subseteq T$. Then the *restriction* of γ to X and Y is $\gamma \restriction_X^Y = (X, \Delta, Y)$, where $\Delta = \{(p, l, q) \in \Gamma \mid p \in X \wedge q \in Y\}$.

Lemma 3.19. *Let s, t be states with $gb(s) \supseteq X$ and $gb(t) \supseteq Y$, and suppose that γ is safe for (s,t). Then whenever s', t' are states such that $s'\nabla p = s\nabla p$ for all $p \in X$ and $t'\nabla q = t\nabla q$ for all $q \in Y$, $\gamma \restriction_X^Y$ is safe for (s',t').*

Corollary 3.20. *With notation as in the above, $\gamma \restriction_X^Y$ is safe for (s,t) for any $X \subseteq gb(s)$ and $Y \subseteq gb(t)$.*

A useful special case of restriction is given below. We note that many state transitions do not affect the environment of a state. For example, in a function application, there is a call $e_1 e_2 : \rho \to e_1 : \rho$ where only the expression part of the state is affected. We introduce some notation for this frequently occurring case.

Lemma 3.21. *Let s and t be states and $\gamma = (S, \Gamma, T)$ a safe size-change graph for (s,t). Then whenever s' is a state with the same environment as s, $\gamma^* := (S, \{(p, l, q) \in \Gamma \mid p \neq \varepsilon\}, T)$ is safe for (s', t).*

Likewise, $\gamma^\dagger := (S, \{(p, l, q) \in \Gamma \mid q \neq \varepsilon\}, T)$ is safe for (s, t') whenever t' has the same environment as t.

Path Substitution The final graph operation that we shall consider is *substitution*. This operation is used to adjust graphs to reflect dataflow. Suppose that t is a state, and t' is obtained from t by mapping a substate of t (say at environment path p) to a substate of t' (at environment path q). Then given a size-change graph γ safe for some state transition (s,t), we may obtain a graph safe for (s,t') by substituting the environment path q for p in the codomain of γ. This is used in state transitions that affect the environment, such as variable lookup and function application, to track dataflow.

Definition 3.22. Let $\gamma = (S, \Gamma, T)$ be a size-change graph. Then we define the *path substitution* operation as follows. Let $p \in T$ and q an arbitrary environment path. Define $T' = \{q +\!\!+ r \mid p +\!\!+ r \in T\}$. Then $\gamma[q/p] = (S, \Gamma', T')$ where $\Gamma' = \{(s, l, q +\!\!+ r) \mid (s, l, p +\!\!+ r) \in \Gamma\}$.

The above definition is asymmetrical: we only consider substitution in the target of a size-change graph, not its source. This is merely due to the fact that we shall not have a use for source substitution in the sequel.

Lemma 3.23. *Let s, t be states and γ be safe for (s,t). If t' is a state such that $t'\nabla q = t\nabla p$, then $\gamma[q/p]$ is safe for (s,t').*

Proof. Suppose that $(u,l,x) \in \gamma[q/p]$. Then $x = q +\!\!+ r$ for some r, where $(u,l,p +\!\!+ r) \in \gamma$. Let $y = p +\!\!+ r$. Therefore $t'\nabla x = t'\nabla(q +\!\!+ r) = t'\nabla q\nabla r = t\nabla p\nabla r = t\nabla(p +\!\!+ r) = t\nabla y$. Hence as γ is safe for (s,t), if $l = >$, $s\nabla u > t\nabla y = t'\nabla x$, as required. Similarly, if $l = \geq$, then $s\nabla u \geq t'\nabla x$. □

3.3.2 Generating Safe Size-Change Graphs

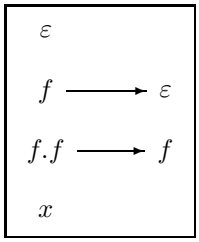
In the previous section we have defined the basic operations on size-change graphs that are required in the definition of the annotated semantics. It is now useful to look at the application of the above constructions in the nontrivial state transitions in the semantics.

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 $\langle x \rangle$) in s is just v. By Lemma 3.23 we therefore have the following:

Lemma 3.24. *The graph $1_s[\varepsilon/x]$ is safe for (s,v).*

For instance, consider the state $s = \mathit{apply} : \{f \mapsto (\!|\mathit{apply} : \{f \mapsto (\!|\mathit{id} : \emptyset|\!)\}|\!), x \mapsto 0\}$ Then the graph $1_s[\varepsilon/f]$ is shown below:



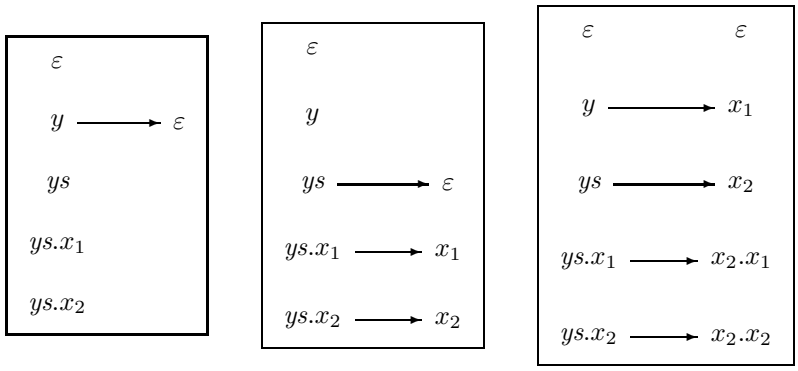
Constructor Application

Consider now a state of the form $s = \mathsf{C}(e_1,\ldots,e_n) : \rho$. Suppose further that for each i, $s_i = e_i : \rho \Downarrow v_i$, and that γ_i is safe for (s_i, v_i). Then $s \Downarrow v := \mathsf{C}(v_1,\ldots,v_n)$. Now each state v_i is bound to x_i in v, whence we can deduce the following:

Lemma 3.25. *The graph $\mathit{constr}(\gamma_1,\ldots,\gamma_n) := {\gamma_1}^*[x_1/\varepsilon] \cup \cdots \cup {\gamma_n}^*[x_n/\varepsilon]$ is safe for (s,v).*

Proof. It suffices to show that ${\gamma_i}^*[x_i/\varepsilon]$ is safe for (s,v) for each i, as union preserves safety. Now γ_i is safe for (s_i,v_i), so by Lemma 3.21, ${\gamma_i}^*$ is safe for (s,v_i). As $v_i = v_i\nabla\varepsilon = v\nabla x_i$, by Lemma 3.23, ${\gamma_i}^*[x_i/\varepsilon]$ is safe for (s,v). □

As an example, consider the state $s = y :: ys : \rho$, where $\rho = \{y \mapsto 1, ys \mapsto [2]\}$. Then $s \Downarrow [1;2]$. The size-change graphs for the reductions $y : \rho \Downarrow 1$, $ys : \rho \Downarrow [2]$ and $s \Downarrow [1;2]$ are shown below:



$y : \rho \Downarrow 1$ $ys : \rho \Downarrow [2]$ $y :: ys : \rho \Downarrow [1;2]$

Function Application

Let us now consider dataflow in function applications. Let s be a state of the form $s = e_1 e_2 : \rho$. Suppose further that $s_1 := e_1 : \rho \Downarrow s_b := (\!|f : \mu|\!)$ with γ safe for (s_1, s_b) and that $s_2 := e_2 : \rho \Downarrow w$ with δ safe for (s_2, w).

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

Lemma 3.26. *The graph* $\mathit{app}_x(\gamma, \delta) := \gamma^{*\dagger} \cup (\delta^*[x/\varepsilon])$ *is safe for* (s, s').

Proof. We first observe that as γ is safe for (s_1, s_b), by Lemma 3.21, γ^* is safe for (s, s_b). Furthermore, for all $q \neq \varepsilon$ in the out-basis of γ, $s'\nabla q = (\!|f : \mu \oplus \{x \mapsto w\}|\!)\nabla q = (\!|f : \mu|\!)\nabla q = s_b\nabla q$. Thus $\gamma^{*\dagger}$ is safe for (s, s').

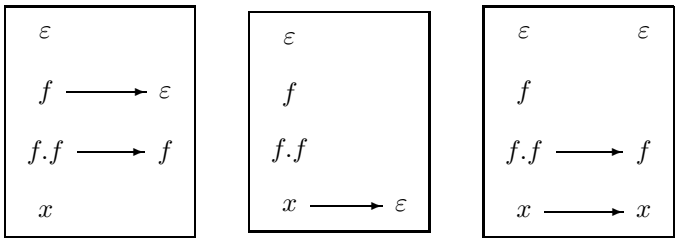
Furthermore, δ is safe for (s_2, w), whence δ^* is safe for (s, w). As $s'\nabla x = w = w\nabla\varepsilon$, by Lemma 3.23, $\delta^*[x/\varepsilon]$ is safe for (s, s'). The result follows as union preserves safety. □

The size-change graph $\mathrm{app}_x(\gamma, \delta)$ consists of two parts: the graph γ (*sans* ε) describes the dataflow between s and the parameters of f already bound in s_b; while the graph δ (with x substituted for ε) describes the dataflow between s and the newly bound value w.

As an examle, let us consider the function

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

Evaluation of the state $f\ x : \rho$, where $\rho = \{f \mapsto (\!|\mathit{apply} : \{f \mapsto (\!|\mathit{id} : \emptyset|\!)\}|\!), x \mapsto 1\}$ proceeds as follows. The state $s_1 = f : \rho$ is evaluated to $s_b = (\!|\mathit{apply} : \{f \mapsto (\!|\mathit{id} : \emptyset|\!)\}|\!)$, and $s_2 = x : \rho$ is evaluated to 1. The next state to be evaluated is then $s' = (\!|\mathit{apply} : \{f \mapsto (\!|\mathit{id} : \emptyset|\!), x \mapsto 1\}|\!)$. The size-change graphs for $s_1 \Downarrow s_b$, $s_2 \Downarrow 1$ and $s \to s'$ are shown below:



Pattern Matching

To conclude the description of dataflow graphs in the evaluation of programs, we shall consider the pattern matching construct. Let $s = \textbf{match}\ e\ \textbf{with}\ \left\langle \mathsf{C}_i(y_1^i, \ldots, y_{n_i}^i) \to e_i \right\rangle_{i=1}^{k} : \rho$. Suppose that $s_m := e : \rho \Downarrow w = \mathsf{C}_i(v_1, \ldots, v_{n_i})$. Then evaluation of s entails evaluation of the state $s' = e_i : \rho \oplus \{y_j^i \mapsto v_j\}_{j=1}^{n_i}$. We assume that graph γ is safe for (s_m, w).

The dataflow in the transition $s \to s'$ is similar to that of function applications. The environment of s' is composed of two parts: the base environment ρ, which is carried over unchanged from s, and the new bindings $x_j^i \mapsto v_j$. The graph representing this dataflow is defined below:

Lemma 3.27. *The graph* $\mathit{patmatch}_s^i(\gamma) := 1_s^* \cup \gamma^*[y_1^i, \ldots, y_{n_i}^i / x_1, \ldots, x_{n_i}]$, *is safe for* (s, s').

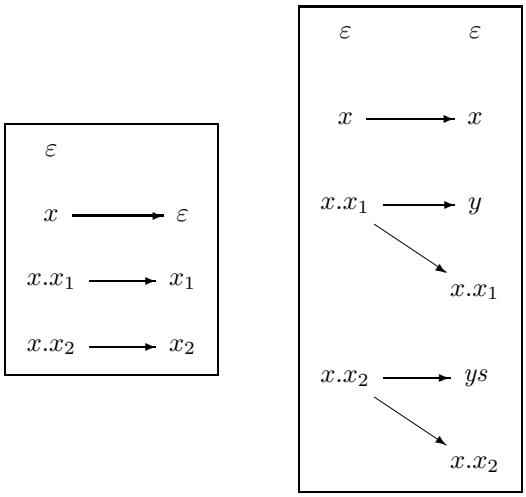
Proof. We first note that for all $p \in \mathrm{gb}(s)$, if $p \neq \varepsilon$, then $s'\nabla p = s\nabla p$, so that 1_s^* is safe for (s, s').

Furthermore, γ is safe for (s_b, w), so γ^* is safe for (s, w). Now for each j, $w\nabla x_j = v_j = s'\nabla y_j^i$. Hence by (the obvious extension of) Lemma 3.23, $\gamma^*[y_1^i, \ldots, y_{n_i}^i / x_1, \ldots, x_{n_i}]$ is safe for (s, s'), as required. □

As an example, consider the program fragment:

```
f x =
  match x with
    [] → ...
  | y::ys → e'
```

for some expression e'. Let e be the body of f, and $s = e : \rho$, where $\rho = \{x \mapsto [1]\}$. Then $x : \rho \Downarrow [1]$, and $s \to s' = e' : \mu$, where $\mu = \rho \oplus \{y \mapsto 1, ys \mapsto []\}$. The size-change graphs are shown below:



3.3.3 The Annotated Semantics

We now have all the required building blocks to define the annotated semantics. This is given in Figures 3.2 (for the reduction relation) and 3.3 (for the call relation).

To simplify notation, we write simply 1^* and 0 in the annotated semantics, eliding the subscripts that indicate the in- and out-bases of this graph. These can be deduced from context without difficulty. Furthermore, whenever this cannot be elided, we use s to denote the left-hand side of a judgement $s \Downarrow v$ or $s \to s'$.

It is worth stressing that the annotated semantics of Figures 3.2 and 3.3 are intended to track *dataflow* for higher-order and algebraic values. In particular, the size-change graphs that are produced in calls and reductions do not contain decreasing edges. This may seem counter-intuitive, but we are not at this stage concerned with using the graphs produced to establish termination. We shall show in Chapter 4 how these graphs can be adapted, in the context of the *static* call graph, to prove termination. Likewise, we do not generate dataflow edges for first-order values (as shown by the empty graph produced for primitive operator evaluation).

The results of the previous section let us deduce the following property:

Lemma 3.28. *Suppose that $s \Downarrow v, \gamma$ (resp. $s \to s', \gamma$). Then γ is safe for (s, v) (resp. (s, s')).*

3.4 The Static Call Graph

We have now defined the dynamic call graph of a program in Chapter 2, defined the order on values that we shall be using and size-change graphs to record order information alongside state transitions, and finally shown how dataflow graphs can be produced alongside the execution of a program. The final component of our analysis is a computable approximation

Values, Variables and Constants

$$\frac{}{v \Downarrow v, 1_v}\ {\scriptstyle v \in Value} \qquad \frac{}{x : \rho \Downarrow \rho(x), 1_s[\varepsilon/x]} \qquad \frac{}{c : \rho \Downarrow c, 0}$$

Function References and Closures

$$\frac{}{f : \rho \Downarrow (\!|f : \emptyset|\!), 0} \qquad \frac{Body(f) : \rho \Downarrow v, \gamma}{(\!|f : \rho|\!) \Downarrow v, \gamma^*}\ {\scriptstyle \mathrm{dom}\,\rho = Params(f)}$$

Primitive Operators

$$\frac{(\forall i) e_i : \rho \Downarrow c_i, \gamma_i \quad Apply\ op\ \langle c_1, \ldots, c_n\rangle = c}{op(e_1, \ldots, e_n) : \rho \Downarrow c, 0}\ {\scriptstyle (\forall i) c_i \in PrimConst,\ c \notin Error}$$

Constructors and Pattern Matching

$$\frac{(\forall i) e_i : \rho \Downarrow v_i, \gamma_i}{\mathsf{C}(e_1, \ldots, e_n) : \rho \Downarrow \mathsf{C}(v_1, \ldots, v_n), \mathrm{constr}(\gamma_1, \ldots, \gamma_n)}$$

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

Conditionals

$$\frac{e_g : \rho \Downarrow \mathbf{true}, \delta \quad e_t : \rho \Downarrow v, \gamma}{\mathbf{if}\ e_g\ \mathbf{then}\ e_t\ \mathbf{else}\ e_f : \rho \Downarrow v, \gamma^*} \qquad \frac{e_g : \rho \Downarrow \mathbf{false}, \delta \quad e_f : \rho \Downarrow v, \gamma}{\mathbf{if}\ e_g\ \mathbf{then}\ e_t\ \mathbf{else}\ e_f : \rho \Downarrow v, \gamma^*}$$

Function Application

$$\frac{e_1 : \rho \Downarrow (\!|f : \mu|\!), \gamma_1 \quad e_2 : \rho \Downarrow w, \gamma_2 \quad Bind\ (\!|f : \mu|\!)\ w \Downarrow v, \delta}{e_1 e_2 : \rho \Downarrow v, \mathrm{app}_x(\gamma_1, \gamma_2); \delta}\ {\scriptstyle (\!|f:\mu|\!) \in Closure' \wedge\ x = \mathrm{hd}(Params(f) \setminus \mathrm{dom}\,\mu)}$$

Program Template Variables

$$\frac{}{X : \rho \Downarrow \mathfrak{V}(X), 0}\ {\scriptstyle \text{with valuation } \mathfrak{V}}$$

Figure 3.2: Dataflow: Evaluation of states

Primitive Operators

$$\frac{e_1 : \rho \Downarrow c_1, \gamma_1 \quad \cdots \quad e_i : \rho \Downarrow c_i, \gamma_i}{op(e_1, \ldots, e_n) : \rho \to e_{i+1} : \rho, 1^*}\ {\scriptstyle i<n \ (\forall j \le i) c_j \in PrimConst}$$

Closures

$$\frac{}{(\!|f : \rho|\!) \to Body(f) : \rho, 1^*}\ {\scriptstyle \mathrm{dom}\,\rho = Params(f)}$$

Constructors and Pattern Matching

$$\frac{e_1 : \rho \Downarrow v_1 \quad \cdots \quad e_i : \rho \Downarrow v_i}{\mathsf{C}(e_1, \ldots, e_n) : \rho \to e_{i+1} : \rho, 1^*}\ {\scriptstyle i<n}$$

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

Conditionals

$$\frac{\begin{array}{c}\\ e_g : \rho \Downarrow \textbf{true}\end{array}}{\textbf{if}\ e_g\ \textbf{then}\ e_t\ \textbf{else}\ e_f : \rho \to e_g : \rho, 1^*} \qquad \frac{\begin{array}{c}\\ e_g : \rho \Downarrow \textbf{false}\end{array}}{\textbf{if}\ e_g\ \textbf{then}\ e_t\ \textbf{else}\ e_f : \rho \to e_f : \rho, 1^*}$$

$$\textbf{if}\ e_g\ \textbf{then}\ e_t\ \textbf{else}\ e_f : \rho \to e_t : \rho, 1^*$$

Function Application

$$\frac{}{e_1 e_2 : \rho \to e_1 : \rho, 1^*}$$

$$\frac{e_1 : \rho \Downarrow (\!|f : \mu|\!)}{e_1 e_2 : \rho \to e_2 : \rho, 1^*}\ {\scriptstyle (\!|f:\mu|\!) \in Closure'}$$

$$\frac{e_1 : \rho \Downarrow (\!|f : \mu|\!), \gamma_1 \quad e_2 : \rho \Downarrow v, \gamma_2}{e_1 e_2 : \rho \to Bind\ (\!|f : \mu|\!)\ w, \mathrm{app}_x(\gamma_1, \gamma_2)}\ {\scriptstyle (\!|f:\mu|\!) \in Closure'}$$

Figure 3.3: Dataflow: Calls

of the dynamic call graph. We do not wish to be tied to a particular approximation, however, so that we will describe the requirements for the approximation abstractly in this section. In the next chapter, several possibilities meeting these requirements will be presented.

3.4.1 Abstracting the Dynamic Call Graph

In order to soundly approximate the dynamic call graph, which may be infinite, we shall use the framework of *abstract interpretation* [CC77]. To this end we shall assume a *finite* set of *abstract states*[1]. The static call graph will then be defined by an approximation of the $\Downarrow$ and $\rightarrow$ judgements of the operational semantics, together with size-change information.

Abstract States

The abstract domain in our analysis will be a finite set of abstract states, together with an information order $\sqsubseteq$:

Property 3.29. *The given set AbsState of abstract states is finite. The relation $\sqsubseteq$ is a partial order on AbsState.*

An abstract state represents a set of concrete states. The $\sqsubseteq$ order compares abstract states by information content: if $s \sqsubseteq s'$, then the set of states represented by s is smaller than the set of states represented by s' (*i.e.* s encodes more information than s').

Readers familiar with abstract interpretation might expect a requirement that (*AbsState*, $\sqsubseteq$) be a (necessarily complete) lattice. This will not be necessary in our formulation; we shall return to this point in Section 4.3.

We further define an *abstraction* map $\alpha : \mathit{State} \rightarrow \mathit{AbsState}$ that maps concrete states to abstract states. The intention is that $\alpha(s)$ be the *most defined* abstract state representing s. In other words, $\alpha(s)$ should be the *least* (in the $\sqsubseteq$ order) element of *AbsState* representing s.

The Static Call Graph

We may now define the components of the static call graph. This is intended to approximate any possible dynamic call graph, for appropriate choices of values of the free variables. In addition, the call graph is annotated with size-change graphs to prove termination.

Definition 3.30. Fix an abstraction $\mathit{State} \xrightarrow{\alpha} \mathit{AbsState}$. A *static call graph* of a program P is a pair $\mathit{StaticCG} = (\rightarrow^{\alpha}, \Downarrow^{\alpha})$, where:

1. $\rightarrow^{\alpha} \subseteq \mathit{AbsState} \times \mathit{AbsState}$ is the abstract *call relation*.
2. $\Downarrow^{\alpha} \subseteq \mathit{AbsState} \times (\mathit{AbsState} \cup \mathit{Error})$ is the abstract *evaluation relation*.

[1] Finiteness of this set is a stronger condition than is necessary, and in particular widening operators (convergence accelerators) can be used to deal with infinite sets of abstract states.

The static call graph merely defines approximations to the call and reduction relations. However, to prove size-change termination of programs, we shall be more concerned with *annotated* call graphs, which include information about size changes in the form of size-change graphs.

Definition 3.31. Fix an abstraction $State \xrightarrow{\alpha} AbsState$ as above. An *annotated static call graph* is a quadruple $StaticCG = (\rightarrow^\alpha, \Downarrow^\alpha, \mathrm{gb}^\alpha, G)$, where:

1. $(\rightarrow^\alpha, \Downarrow^\alpha)$ is a static call graph,
2. For each abstract state s, $\mathrm{gb}^\alpha(s)$ is a set of environment paths, the *graph basis* of s, and
3. G assigns a set of size-change graphs to each call or evaluation in $\rightarrow^\alpha$ and $\Downarrow^\alpha$ (respectively). We write $G(s \rightarrow^\alpha s')$ and $G(s \Downarrow^\alpha v)$ for the sets of graphs associated with the respective calls and reductions.

The G function annotates each edge in the static call graph with information about size changes along this control flow. The intention is that *some* graph in $G(s \rightarrow^\alpha s')$ should safely describe (s, s'). In the sequel, we shall write size-change graph annotations in the same way as in the dataflow semantics of Section 3.3: the static call graph will be composed of edges of the form $s \rightarrow^\alpha s', \gamma$ and $s \Downarrow^\alpha v, \gamma$ for γ a size-change graph. This is equivalent, and for instance we can define $G(s \rightarrow^\alpha s') = \{\gamma \mid s \rightarrow^\alpha s', \gamma\}$. In the dataflow semantics only *one* graph was produced for any edge in the call graph, but this is a consequence of determinism and does not hold in general in static analyses.

3.4.2 Safety

We have defined the static call graph with the intention that it should approximate any relevant dynamic call graph. We shall now make this precise with the notion of *safety*. As static call graphs can be separated into the *control* aspect (approximating the dynamic call graph) and the *dataflow* aspect (defining size-change graphs), we will accordingly define safety for these in turn.

Control-safety of a static call graph indicates that it is a safe approximation to any possible dynamic call graph. As described above, due to our use of free variables to allow classes of programs to be analysed, control safety is relative to a choice of abstract values for free variables.

Definition 3.32. Let T be a program template with variables $X_1, \ldots, X_n$, and let $w_1, \ldots, w_n$ be abstract values. The set of dynamic call graphs for T that are *compatible with the abstract valuation* $\mathfrak{W} = \{X_i \mapsto w_i\}_{i=1}^n$ is the set of dynamic call graphs for T with valuations $\mathfrak{V} = \{X_i \mapsto v_i\}_{i=1}^n$, where $v_i \in Value$ for each i, and $\alpha(v_i) \sqsubseteq w_i$.

In other words, an abstract valuation assigning an abstract value to each free variable represents any concrete valuation assigning compatible concrete values. To indicate that a free variable X is intended to be restricted to values compatible with an abstract constant c, we will use the following notation:

... <X : c> ...

where the name of the variable may be elided if it is not needed elsewhere.

Definition 3.33 (Control Safety)**.** A static call graph $\mathit{StaticCG} = (\rightarrow^\alpha, \Downarrow^\alpha)$ for a program template T with free variables $X_1, \ldots, X_n$, is *control-safe with valuation* $\mathfrak{W} = \{X_i \mapsto w_i\}_i$, where w_i is an abstract value for each i, if: for any dynamic call graph for T that is compatible with $\mathfrak{W}$,

1. Whenever $s \rightarrow s'$ in the dynamic call graph, then if $t \sqsupseteq \alpha(s)$, there exists $t' \sqsupseteq \alpha(s')$ such that $t \rightarrow^\alpha t'$,
2. Whenever $s \Downarrow v$ in the dynamic call graph, $v \in \mathit{Value}$, if $t \sqsupseteq \alpha(s)$ there exists $w \sqsupseteq \alpha(v)$ such that $t \Downarrow^\alpha w$,
3. Finally, if $s \Downarrow e$ for $e \in \mathit{Error}$ in the dynamic call graph, and $t \sqsupseteq \alpha(s)$, then $t \Downarrow^\alpha e$.

The definition of control safety corresponds to our intuition that the static call graph should cover any possible choice of dynamic call graph.

We now define dataflow safety. Dataflow safety guarantees that the size-change graphs annotating the static call graph are indeed correct.

Definition 3.34 (Dataflow Safety)**.** Consider an annotated static call graph $\mathit{StaticCG} = (\rightarrow^\alpha, \Downarrow^\alpha, \mathrm{gb}^\alpha, G)$ for a program template T as before. Then $\mathit{StaticCG}$ is *dataflow safe* (with an appropriate valuation as above) if $\mathit{StaticCG}$ is control safe, and

- For each abstract state t and $\alpha(s) \sqsubseteq t$, $\mathrm{gb}^\alpha(t) \subseteq \mathrm{gb}(s)$.
- Whenever $\gamma \in G(t \rightarrow^\alpha t')$ (resp. $\gamma \in G(t \Downarrow^\alpha w)$), the in-basis of γ is $\mathrm{gb}^\alpha(t)$, and the out-basis of γ is $\mathrm{gb}^\alpha(t')$ (resp. $\mathrm{gb}^\alpha(w)$).
- Suppose that $\alpha(s) \sqsubseteq t$ and $\alpha(s') \sqsubseteq t'$ (resp. $\alpha(v) \sqsubseteq w$). Then if $t \rightarrow^\alpha t'$ (resp. $t \Downarrow^\alpha w$), then some $\gamma \in G(t \rightarrow^\alpha t')$ is safe for (s, s') (resp. some $\gamma \in G(t \Downarrow^\alpha w)$ is safe for (s, v)).

Dataflow safety guarantees that the size-change graph associated with an arrow in the static call graph is safe for *any* arrow in a dynamic call graph that this could represent. The condition on graph bases ensures that this assertion is well-formed.

3.5 The Size-Change Termination Criterion

We may now state the size-change termination condition precisely, and prove soundness of our termination analysis, given any safe abstraction of the dynamic call graph. We will

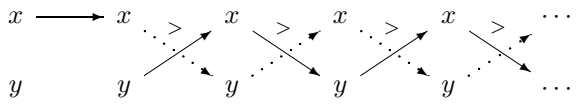


Figure 3.4: Multipaths and Threads

also briefly describe the algorithmic aspects of this test, due to Lee *et al* [LJBA01]. The definitions and results in this section are largely unchanged from Lee *et al* [LJBA01], but are presented for completeness, and to clarify the termination criterion used in SCT analyses.

3.5.1 Infinite Descent and the SCT Criterion

A program is size-change terminating with respect to a given (annotated) static call graph if any infinite path through the call graph leads to infinite descent in some value appearing in size-change graphs. We shall make this precise below.

Definition 3.35 (Multipaths)**.** A *multipath* $\mathscr{M}$ is a finite or infinite sequence of size-change graphs $\mathscr{M} = \gamma_1, \gamma_2, \ldots.$ such that the out-basis of γ_i equals the in-basis of γ_{i+1} for each i. A *thread through* $\mathscr{M}$ is a finite or infinite sequence of triples (x_i, l_i, x_{i+1}), where x_i lies in the in-basis of γ_i (so x_{i+1} lies in the out-basis of γ_i, necessarily) and $\gamma_i(x_i, x_{i+1}) = l_i$.

Note that a thread through a multipath need not start at the first size-change graph of the multipath. As an example, consider the multipath shown in Figure 3.4 (with $\geq$ labels omitted for clarity). An infinite thread is shown in dotted lines in this example.

A thread is infinitely decreasing if it consists entirely of $\geq$ and $>$ arrows, and infinitely many of the arrows indicate decreases. A multipath is then said to cause infinite descent if it has an infinitely decreasing thread.

Definition 3.36 (Infinite descent)**.** A multipath $\mathscr{M}$ has the *infinite descent* property if some infinite thread $(x_i, l_i, x_{i+1})_{i \geq i_0}$ in $\mathscr{M}$ satisfies: each l_i is one of $>$, $\geq$; and infinitely many of the l_i are $>$.

The thread shown in Figure 3.4 is infinitely decreasing, and so the multipath has infinite descent.

The purpose of the infinite descent property for multipaths is that multipaths with the infinite descent *cannot* occur in an actual execution of the program, as this would cause some value to decrease infinitely in a well-founded order. Correspondingly, we define:

Definition 3.37 (SCT)**.** A static call graph $\mathit{StaticCG} = (\rightarrow^\alpha, \Downarrow^\alpha, \mathrm{gb}^\alpha, G)$ *has the SCT property* if: whenever there is an infinite sequence of calls $(s_i \rightarrow^\alpha s_{i+1}, \gamma_i)_{i \geq 0}$ in *StaticCG*, then the multipath $\gamma_0, \gamma_1, \gamma_2, \ldots$ has the infinite descent property.

A static call graph thus has the SCT property if the graphs annotating any infinite call sequence form a multipath with infinite descent property. Note that the graphs γ_i in the above

definition are guaranteed to form a multipath, whenever *StaticCG* is safe (the out-basis of γ_i necessarily matches the in-basis of γ_{i+1}).

The SCT property does yield termination. We first show that multipaths with infinite descent cannot occur in the evaluation of a program:

Lemma 3.38. *Let $(s_i)_{i\geq 0}$ be an infinite sequence of states. Then there is no multipath $\mathscr{M} = \gamma_0, \gamma_1, \ldots$ with the infinite descent property such that γ_i is safe for (s_i, s_{i+1}).*

Proof. Suppose for a contradiction that each γ_i is safe for (s_i, s_{i+1}). As $\mathscr{M}$ has the infinite descent property, there is an infinite thread $(x_i, l_i, x_{i+1})_{i\geq i_0}$ in $\mathscr{M}$ that is infinitely decreasing, so each l_i is one of $>$ or $\geq$, and infinitely many of the l_i are $>$. Then as γ_i is safe for (s_i, s_{i+1}), for each $i \geq i_0$, $s_i \nabla x_i \geq s_i \nabla x_{i+1}$. Furthermore, infinitely many of the inequalities are strict. This is a contradiction as the $>$ relation is well-founded, by Proposition B.3. □

Theorem 3.39 (Soundness)**.** *Let T be a program template with free variables $X_1, \ldots, X_n$, and suppose that StaticCG is a dataflow safe static call graph for T under valuation $\mathfrak{W} = \{X_i \mapsto w_i\}_{i=1}^n$. Then any instance of T with valuation compatible with $\mathfrak{W}$ terminates.*

Proof. Let $\mathfrak{V} = \{X_i \mapsto v_i\}_{i=1}^n$, where $v_i \in \mathit{Value}$ for each i be a valuation that is compatible with $\mathfrak{W}$, and suppose that T, with valuation $\mathfrak{V}$, is nonterminating. By Property 2.14 (p. 22), there is an infinite sequence of calls $(s_i \to s_{i+1})_{i\geq 0}$ in the dynamic call graph of T with $\mathfrak{V}$.

By control safety of *StaticCG* we may construct a sequence of abstract states $(t_i)_{i\geq 0}$ such that $t_i \sqsupseteq \alpha(s_i)$ and $t_i \to^\alpha t_{i+1}, \gamma_i$ for some γ_i, for each i, as follows. Put $t_0 = \alpha(s_0)$. Then for each i, given $t_i \sqsupseteq \alpha(s_i)$, as $s_i \to s_{i+1}$, by control safety we may find $t_{i+1} \sqsupseteq \alpha(s_{i+1})$ such that $t_i \to^\alpha t_{i+1}$. Let $S_i = G(t_i \to^\alpha t_{i+1})$.

We now use dataflow safety of *StaticCG* to construct a multipath that covers the infinite trace $(s_i)_{i\geq 0}$. Pick $\gamma_i \in S_i$ safe for (s_i, s_{i+1}) for each i, and let $\mathscr{M} = \gamma_0, \gamma_1, \ldots$. As before $\mathscr{M}$ is a multipath: the out-basis of γ_i is $\mathrm{gb}^\alpha(t_{i+1})$, as is the in-basis of γ_{i+1} (by dataflow safety).

We may now apply Lemma 3.38 to conclude that $\mathscr{M}$ does not have the infinite descent property, as required. □

The SCT criterion is therefore indeed a sound test for termination. In the next section we shall see how it may be implemented, given a static call graph.

3.5.2 Algorithm

The SCT property for static call graphs implies program termination (though the reverse implication does not hold). Moreover, as we shall now see, this property is decidable. It is thus an appropriate basis for a static analysis to detect termination. We shall describe one of the methods for deciding this property, though other algorithms have been given [LJBA01].

Throughout, the *set of graphs* in a static calls graph will be the set of triples (s, γ, s'), where $s \to^\alpha s'$ and $\gamma \in G(s \to^\alpha s')$. We shall call such a triple (s, γ, s') a *labelled size-change*

graph, and define composition as expected:

$$(s, \gamma, s'); (s', \delta, s'') = (s, \gamma; \delta, s'')$$

Composition of graphs whose middle labels do not match is undefined. Note that in the above the graph composition $\gamma; \delta$ is well-defined — the out-basis of γ is $\mathrm{gb}^\alpha(s')$, and thus equal to the in-basis of δ.

Definition 3.40. The *closure* of a set S of labelled size-change graphs is defined as:

$$\begin{aligned} \mathrm{Cl}(S) &= \{G_1; G_2; \ldots; G_n \mid (\forall i) G_i \in S \wedge t_i = s_{i+1}\} \\ &\quad \textbf{where } G_i = (s_i, \gamma_i, t_i) \text{ for each } i \end{aligned}$$

Lemma 3.41. *If S is a finite set of labelled size-change graphs, then Cl(S) is finite.*

Proof. Any labelled graph in $\mathrm{Cl}(S)$ must be of the form (s, γ, t), where there are labelled graphs (s, δ_1, s') and (t', δ_2, t) in S. The set of labels that may appear in $\mathrm{Cl}(S)$ is thus finite.

Furthermore, the in-basis of γ is $\mathrm{gb}^\alpha(s)$, while its out-basis is $\mathrm{gb}^\alpha(t)$. There are only finitely many graphs with given in- and out- bases: each arrow in γ is of the form (x, l, y) where $x \in \mathrm{gb}^\alpha(s)$, $y \in \mathrm{gb}^\alpha(t)$ and $l \in \{>, \geq\}$. There are thus finitely many possible labelled graphs that may appear in $\mathrm{Cl}(S)$. □

It is therefore possible to compute $\mathrm{Cl}(S)$ from a finite set S by enumerating all possible compositions, though more efficient methods are available as this amounts to computing the transitive closure of a relation.

Definition 3.42. A labelled size-change graph (a, γ, b) is a *self-call* if $a = b$. A self-call is *idempotent* if $\gamma; \gamma = \gamma$. Finally, a self-call has an *in-situ decrease* if $\gamma(x, x) = >$ for some x.

This yields an effectively computable procedure for checking the SCT property:

Theorem 3.43. *A static call graph with set of labelled graphs S has the SCT property iff every idempotent self-call in Cl(S) has an in-situ decrease.*

Proof. The proof of correctness of this criterion is due to Lee *et al* [LJBA01]. We reproduce the proof in our setting for completeness, and as a useful aid to the intuition behind this criterion.

Throughout this section we shall denote an arbitrary multipath by $\mathscr{M} = \gamma_0 \gamma_1 \cdots$, where $\gamma_i : s_i \to s_{i+1}$. Given such a multipath, we let $\gamma_i^j = \gamma_i; \cdots; \gamma_{j-1}$.

> **Lemma.** *With $\mathscr{M}$ as above, $\gamma_i^j(x, y) = >$ iff there is a decreasing thread from x to y starting at i and ending at j. Furthermore $\gamma_i^j(x, y) \succcurlyeq \geq$ iff there is a nonincreasing thread from x to y starting at i and ending at j.*

We fix i and proceed by induction on $j > i$. The base case $j = i + 1$ is trivial.

Let us now prove the inductive case. We write $\text{Thread}_i^j(x, y)$ to mean that there is a nonincreasing thread from x to y starting at i and ending at j, and $\text{ThreadDec}_i^j(x, y)$ if the thread is decreasing. Then:

$$
\begin{array}{ll}
 & \gamma_i^{j+1}(x,y) = > \\
\iff & \{\text{definition of } \gamma_i^{j+1}\} \\
 & (\gamma_i^j ; \gamma_j)(x,y) = > \\
\iff & \{\text{ composition}\} \\
 & (\exists z)(\gamma_i^j(x,z) = > \wedge \gamma_j(z,y) \succcurlyeq \geq) \vee (\gamma_i^j(x,z) \succcurlyeq \geq \wedge \gamma_j(z,y) = >) \\
\iff & \{\text{inductive hypothesis}\} \\
 & (\exists z)(\text{ThreadDec}_i^j(x,z) \wedge \gamma_j(z,y) \succcurlyeq \geq) \vee (\text{Thread}_i^j(x,z) \wedge \gamma_j(z,y) = >) \\
\iff & \{\text{property of threads}\} \\
 & \text{ThreadDec}_i^{j+1}(x,y)
\end{array}
$$

as required. The proof that $\gamma_i^{j+1}(x, y) \succcurlyeq \geq$ iff $\text{Thread}_i^{j+1}(x, y)$ is similar and omitted. □

($\Longrightarrow$) Suppose first that some idempotent self-call in $\text{Cl}(S)$ has no in-situ decrease, say $\gamma \in \text{Cl}(S)$ has no in-situ decrease. Then $\gamma = \gamma_0; \cdots ; \gamma_{N-1}$, where $\gamma_i : s_i \to s_{i+1}$ for each i. Let $\mathscr{M} = (\gamma_0 \ldots \gamma_{N-1})^\omega$. We show that $\mathscr{M}$ has no infinitely decreasing thread.

Suppose for a contradiction that $\langle (x_i, l_i, x_{i+1}) \rangle_{i \geq i_0}$ is an infinitely decreasing thread. Consider the (infinite) sequence of labels $\langle x_{jN} \rangle_j$. Then as the set of elements of this sequence is finite, some element must appear infinitely often, say $x = x_{j_0 N} = x_{j_1 N} = \cdots$, where $j_0 < j_1 < \cdots$. As this thread is infinitely decreasing, for each i, $l_i \succcurlyeq \geq$, while $l_i = >$ for infinitely many i.

Thus for some j_k, there is necessarily a $>$ label in the thread between $x_{j_0 N}$ and $x_{j_k N}$. Then if $\delta = \gamma_{j_0 N}^{j_k N}$, $\delta(x, x) = \delta(x_{j_0 N}, x_{j_k N}) = >$. That is, δ has an in-situ decrease. However, $\delta = \gamma_{j_0 N}^{j_1 N}; \gamma_{j_1 N}^{j_2 N}; \cdots ; \gamma_{j_{k-1} N}^{j_k N} = \gamma^{j_k - j_0} = \gamma$ as γ is idempotent. Hence γ has an in-situ decrease, a contradiction.

($\Longleftarrow$) Now suppose that $\mathscr{M}$ is a multipath with no infinitely decreasing threads. We construct an idempotent graph $\gamma \in \text{Cl}(S)$ such that γ has no in-situ decrease. Let $\mathscr{M} = \gamma_0 \gamma_1 \cdots$, where $\gamma_i : s_i \to s_{i+1}$ for each i.

We appeal to Ramsey's theorem. Let $S = \{\{x, y\} \subseteq \mathbb{N} \mid x \neq y\}$ be the set of two-element sets of natural numbers. Let $g(\{x, y\}) = \gamma_x^y$, where $x < y$ wlog. Then $g(\{x, y\})$ is a size-change graph of type $s_x \to s_y$.

We may now partition S according to the equivalence relation generated by g: $A \sim B$ iff $g(A) = g(B)$. The range of g is finite (the set of size-change graphs is finite), so the set of equivalence classes is finite.

By Ramsey's theorem, there is an infinite set A of natural numbers such that all $\{x, y\} \in S$ such that $x, y \in A$ lie in the same equivalence class. That is, for some graph γ, $\gamma_x^y = \gamma$ whenever $x, y \in A$ and $x < y$.

We now show that γ is an idempotent self-loop. Suppose that $\gamma : s \to t$.

Pick $a < b < c \in A$ (as A is infinite). Then $\gamma = g(\{a, b\}) = g(\{a, c\}) = g(\{b, c\})$. Thus $\gamma : s_a \to s_b$ and $\gamma : s_b \to s_c$. We can deduce that $s = s_a = s_b$, while $t = s_b = s_c$. Hence $s = t$ and γ is a self-loop.

Furthermore,

$$\begin{aligned}
\gamma &= g(\{a, c\}) \\
&= \gamma_a; \cdots ; \gamma_{c-1} \\
&= (\gamma_a; \cdots ; \gamma_{b-1}); (\gamma_b; \cdots ; \gamma_{c-1}) \\
&= g(\{a, b\}); g(\{b, c\}) \\
&= \gamma^2
\end{aligned}$$

as required.

Finally, we show that γ has no in-situ descent. Suppose for a contradiction that $\gamma(x, x) = >$. Let $\{a_0, a_1, \ldots\} = A$, where $a_0 < a_1 < \cdots$. Then as $\gamma = \gamma_{a_i}^{a_{i+1}}$ for each i, and $\gamma(x, x) = >$, for each i there is a decreasing thread from x to x starting at a_i and ending at a_{i+1}. The composition of these yields an infinitely decreasing thread, a contradiction. □

Complexity of Size-Change Termination

The graph closure algorithm shown above is straightforward and directly leads to an implementation. However, as we shall see its time and space complexity is rather high. We let $C(k, N)$ be the maximum size of the closure of a set of graphs of size N, where the graph basis of each graph has size k. For simplicity, we shall restrict our attention to sets of graphs in which all labels are identical (and thus all graph bases also), so that all graphs are composable.

Lemma 3.44. *For any N, k, $C(k, N) \leq 3^{k^2}$.*

Proof. Each graph with graph basis S of size k is determined by its representation as a function $\gamma : S \times S \to \{>, \geq, \bot\}$. There are $3^{|S| \cdot |S|} = 3^{k^2}$ such functions, so the size of any set of graphs with bases S is bounded by 3^{k^2}. □

This theoretical maximum is a bound on any set of graphs, not merely the sets of graphs that arise as closures. However, as the following shows the sizes of closures are not substantially smaller:

Lemma 3.45. *For any k, $C(k, k-1) \geq k!$*

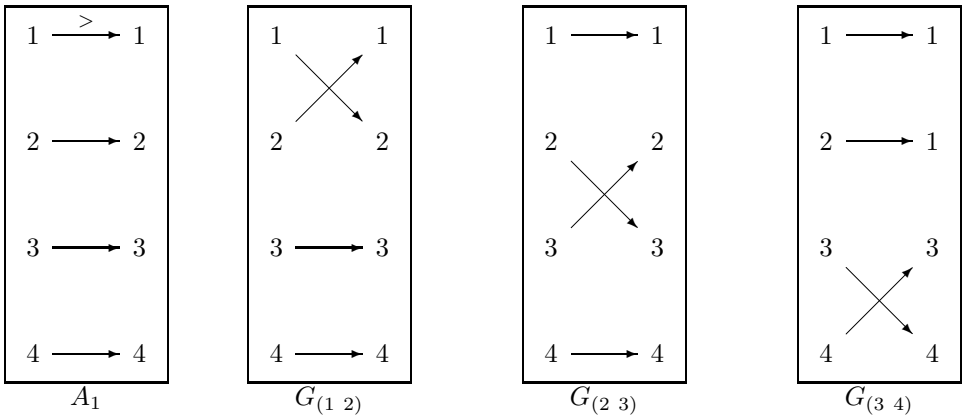


Figure 3.5: Proof of Lemma 3.46

Proof. The crucial observation is that we may represent permutations on $\{1, \ldots, k\}$ as size-change graphs: if σ is such a permutation, the graph G_σ has arrows $\{(i, \geq, i\sigma) \mid 1 \leq i \leq k\}$. Furthermore, $G_\sigma = G_\tau$ iff $\sigma = \tau$, and $G_\sigma; G_\tau = G_{\sigma\tau}$. We therefore have an embedding of the symmetric group S_k in the set of graphs with graph basis of size k.

We now note that S_k is generated by the set of transpositions of adjacent elements $T = \{(i\ i+1) \mid 1 \leq i < k\}$. It follows that the set of graphs $G_T = \{G_\sigma \mid \sigma \in T\}$, of size $k-1$, generates all $k!$ permutations in S_k, as required. □

We may improve on this bound slightly by producing graphs with decreasing edges in additions to straightforward permutations.

Lemma 3.46. *For any k, $C(k, k) \geq 2^k k!$.*

Proof. Define graphs $A_j = \{(i, \geq, i) \mid 1 \leq i \leq k \wedge i \neq j\} \cup \{(j, >, j)\}$. Put $X = G_T \cup \{A_1\}$ where G_T is as before. Then $|X| = k$. The set X (for $k = 4$) is shown in Figure 3.5.

First, we note that if $\sigma_j = (1\ j)$, then $G_{\sigma_j}; A_1; G_{\sigma_j} = A_j$. $\mathrm{Cl}(X)$ therefore contains all A_j. Furthermore, as above $\mathrm{Cl}(X)$ contains all G_σ for permutations σ.

Now fix a permutation σ. For any subset $U \subseteq \{1, \ldots, k\}$, let A_U be the composition of the A_j for $j \in U$ (this is well-defined as the A_j commute). Then $G_\sigma; A_U$ is a distinct graph for each U, as the set U is precisely the set of target nodes in $G_\sigma; A_U$ that are reached by a decreasing arrow. Furthermore, for any τ and V, $G_\tau; A_V = G_\sigma; A_U$ implies $\tau = \sigma$, as the permutation may be recovered by ignoring edge labels.

We therefore find that for each σ, $\mathrm{Cl}(X)$ contains a graph for each subset of $\{1, \ldots, k\}$, and that these are all pairwise distinct. Thus $|\mathrm{Cl}(X)| = 2^k k!$, as required. □

The exponential complexity of the closure-based SCT algorithm is not surprising given the following complexity result for the SCT problem [LJBA01].

We consider the following problem (SCT):

1. *Input*: a multigraph $\mathcal{G}$, with each edge labeled with a size-change graph.

2. *Decision problem*: to recognise those graphs $\mathcal{G}$ such that any infinite path through $\mathcal{G}$ has the infinite descent property.

3. *Input size*: the size of the input is the sum of the number of edges in size-change graphs. Note that this is necessarily greater than the number of edges in $\mathcal{G}$.

Lee *et al* [LJBA01] prove the following:

Theorem 3.47. *The SCT problem can be decided in PSPACE.*

Proof sketch. Define *FLOW* to be the set of infinite paths through $\mathcal{G}$, and *DESC* to be the set of infinitely decreasing paths through $\mathcal{G}$. The SCT problem is then equivalent to $FLOW = DESC$.

The set *FLOW* can be accepted by a Büchi automaton. It suffices to build the automaton isomorphic to $\mathcal{G}$ (that is, with a state for each node in $\mathcal{G}$ and transitions corresponding to edges) in which each state is accepting.

Lee *et al* show that *DESC* can likewise be accepted by a Büchi automaton. The automaton constructed for this purpose contains an isomorphic copy of the call graph, together with a pair of states for each node label (environment path) appearing in a size-change graph. In addition to transitions arising from $\mathcal{G}$ the automaton contains a transition for each edge in a size-change graph.

Both automata have size polynomial in the size of the input (that is, the sum of the sizes of size-change graphs). As the SCT problem reduces to the equivalence of these automata, and as this is decidable in PSPACE, the result follows. □

Furthermore, this complexity result is sharp:

Theorem 3.48. *The SCT problem is PSPACE-hard.*

Proof sketch. The proof, again due to Lee *et al*, proceeds by reduction of the termination problem for Boolean programs to the SCT criterion. The details of the construction would lead us too far astray, and we merely state the results.

Given a Boolean program with m instructions and k variables, an annotated static call graph $\mathcal{G}$ may be constructed with $m + 1$ nodes, at most $2m$ edges and $2k + 1$ distinct environment paths. The size of the static call graph is thus polynomial in m and k, and so polynomial in the size of the boolean program.

This static call graph $\mathcal{G}$ has the following property: the input Boolean program is terminating iff $\mathcal{G}$ does not have the SCT property. As the termination problem for Boolean programs is PSPACE-hard, and PSPACE is closed under complementation, this yields the desired result. □

We can therefore conclude that the SCT problem is PSPACE-complete in the size of the input static call graph. In practice however we find that SCT admits an efficient implementation when applied to natural programs. This may be attributed to two factors. First,

though the call graph may grow large, the number of different self-calls from a node to itself is expected to be small. The exponential blowup is therefore confined to small subsets of the annotated call graph. Furthermore, self-calls with permuted arguments are rather rare in real programs, and hence the $k!$ factor in the bound shown above is rarely obtained in practice.

3.6 References

The order on functional values is due to Jones and Bohr [JB04]; this excludes the case of constants, both primitive and algebraic. The order relation used by Jones and Bohr includes an extra case: $e_1 : \rho < e_2 : \rho$ whenever e_1 is a subexpression of e_2. This is not included in our presentation as this order cannot be used to prove termination in our context (a function call can never be to a subexpression). The alternative would be to order functions by textual order, but this does not appear natural.

We have introduced the notion of dataflow graphs tracking exact dataflow between values (but not size changes). This is an extension of the annotated semantics of Jones and Bohr. However, Jones and Bohr restrict the size-change graphs in the annotated semantics to immediate parameters. This precludes the use of some more precise abstractions to improve precision (as shown in the next chapter). Likewise, the (general) notions of graph basis and environment paths were formulated by the author and Neil Jones, and are not present in Jones and Bohr [JB04]. The purpose of our generalised framework is to allow the control-flow abstraction to be varied. The basis of our control-flow analyses is abstract interpretation [CC77]. Finally, we note that the analysis of Jones and Bohr does not deal with free variables or arbitrary unknown values in λ-expressions.

The size-change termination algorithm is due to Lee, Jones and Ben-Amram [LJBA01]; our presentation is essentially unchanged. The lower bounds that we have given for the size of the closure set do not, as far as we are aware, appear in a publication. The PSPACE-completeness result is also due to Lee *et al*; Lee [Lee02a, BAL06] gives quadratic and cubic approximations to the SCT test.

It has been suggested that *defunctionalisation* [Rey72] might provide an alternative to defining a new termination analysis for higher-order languages. Any straightforward defunctionalisation algorithm will, in general, produce a program that acts on a syntactic representation of functions as closures — thus the first-order SCT analysis [LJBA01] would recover our order on functional values. However, standard control-flow analysis for first-order programs is entirely inadequate when dealing with such programs, and so a specific analysis would be necessary nonetheless.

Chapter 4

Static Analysis

We have in Chapter 2 defined the semantics of our core language, and the dynamic call graph of a program. Chapter 3 saw the addition of an order on values, and the annotated dynamic call graph reflecting dataflow information.

We shall now turn to the problem of defining an abstraction and static analysis to compute a (suitable) static call graph for a program, satisfying the safety conditions given in Section 3.4. This can then be used to prove termination using the SCT criterion and associated algorithm described in Section 3.5.

In this aspect of the analysis we have a certain amount of freedom, and wish to define several static analyses. This should allow the user some control over the complexity / precision tradeoff inherent to such static approximations of a program's semantics. Accordingly, we define three static analyses. The first of these is based on 0CFA [Shi91], and is thus an instantiation of a known control-flow analysis. The 0CFA analysis is a minimal analysis, offering the lowest complexity at the cost of least precision.

Precision may be improved by keeping more information about environments to distinguish states in the static call graph, leading to the *k-bounded CFA* analysis. This is inspired by, but distinct from, the traditional notion of k-CFA. The k-bounded CFA analysis is particularly useful to solve some of the precision problems that occur through the common use of higher-order functions such as *fold* or *map*.

Finally, we present a novel static analysis, based on a representation of environments as tree automata. The aim of this analysis is to handle the case of programs that create environments of unbounded depth more naturally and more precisely than in k-bounded CFA analyses.

Outline The structure of this chapter is as follows. In Section 4.1 we introduce the general ideas underlying all three control flow analyses presented in this chapter. The following section (Section 4.2) is concerned with adapting the dataflow graphs of Section 3.3 to size-change graphs for proving termination. Section 4.3 recasts our definitions in the framework

of abstract interpretation slightly more formally.

Following these general results, the rest of the chapter is devoted to the description of the three static analyses that we propose: the existing 0CFA analysis (Section 4.4), k-bounded CFA (Section 4.5) and the novel tree automata analysis (Section 4.6). In each of these sections, the analysis is presented first informally and then in detail, before it is proved sound. Finally, each section ends with a discussion of the merits and drawbacks of the analysis described.

4.1 Generalities

4.1.1 Abstract States

In this chapter we will be concerned with mapping the (infinite) set of concrete states to a finite set of abstract states, each abstract state representing a set of concrete states. It is therefore useful to examine the steps that are involved in such a mapping, as the broad features of all abstractions will be similar.

We recall that states are finite *trees*, with node labels corresponding to the different types of states: $(\!|f|\!)$ for a closure state, where f is a function name or primitive operator; e for an expression state, where e is an expression; C for a constructor state and c for a primitive constant state. The constant node labels only appear on leaf nodes, while all other labels may appear on internal or leaf nodes.

We can therefore see that the set of concrete states is infinite through two separate causes:

1. There are infinitely many primitive constants in general, and thus infinitely many possible labels for leaf nodes.

2. The *depth* of trees representing states is unbounded.

We shall address these two issues in this chapter to design an abstraction of states. The problem of abstracting constants will be dealt with first, as it is of less importance — in the size-change termination method, a precise abstraction of primitive constant values is not required. The main challenge is the abstraction of states of unbounded depth, as this has a significant impact on precision. As an example, consider the following program:

```
compose f g x = f (g x)
id x = x
succ x = x + 1

toChurch n f =
  if n = 0 then id else compose f (toChurch (n-1) f)

toChurch <X> succ
```

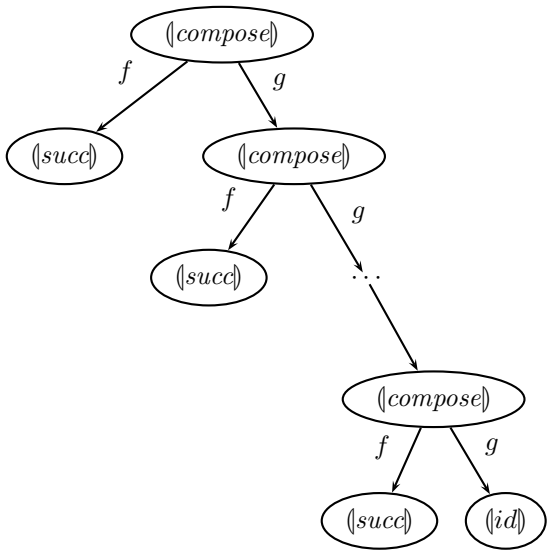


Figure 4.1: Environment Trees of Unbounded Depth

This program computes the Church numeral representation of a natural number n, and applies it to the successor function. It is straightforward to see that the result of evaluating this program for a particular value $X = n$ is a tree of the form shown in Figure 4.1, with depth n. The dynamic call graph for this program template (for all possible values of X) therefore includes states of unbounded depth. It is this situation that we must deal with in the sequel.

4.1.2 Constants

We shall assume that we have defined a finite set *AbstConst* of abstract primitive constants, together with an information order $\sqsubseteq$. As before we further require an abstraction function $\alpha : \mathit{PrimConst} \to \mathit{AbstConst}$. Finally, it is convenient to assume a top element $\top_c$ of the set *AbstConst*, representing any possible concrete constant.

In addition to defining abstract constants we must define the action of each primitive operator on abstract constants. Recall that in the exact semantics, the meaning of primitive operators is given by the *Apply* function, such that *Apply op cs*, where *op* is an operator and *cs* a list of values, is *either* a constant *or* an error value.

The behaviour of each primitive operator must be approximated in the abstract interpretation. To justify our framework for this, consider the operator $-$. In the exact semantics, $2 - 1$ evaluates to 1, while $0 - 1$ evaluates to *PrimopErr*. Let us consider the operation of $-$ on abstract constants by example: what should the value of $\top_c - \top_c$ be? It can readily be seen that as each occurrence of $\top_c$ represents any constant, the result can be any natural number, or *PrimopErr* (covering cases such as $0 - 1$). This justifies the following definition:

Definition 4.1. The operation of primitive operators on constants is defined by a function $\mathit{Apply}^\alpha : \mathit{PrimOp} \to \mathit{AbstConst}^* \to \mathit{AbstConst}$ and a predicate *MayFail* taking an operator and a list of abstract constants. This is *safe* if for any operator *op* and abstract constants $cs = \langle c_1, \ldots, c_n \rangle$ $(n = \sharp op)$, if $\alpha(d_i) \sqsubseteq c_i$ for each i then

1. $\mathit{Apply}\ op\ ds \in \gamma(\mathit{Apply}^\alpha\ op\ cs)$ if $\mathit{Apply}\ op\ ds$ is not an error, and
2. $\mathit{MayFail}(op, cs)$ otherwise.

Furthermore, we shall require that Apply^α be monotonic, where the order on $\mathit{AbstConst}^*$ is defined pointwise. Similarly, we require that *MayFail* be monotonic as a function into $\{\mathrm{T}, \mathrm{F}\}$ where $\mathrm{T} \sqsubseteq \mathrm{F}$.

This definition merely states that the behaviour of operators in the abstract world should soundly approximate their behaviour in the concrete world, as expected. The monotonicity requirement further adds that on more precise inputs, these functions should produce more precise results, which is certainly natural.

Example 4.2. We consider a simple abstract domain $\{\mathbb{N}, \mathbb{N}_{>0}, \top_c\}$, each constant representing the obvious set, and define the behaviour of $-$ on this domain. First, note that $\mathit{MayFail}(-, \langle c_1, c_2 \rangle)$ should hold for all abstract constants c_1 and c_2. For, all the abstract constants include 1 and 2, so any operation $c_1 - c_2$ could represent $1 - 2 = \mathit{PrimopErr}$. Also, for any c_1, c_2, $\mathit{Apply}^\alpha\ -\ \langle c_1, c_2 \rangle = \mathbb{N}$. This is always safe (if the $-$ operation succeeds, it is guaranteed to return a natural number) and is indeed the most precise value possible — to see this, note that all abstract constants include the constant 1, and therefore $0 = 1 - 1$ is a possible return value.

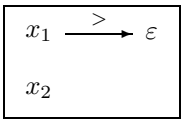
For our purposes, merely approximating the return value of primitive operators will not be enough, however. As we are concerned with size-change termination, it is crucial to be able to infer facts such as: "the result of evaluating $x - 1$ is less than the value of x". We make this precise below:

Definition 4.3. The size-change characteristics of primitive operators is captured by the $\mathit{SizeChange}_{\mathrm{op}}$ function such that $\mathit{SizeChange}_{\mathrm{op}}\ op\ \langle d_1, \ldots, d_n \rangle$ (where $n = \sharp op$) is a size-change graph γ with in-basis $\{x_1, \ldots, x_n\}$ and out-basis $\{\varepsilon\}$. This is *safe* if: whenever $\alpha(d_i) \sqsubseteq c_i$ for each i, and $\mathit{Apply}\ op\ \langle d_1, \ldots, d_n \rangle = r$, then γ is safe for $(\!|op : \{x_i \mapsto d_i\}_{i=1}^n|\!), r)$.

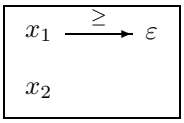
Note that this definition only requires size-change graphs to be correct when the operation on exact constants succeeds (does not return an error).

Example 4.4 (Example 4.2 continued)**.** Let us consider the size-change graphs that may be generated for the $-$ operator. This justifies the inclusion of $\mathbb{N}_{>0}$ as an abstract constant. For, if a is any concrete constant and b is represented by $\mathbb{N}_{>0}$, if $a - b$ succeeds, its value is guaranteed to be less than the value of a. On the other hand, every abstract constant other

than $\mathbb{N}_{>0}$ includes 0, so any operation $c_1 - c_2$ in the abstract domain, where c_2 is not $\mathbb{N}_{>0}$, could represent $a - 0 = a$. We therefore have the following size-change graphs:



$x_1 - x_2$
$x_2 \sqsubseteq \mathbb{N}_{>0}$



$x_1 - x_2$
otherwise

Beyond these general requirements we shall not elaborate further on the choice of abstract domain for constants. With the exceptions of some adjustments, such as including the set of positive natural numbers $\mathbb{N}_{>0}$ as a constant, which are essential to allow observable decreases when common operators such as $-$ are applied, the choice of abstract domain for constants has little impact on the precision of the analysis.

4.2 Finding Size-Change Termination

In Section 3.3, we have defined an instrumentation of the operational semantics of our language to track *dataflow*. This gives rise to a pair of semantic judgements $s \Downarrow v, \gamma$ and $s \to s', \gamma$ where γ is safe for (s, v) (resp. (s, s')).

Our aim is now to construct the static call graph of a program, and in the remainder of this chapter we shall show several different approximations of the call graph to achieve this. Furthermore, the static call graph is annotated with size-change graphs to track both dataflow and size changes.

The graph constructions of Section 3.3 may be used directly to construct dataflow graphs in the static call graph, and we therefore do not need to redefine safe dataflow graphs. The graphs annotating the static call graph will approximate those annotating the dynamic call graph.

Size Changes The graphs defined in the annotated semantics track dataflow, but in order to define the size-change termination analysis we must also consider *size changes*. We recall that we are able to compare two kinds of values:

- Primitive constants, for which we assume a given order
- Higher-order values, compared using the subtree order

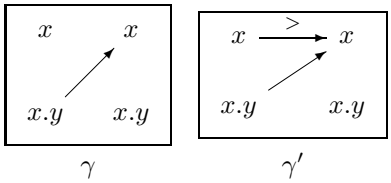
We may produce size-change graphs tracking the sizes of primitive constants using the $\mathit{SizeChange}_{\mathrm{op}}$ function, which generates size-change information for each primitive operator. As we shall see in the following, size changes for higher-order values are implied by the dataflow graphs already defined.

4.2.1 Completion of Graphs

The dataflow graphs defined in the annotated semantics do not include $>$ edges indicating decreases. It may therefore seem that these graphs do not form a useful basis for size-change termination — if no decreases are observed, we cannot hope to prove size-change termination.

However, such dataflow graphs may *imply* decreases between values, though these are not explicitly represented in the graph. We shall introduce the operation of *completion*, which restores any implied decreases as explicit information in the graph. Let us first consider an example:

Example 4.5. Consider the graph basis $A = \{\langle x\rangle, \langle x, y\rangle\}$ and let γ be the graph γ with in- and out-bases A and single arrow $(\langle x, y\rangle, \geq, \langle x\rangle)$. This graph contains no decrease labels. However, consider any (s, s') for which γ is safe. Then $s'\nabla\langle x\rangle \leq s\nabla\langle x, y\rangle$. Now $v = s\nabla\langle x, y\rangle = (s\nabla\langle x\rangle)\nabla\langle y\rangle$, so v is a strict substate of $w = s\nabla\langle x\rangle$. As our order on states is the subtree order, we deduce that $v < w$. Putting all of this together, $s'\nabla\langle x\rangle \leq s\nabla\langle x, y\rangle = v < w = s\nabla\langle x\rangle$. We may conclude that the graph γ' is also safe for (s, s'), where γ and γ' are shown below:



Generalising this particular example, we may observe the following. Suppose that a size-change graph contains an edge $(p, \geq, q)$. Then for any pair of states (s, s') for which this graph is safe, $s'\nabla q$ is a subtree of or equal to $s\nabla p$. Then for any proper prefix p' of p, as $s\nabla p' > s\nabla p$, the decreasing edge $(p', >, q)$ is valid.

A dataflow graph may therefore imply size-change information. We make this explicit using the *completion* operator on graphs, which restores any implied information.

Definition 4.6. Let $\gamma = (A, \Gamma, B)$ be a size-change graph. Then the *completion* of γ is $\overline{\gamma} := (A, \overline{\Gamma}, B)$, where:

$$\begin{aligned}\overline{\Gamma} \quad = \quad & \Gamma \cup \{(p, >, q) \mid (p +\!\!+ r, l, q) \in \Gamma \text{ for some } l \text{ and } r \neq \varepsilon\} \\ & \cup \{(p, >, q +\!\!+ r) \mid r \neq \varepsilon \wedge q +\!\!+ r \in B \wedge (p, l, q) \in \Gamma \text{ for some } l\}\end{aligned}$$

A graph γ is *complete* if $\overline{\gamma} = \gamma$.

That is, a complete graph explicitly represents all the size-change information that is entailed by it.

The following lemma (graph completion preserves safety) justifies our claim that completion does not add information not already implied by a size-change graph:

Lemma 4.7. *Let γ be a size-change graph, and suppose that γ is safe for (s, s'). Then $\overline{\gamma}$ is safe for (s, s').*

Proof. Suppose that $(x, l, y) \in \overline{\gamma}$. If $(x, l, y) \in \gamma$ then safety is given. Otherwise, there are two cases.

First suppose that the edge is $(p, >, q)$, where $(p +\!\!+ r, m, q) \in \gamma$ for some m and $r \neq \varepsilon$. Then as γ is safe for (s, s'): $s'\nabla q \leq s\nabla(p +\!\!+ r) = (s\nabla p)\nabla r < s\nabla p$, as required (as $r \neq \varepsilon$).

Now suppose that the edge is $(p, >, q +\!\!+ r)$, where $r \neq \varepsilon$ and $(p, m, q) \in \gamma$. Then by safety of γ, $s\nabla p \geq s'\nabla q > (s'\nabla q)\nabla r = s'\nabla(q +\!\!+ r)$, as required. □

4.2.2 Restricting Graph Bases

The dataflow graphs (which give, using the completion operator, proper size-change graphs) defined previously track dataflow at arbitrary environment paths. In particular, the graph bases of a graph γ annotating (say) a call $s \to s'$ are the full graph bases of s and s'.

While this gives the maximal possible precision, it is not possible to achieve this in general in the abstract interpretation. This is due to two reasons:

- The full graph basis of an abstract state (representing a set of states) will not in general be known
- To obtain an effectively computable termination test, we require an *a priori* finite set of size-change graphs.

Thus in general the size-change graphs produced in our static analysis will approximate the exact size-change graphs. The mechanism for achieving this in practice is described below.

k-Restricted Graphs

The set of graph bases of states is infinite, as graph bases may contain environment paths of arbitrary lengths. However, the set of environment paths of any fixed length is finite (as each element in an environment path is a function parameter appearing in the program). A natural way to restrict graph bases to an *a priori* finite set is therefore to limit environment paths to a bounded length.

The analyses that we shall describe in the remainder of this chapter will use this restriction: the abstract graph basis $\mathrm{gb}^{\alpha}(s)$ of an abstract state s will only contain environment paths of length at most k, where k is fixed in the analysis. The corresponding operation on size-change graphs is defined below:

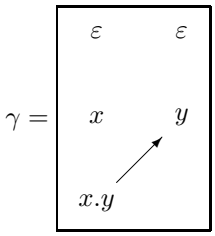
Definition 4.8. Let $\gamma = (A, \Gamma, B)$ be a size-change graph, and $k > 0$. Then the *k-restriction* of γ is the graph $[\gamma]_k := \gamma\restriction^{B'}_{A'}$, where $A' = \{p \in A \mid |p| \leq k\}$ and $B' = \{p \in B \mid |p| \leq k\}$.

It follows by Lemma 3.19 (safety of restriction) that $[\gamma]_k$ is safe whenever γ is: no safety is lost, though precision may be reduced.

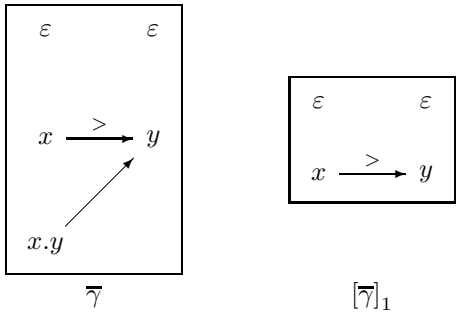
Restriction and Completion

We have described two operations on size-change graph: *completion* restores implied decreasing edges, while *restriction* loses information in order to guarantee a finite (computable) analysis. These operations interact in a nontrivial way: in order to preserve precision, we must ensure that completion is performed before restriction. We shall describe this with an example.

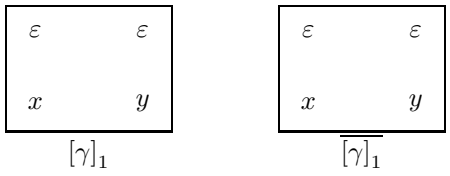
Consider the following size-change graph:



Then the graphs $\overline{\gamma}$ and $[\overline{\gamma}]_1$ are shown below:



However, if restriction is performed *before* completion, then information is lost, as shown below:



It is clear that this situation may lead to reduced precision of our termination analysis: in the first case we are given a decreasing edge, which may prove termination of a recursive function call, whereas in the second case no information is given. We shall thus ensure that the graphs produced in the abstract interpretation are complete at all stages without comment, avoiding this issue.

4.3 Static Analysis and Abstract Interpretation

4.3.1 Abstract States

Recall that we have formulated our analysis framework (in Section 3.4) as follows. We require a set *AbsState* of abstract states together with an order $\sqsubseteq$ on the set *AbsState*. The intended meaning of $\sqsubseteq$ is (conventionally as we are basing our framework on abstract interpretation): if $s \sqsubseteq s'$ then s is *more defined* than s'[1]. Finally, we require that the set *AbsState* be finite. We do not require any properties of the relation $\sqsubseteq$, other than the fact that it should be a partial order.

Furthermore, the *abstraction function* $\alpha : State \to AbsState$ maps a state to the *most defined* (least in the $\sqsubseteq$ order) abstract state that can represent it.

In the remainder of this chapter we shall present several static analyses, with different instantiations of the set *AbsState*. Each analysis defines approximations of the evaluation and call relations, $\Downarrow^\alpha$ and $\to^\alpha$ respectively. It is convenient to define these relations by a collection of inference rules, as in the definition of their counterparts in the exact semantics.

4.3.2 The Abstract Domain

The presentation outlined in the above may seem slightly nonstandard, and indeed differs in some ways from accounts of abstract interpretation. In particular, we have not required that our abstract domain form a lattice or assumed any specific properties of its order.

The difference lies in the fact that the $\Downarrow^\alpha$ and $\to^\alpha$ relations are in our case *nondeterministic* — for an abstract state s there may be several distinct abstract values v such that $s \Downarrow^\alpha v$. It is a common requirement that the abstract semantic relations be functions in abstract interpretation.

The two approaches may be reconciled in our case by taking not abstract states as the elements of our abstract domain, but *sets* of abstract states. It is this alternative viewpoint that we wish to develop briefly in this section.

Assume given the (finite) partially ordered set $(AbsState, \sqsubseteq)$. The intention is that our abstract domain should be the powerset of *AbsState*.

However, defining our abstract domain to be the powerset of the set *AbsState* will not suffice. For, the powerset operation interacts with the order $\sqsubseteq$ on *AbsState*, and this will cause difficulties. As a concrete example, suppose that $a \sqsubset b$ are abstract states, and let $B = \{b\}$ and $B' = \{a, b\}$. Then $B \subsetneq B'$, but B and B' in fact represent the *same* property, namely b. This problem can be solved using the construction of the *lattice of down-sets* of a set [DP02, p. 37]. Cognoscenti will recognise that this is just the *Hoare powerdomain*, used elsewhere in program analysis (for instance, strictness analysis for higher-order functions

[1] This is often counter-intuitive, but derives from the subset order on sets of states represented by abstract states. Unfortunately this conflicts with the $\supset$ symbol for implication used by Russell and Whitehead, which uses the reverse order.

[BHA86]). We briefly outline this construction in the remainder of this section, for reference, omitting proofs as these are straightforward.

Definition 4.9. Let $(X, \sqsubseteq)$ be a partially ordered set. A *down-set* in X is a subset S of X such that for all $x \in S$, if $x \sqsupseteq y \in X$ then $y \in S$. The *set of down-sets of* X is $\mathbb{P}^{\downarrow}(X) = \{S \subseteq X \mid (\forall x \in S)(\forall y \in X) y \sqsubseteq x \Rightarrow y \in S\}$.

The set of down-sets, ordered by inclusion, is indeed a lattice.

Lemma 4.10. *If $(X, \sqsubseteq)$ is a partially ordered set, then $(\mathbb{P}^{\downarrow}(X), \subseteq)$ is a lattice. The join and meet operations are given by union and intersection respectively.*

This lattice (necessarily complete when X is finite) is the domain of our abstract interpretation. It now merely remains to define the *abstraction* and *concretisation* functions in this framework.

Let us first define concretisation, given the abstraction function (for single concrete states) $\alpha : C \to X$.

Definition 4.11. Let C be a set, $(X, \sqsubseteq)$ be a partially ordered set, and let $\alpha : C \to X$. Define the concretisation map $\gamma : X \to \mathbb{P}(C)$ by:

$$\gamma(s) = \{x \in C \mid \alpha(x) \sqsubseteq s\}$$

This definition reflects the fact that $\alpha(x)$ is intended to be the *most defined* (least) element that represents x. We note a few properties:

Lemma 4.12. *With notation as in the previous definition,*

1. *γ is monotonic*
2. *For each $x \in C$, $x \in \gamma(\alpha(x))$*
3. *For each $a \in X$, if $t \in \gamma(a)$ then $\alpha(t) \sqsubseteq a$.*

Definition 4.13. Let C be a set, $(X, \sqsubseteq)$ be a partially ordered set, and let $\alpha : C \to X$. Given the map γ defined above, define maps $\alpha^{\mathbb{P}} : \mathbb{P}(C) \to \mathbb{P}^{\downarrow}(X)$ and $\gamma^{\mathbb{P}} : \mathbb{P}^{\downarrow}(X) \to \mathbb{P}(C)$ by:

$$\begin{aligned} \alpha^{\mathbb{P}}(S) &= \bigvee_{s \in S} \{x \in X \mid x \sqsubseteq \alpha(s)\} \\ \gamma^{\mathbb{P}}(A) &= \bigcup_{a \in A} \gamma(a) \end{aligned}$$

Lemma 4.14. *With $\alpha^{\mathbb{P}}$, $\gamma^{\mathbb{P}}$ as above, $\alpha^{\mathbb{P}}(\gamma^{\mathbb{P}}(A)) \subseteq A$ and $\gamma^{\mathbb{P}}(\alpha^{\mathbb{P}}(S)) \supseteq S$ for all A, S. Furthermore $\alpha^{\mathbb{P}}$ and $\gamma^{\mathbb{P}}$ are monotonic.*

In other words, we have defined a Galois connection between $\mathbb{P}(C)$ and $\mathbb{P}^{\downarrow}(X)$. This is similar to the definition of a Galois connection from an *extraction function* [NNH99, p.235], with the difference that we assume the set X is itself given a partial order $\sqsubseteq$.

Finally, let us return to our definition of the $\Downarrow^\alpha$ relation as a *nondeterministic* relation of type $AbsState \times AbsState$. It is clear that by going to the powerset, we may now represent this by a function. More precisely, the equivalent deterministic evaluation relation is given by:

$$S \Downarrow^\alpha T \text{ where } T = \{z \mid (\exists x \in S)(\exists y) x \Downarrow^\alpha y \sqsupseteq z\}$$

We have therefore shown that our approach is a special case of the general framework of abstract interpretation, and shall be free to make use of the simpler approach in the sequel.

We shall finally remark that an equivalent definition of our abstract domain can be given by considering *maximal* elements, rather than downwards-closed sets. This alternative approach may be more intuitive, and corresponds more closely to an implementation (where we are more likely to represent and store maximal elements rather than downwards-closed sets). However, we have eschewed this in favour of the down-set approach as the latter is notationally simpler.

4.3.3 Galois Insertions

We have thus far shown that our approach gives rise to a *Galois connection*, as is required of the abstraction and concretisation functions in abstract interpretation. This ensures that no safety is lost by working on the abstract domain.

It is possible to specify a stronger condition, namely that the connection between the concrete and abstract domains should be a *Galois insertion*:

Definition 4.15. Let $(A, \sqsubseteq)$ and $(B, \leq)$ be (complete) lattices, and suppose that $\alpha : A \to B$ and $\gamma : B \to A$ are monotonic maps. Then $A \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} B$ is a *Galois insertion* if:

1. If $a \in A$, then $\gamma(\alpha(a)) \sqsupseteq a$, and
2. If $b \in B$, then $\alpha(\gamma(b)) = b$

The difference with Galois connections is that we require that no precision is lost by performing concretisation followed by abstraction. This occurs precisely when the abstract domain contains no redundant elements.

Let us return to our setting, where $\alpha : C \to X$ is given, and the abstract domain is $\mathbb{P}^\downarrow(X)$. It is easy to characterise those abstraction maps α which give rise to Galois insertions:

Lemma 4.16. *Let $\alpha : C \to X$ and $\alpha^\mathbb{P}$, $\gamma^\mathbb{P}$ be defined as before. Then $\langle \alpha^\mathbb{P}, \gamma^\mathbb{P} \rangle$ is a Galois insertion between $\mathbb{P}(C)$ and $\mathbb{P}^\downarrow(C)$ iff α is surjective.*

From this result we may conclude that it is natural in our setting to require α to be surjective, so that the resulting abstraction and concretisation functions form a Galois insertion. For, suppose that this is not the case, and let x be an element not in the image of α. Let $A = \{t \mid t \sqsubseteq x\}$, and $B = \{t \mid t \sqsubset x\}$. These are trivially both members of $\mathbb{P}^\downarrow(X)$.

Furthermore,

$$\gamma^{\mathbb{P}}(A) = \bigcup_{a \sqsubseteq x} \gamma(a) = \gamma(x) \cup \bigcup_{a \sqsubset x} \gamma(a) = \gamma^{\mathbb{P}}(B) \cup \{y \in C \mid \alpha(y) \sqsubseteq x\}$$

But for each $y \in C$ such that $\alpha(y) \sqsubseteq x$, $\alpha(y) \sqsubset x$ (as x is not in the image of α), so $\alpha(y) \in B$, and thus $y \in \gamma^{\mathbb{P}}(B)$. Hence $\gamma^{\mathbb{P}}(A) = \gamma^{\mathbb{P}}(B)$.

In the case where α is not surjective, we have therefore exhibited *two* abstract representations for the *same* set of states, which does not increase precision or expressiveness and is wasteful. In practice however we shall find a use for abstractions that do not give rise to Galois insertions and shall note this whenever applicable.

4.4 0CFA

We shall now describe the first of our static flow analyses, namely 0CFA. This analysis is originally due to Olin Shivers [Shi91, Shi88], and is a standard technique for analysing functional programs. We shall define 0CFA, formulated in our setting, and its application to termination analysis in this section. While 0CFA is a well-known analysis, we believe that the precise description of its application to our SCT framework serves to illuminate many of the ideas in subsequent developments. Furthermore, we improve on Jones and Bohr's application of 0CFA[2] by allowing free variables in programs.

4.4.1 Motivation

The Abstract Domain

We recall that the choice of a finite set of abstract states is governed by two requirements: map all constants (possibly infinitely many) to a finite set, and map all environment trees (of unbounded length) to a finite set of environment representations.

The idea of 0CFA is to take the *simplest* possible abstraction of environments: no information at all is kept[3]. 0CFA is thus a minimal analysis, in the sense that any two states with the same program point are identified in the abstract domain. As a result, 0CFA is a *context-insensitive* analysis: the body of a function is represented once in the call graph, even if it is called in many different contexts.

We shall first describe the elements of the abstract domain that each concrete state abstracts to, as this is straightforward. We will later complete the picture, adding in other abstract values and states. The abstraction of all concrete states (apart from primitive

[2] Jones and Bohr [JB04] do not refer to their control-flow analysis as 0CFA, but it is equivalent.

[3] Implicitly, a *global* environment is kept, mapping each variable to a set of possible values. This global environment is often made explicit in presentations of 0CFA [NNH99], but is equivalent to our later use of *closure analysis*.

constants, discussed earlier) is simple: we simply take the label of the root node of the tree describing the state.

Any state $e : \rho$ maps to the abstract state e (no information about environments is kept). However, in order to distinguish two occurences of the same expression in the program text, the *program point* (unique label) of e shall be included in its abstract representation implicitly.

Let us now consider *closures* of the form $(\!|f : \rho|\!)$ where f is a function label. The abstract state corresponding to such a state might be defined as just f, by analogy with the case of expression states. However, this turns out to be too imprecise even for this simple analysis.

For, the *domain* of the environment of a closure state $(\!|f : \rho|\!)$ is significant, and in particular allows partially applied functions to be distinguished from totally applied functions. As an example, the state $(\!|map : \emptyset|\!)$ denotes the *map* function (a function of two arguments), while the state $(\!|map : \{f \mapsto v\}|\!)$ (for some value v) denotes the result of a partial application of *map*. Finally, the state $(\!|map : \{f \mapsto v, xs \mapsto w\}|\!)$ denotes a complete application of *map*, and is qualitatively different from the previous two states, as this is not a value.

The loss of precision incurred from identifying the *map* function and one of its partially or totally applied occurrences is very substantial. Correspondingly, we shall define the abstraction of a closure state to include the *domain* of the environment

$$\alpha((\!|f : \rho|\!)) = (\!|f : \operatorname{dom} \rho|\!)$$

The range of α is finite, as for each function there are only finitely many domains of valid environments. This handles the precision issue raised above. It is worth noting that this issue arises because our language allows definitions of (curried) functions with several arguments, whereas a language relying on nested λ-abstractions for functions of several arguments does not present this difficulty.

To complete our overview of the abstract domain, we must deal with constants, both primitive and algebraic. As stated previously, the abstraction of primitive constants is of little import, and we shall not consider it further here. Finally, an algebraic constant state of the form $\mathsf{C}(v_1, \ldots, v_n)$ maps to the abstract value C.

Closure Analysis

In the abstract domain of 0CFA, no information about environments is kept. This information must somehow be recovered (approximately) in the abstract interpretation, however. This is achieved by *closure analysis*, which we outline below.

Consider a higher-order function, say *map* in keeping with our previous examples. Any (fully applied) occurence of *map* in the program will be represented by the abstract state $s = (\!|map : \{f, xs\}|\!)$. Now suppose that t is a concrete state represented by s. Then t is an instance of *map*, so t applies its argument f, leading to a call $t \to t'$ for some state t'. By definition of safety, there must be a corresponding call $s \to^{\alpha} s'$, where s' represents t'.

The issue is that the abstract representation of s does not give any information about

the parameter f (indeed, as this is a context-insensitive analysis it cannot). *Closure analysis* allows the set of functions that could be bound to function parameters to be approximated. Thus closure analysis can deduce that the f parameter of *map* could be bound to states represented by abstract states $s_1, \ldots, s_n$ and no others, allowing the required call to be approximated in the above example.

Closure analysis is intrinsically a *whole-program* analysis — the set of functions that f could represent is of course dependent on the uses of *map* in the entire program.

To compute an approximation to the set of possible values of the x parameter of a function h, we focus on *binding sites* for x. That is, we observe that the only semantic rule that may bind x in an environment is the function application rule, where

1. $e_1 e_2 : \rho$ is an application state,
2. $e_1 : \rho \Downarrow (\!|h : \rho|\!)$, where the first parameter of h not bound in ρ is x, and
3. $e_2 : \rho \Downarrow v$ for some value v.

In the above, v is bound to x in the callee state. Conversely, x can only be bound in the above circumstances. The problem therefore reduces to finding all possible instances of this rule[4].

We may now observe that in the above the application $e_1 e_2$ occurs as a subexpression of the program. For, all states and values that occur in the semantics are labelled by subexpressions of the program (or constants). Furthermore, the evaluations (in the exact semantics) are naturally approximated by corresponding evaluations in the exact semantics. As a result, we shall shortly see that the following approximation is valid:

> The set of values of a parameter x of function h is approximated by the set of values v such that: $e_1 e_2$ is a subexpression of the program, $e_1 \Downarrow^\alpha (\!|h : \rho|\!)$ where the first parameter of h not bound in ρ is x, and $e_2 \Downarrow^\alpha v$.

The above formulation is clearly effectively computable (as the set of subexpressions of the program is finite). Various improvements on this procedure are possible for efficiency and precision, in particular restricting the expression $e_1 e_2$ to be reachable (*i.e.* not dead code).

A Refinement: Arbitrary Values

Consider the following (rather trivial) program:

```
head xs = match xs with y :: ys → y

head <X>
```

where X is a free (template) variable. Certainly this program terminates whenever X is mapped to a list, and we should be able to detect this. However, there are *no* applications

[4]We have tacitly assumed that h is the only function with parameter x, without loss of generality

of the cons constructor in this program, and thus closure analysis can find no approximation to the head and tail of a list, making analysis of this program seemingly impossible.

The apparent contradiction in the above derives from the fact that closure analysis relies on the entire program text being known to find application sites. In effect, when assigning values such as lists to free variables, we are no longer considering a whole program, and as such closure analysis is not sufficient. This problem does not arise when assigning primitive constant values to arbitrary constants, as these cannot contain functions.

To remedy this, we shall introduce a new kind of abstract value, to describe values that are not constructed within the program. This will be used to describe the values that may be assigned to program template variables (such as X above). We shall only be concerned with two kinds of values for such variables:

- Primitive constants
- Algebraic values such as lists (or values of user-defined datatypes)

In the first case, we may map a template variable to an abstract constant. However, in the second case we must deal with the fact that an arbitrary list (say) may not be described accurately by closure analysis.

To this end, we introduce a single abstract value, denoted $*$. This stands for *any* value and therefore safely approximates lists or any other data structure. Whenever the value $*$ is used in the program (for instance, deconstructed in pattern matching), it is assumed to stand for any value with subvalues described by $*$. For instance, matching $*$ against the pattern $x :: xs$ succeeds, with bindings $x \mapsto *$ and $xs \mapsto *$. This is a safe, if coarse, approximation to lists (or indeed any other data structure).

It may appear that the choice of a single representation $*$ for arbitrary values is too imprecise. In fact, we argue that there would be no substantial gain of precision from introducing (say) a separate value for each type. For, the outcome of matching a value described by $*$ against a list pattern is either a type error or pattern matching error (immediate abortion), or success. The only information gained by using a special value to describe lists is that the match operation cannot result in a type error. However, as type errors do not introduce nontermination, no precision is lost for termination analysis.

Note also that the $*$ token describes all values, including lists whose elements may be themselves functions.

4.4.2 Static Analysis

We will now describe the 0CFA analysis in more detail. We first give a precise account of the abstract domain, then proceed to a full definition of the analysis.

The Abstract Domain

We shall now describe the set *AbsState* of abstract representations of states, together with the information order $\sqsubseteq$ on this set. The results of Section 4.3 show that this can be extended to a complete lattice whose elements are sets of abstract states.

Abstract States The abstract domain comprises the following descriptions of states:

1. The set *AbstConst* of primitive constant representations
2. *Expression* states of the form e^l, where e is an expression occurring at program point l
3. *Closure* states of the form $(\!|f : S|\!)$, where f is a function, and $S \subseteq \mathit{Params}(f)$
4. *Algebraic Constant* representations: each constructor C defined in the program is an abstract state
5. The special value $*$, representing unknown values (which may or may not be constructible within the program).

The abstraction map $\alpha : \mathit{State} \to \mathit{AbsState}$ is defined as follows:

$$\begin{aligned} \alpha(c) &= \alpha^C(c) \\ \alpha(e^l : \rho) &= e^l \\ \alpha((\!|f : \rho|\!)) &= (\!|f : \operatorname{dom}\rho|\!) \\ \alpha(\mathsf{C}(v_1, \ldots, v_n)) &= \mathsf{C} \end{aligned}$$

Abstract states of the form c, $(\!|f : S|\!)$ (where $S \subsetneq \mathit{Params}(f)$) and C unambiguously denote *values*, while states of the form $(\!|f : S|\!)$ (where $S = \mathit{Params}(f)$) and e^l unambiguously denote states which are not values, *i.e.* are not fully evaluated. We write AbsVal for the set of abstract states denoting values (excluding $*$).

The information ordering on the domain of abstract states is straightforward: abstract primitive constants are compared with the $\sqsubseteq^C$ order on *AbstConst*, and any other states are incomparable, with the exception of $*$. We have:

$$* \sqsupseteq v \text{ for all } v \in \mathit{AbsVal}$$

For, the representation $*$ denotes *any* proper value (though not states requiring further evaluation), which generalises any $v \in \mathit{AbsVal}$.

Finally, we note that for any $v \in \mathit{AbsVal}$, $\gamma(v) \subseteq \mathit{Value}$. Likewise, whenever $s \notin \mathit{AbsVal}$, $\gamma(s) \cap \mathit{Value} = \emptyset$. Finally, $\gamma(*) = \mathit{Value}$.

Graph Bases Each abstract state $s \in \mathit{AbsState}$ is given a *graph basis* $\mathrm{gb}^\alpha(s)$. The safety condition on these is that $\mathrm{gb}^\alpha(s)$ should be a safe approximation (subset) of $\mathrm{gb}(t)$ for any $t \in \gamma(s)$. The cases for s and $t \in \gamma(s)$ are as follows:

1. s is a primitive constant c. Then $t = c'$ for some c', so $\mathrm{gb}(t) = \{\varepsilon\}$. We therefore take $\mathrm{gb}^\alpha(s) = \{\varepsilon\}$.

2. s is an expression state e^l. Let $S = \mathrm{Scope}(l)$. Now $t = e : \rho$ where $\mathrm{dom}\,\rho = S$. In particular, $\langle x \rangle \in \mathrm{gb}(t)$ for all $x \in S$, so it is safe to take $\mathrm{gb}^\alpha(s) = \{\varepsilon\} \cup \{\langle x \rangle \mid x \in S\}$. No extension of this set is safe in general.

3. s is a closure state $(\!|f : S|\!)$. As before we can take $\mathrm{gb}^\alpha(s) = \{\varepsilon\} \cup \{\langle x \rangle \mid x \in S\}$.

4. s is an algebraic value state C. Let $n = \sharp\mathsf{C}$. Then $t = \mathsf{C}(v_1, \ldots, v_n)$, so as before it is safe to take $\mathrm{gb}^\alpha(s) = \{\varepsilon\} \cup \{\langle x_i \rangle\}_{i=1}^n$.

5. s is the special value $*$. Then t could be any value, and thus the only safe approximation is $\mathrm{gb}^\alpha(s) = \{\varepsilon\}$.

With the exception of the special value $*$, the graph basis $\mathrm{gb}^\alpha(s)$ is just the *1-limited graph basis*: the set of environment paths of length at most 1 in any concrete state that s could represent.

Reachable States and Galois Insertions As we have informally observed previously, a state of the form e^l (say) is intended to represent *reachable* states of the form $e : \rho$. In particular, this only represents states which can be approximated *via* closure analysis. As a consequence it was necessary to introduce the special case $*$ for arbitrary values assigned to free program variables.

However, this distinction is *not* represented by the abstraction and concretisation functions. For instance, $\gamma(e^l)$ is the set of *all* states (trees) whose root node is e^l. In general this will include unreachable states. This appears to be unavoidable, as closure analysis is defined in terms of the abstract interpretation — introducing this notion in the abstract domain would be dangerously circular.

A somewhat surprising consequence of this is that the abstraction that we have defined does *not* give rise to a Galois insertion. This follows by Lemma 4.16 from the fact that the abstraction function α is not surjective. Indeed, $*$ is not in the range of α.

The cause of this is that for any concrete value v, $\alpha(v)$ should be the *most* defined abstract value representing v. The special token $*$ is never the most defined such representation.

As a result, there appears to be wasteful duplicate representations in the abstract domain (lifted to powersets). For, $\gamma^{\mathbb{P}}(\mathit{AbsVal}) = \gamma^{\mathbb{P}}(\mathit{AbsVal} \cup \{*\}) = \mathit{Value}$. The difference between these two sets of values is that the first denotes only those values that may be constructed *within* the program, whereas the second denotes *any* value, even if cannot be produced in

the program. That is, the value $*$ acts as a marker that we are not restricted to constructible values.

The fact that there are two different abstract representations, with the *same* concretisation, is an inconvenience that we shall have to deal with in the sequel.

Abstract Interpretation

All the elements of the 0CFA analysis have now been defined. In particular, we have introduced the abstract domain, the use of closure analysis and the specific treatment of algebraic constants. The full analysis is now given in Figures 4.2 and 4.3. This defines the abstract evaluation relation $\Downarrow^\alpha$ and the abstract call relation $\rightarrow^\alpha$. We omit error productions from the description of the static analysis for concision, as these are straightforward.

Furthermore, each judgement is annotated to produce a size-change graph. The size-change graphs annotating the given rules are simplified for clarity, as we elide the graph basis information that is needed to construct the graph. This can be deduced by inspection of the states appearing in the semantic judgements.

The size-change graphs constructed in the analysis of Figure 4.2 are for the most part directly derived from the graphs in the annotated exact semantics of Figure 3.2. The only exception is that we must define the effects of primitive operators. We have defined the $\mathit{SizeChange}_{\mathrm{op}}$ function to capture the effect of each primitive operator abstractly. Given an expression $\mathit{op}(e_1, \ldots, e_n)$ it thus suffices to evaluate each e_i to an abstract constant c_i, producing a graph γ_i. The γ_i relate the values of the e_i to any variables in scope; and thus the relationship between the result of evaluating $\mathit{op}(e_1, \ldots, e_n)$ and variables in scope is given by:

$$\mathit{OpGraph}\ \mathit{op}\ \langle (c_1, \gamma_1), \ldots, (c_n, \gamma_n) \rangle := (\bigcup_{i=1}^{n} \gamma_i[x_i/\varepsilon]); \mathit{SizeChange}_{\mathrm{op}}\ \mathit{op}\ \langle c_1, \ldots, c_n \rangle$$

4.4.3 Soundness

Let us now prove that the 0CFA size-change termination analysis is sound. By Theorem 3.39, it suffices to prove control-safety and dataflow-safety of the 0CFA call graph. The control-safety property requires us to show that for any edge in the dynamic call graph $s \rightarrow s'$ (or $s \Downarrow v$), if $t \sqsupseteq \alpha(s)$ then there exists $t' \sqsupseteq \alpha(s')$ such that $t \rightarrow^\alpha t'$.

The proof of soundness of 0CFA requires a precise characterisation of reachable program states, to show that closure analysis safely approximates the values of function parameters. We therefore first define this property, then proceed to show that control-safety of the call graph follows.

The reader should note that this section gives the 0CFA soundness proof in full detail for reference, and as it is used later to prove soundness of k-CFA, but that beyond the structure

Values, Function Names and Constants

$$\frac{}{v \Downarrow v, 1}\ {\scriptstyle v \in AbsVal} \qquad \frac{}{f \Downarrow (\!|f : \emptyset|\!), 0} \qquad \frac{}{c \Downarrow \alpha(c), 0}$$

Primitive Operators

$$\frac{(\forall i) e_i \Downarrow c_i, \gamma_i \quad \mathit{Apply}^{\alpha}\ \mathit{op}\ \langle c_1, \ldots, c_n \rangle = c}{\mathit{op}(e_1, \ldots, e_n) \Downarrow c, \mathit{OpGraph}\ \mathit{op}\ \langle (c_1, \gamma_1), \ldots, (c_n, \gamma_n) \rangle}$$

Evaluating Closures

$$\frac{\mathit{Body}(f) \Downarrow v, \gamma}{(\!|f : S|\!) \Downarrow v, \gamma^*}\ {\scriptstyle S = Params(f)}$$

Constructors and Pattern Matching

$$\frac{e_1 \Downarrow v_1, \gamma_1 \quad \cdots \quad e_n \Downarrow v_n, \gamma_n}{\mathsf{C}(e_1, \ldots, e_n) \Downarrow \mathsf{C}, [\mathrm{constr}(\gamma_1, \ldots, \gamma_n)]_1}$$

$$\frac{e \Downarrow c, \gamma \quad e_l \Downarrow v, \delta}{s = \mathbf{match}\ e\ \mathbf{with}\ \left\langle \mathsf{C}_i(x_1^i, \ldots, x_{n_i}^i) \rightarrow e_i \right\rangle_{i=1}^{k} \Downarrow v, \mathrm{patmatch}_s^l(\gamma); \delta}\ {\scriptstyle c \sqsupseteq \mathsf{C}_l}$$

Conditionals

$$\frac{e_g \Downarrow c \sqsupseteq \alpha(\mathbf{true}), \gamma_g \quad e_t \Downarrow v, \gamma}{\mathbf{if}\ e_g\ \mathbf{then}\ e_t\ \mathbf{else}\ e_f \Downarrow v, \gamma^*} \qquad \frac{e_g \Downarrow c \sqsupseteq \alpha(\mathbf{false}), \gamma_g \quad e_f \Downarrow v, \gamma}{\mathbf{if}\ e_g\ \mathbf{then}\ e_t\ \mathbf{else}\ e_f \Downarrow v, \gamma^*}$$

Function Application

$$\frac{e_1 \Downarrow (\!|f : S|\!), \gamma_1 \quad e_2 \Downarrow w, \gamma_2 \quad \mathit{Bind}\ (\!|f : S|\!)\ w \Downarrow v, \delta}{e_1 e_2 \Downarrow v, [\mathrm{app}_x(\gamma_1, \gamma_2)]_1 ; \delta}\ {\scriptstyle S \subsetneq Params(f) \wedge x = \mathrm{hd}(Params(f) \setminus S)}$$

Closure Analysis

$$\frac{(\exists e_1 e_2 \in P) \quad e_1 \Downarrow (\!|f : S|\!), \gamma \quad e_2 \Downarrow v, \delta}{x \Downarrow v, \overline{1[\varepsilon/x]}}\ {\scriptstyle S \subsetneq Params(f) \wedge x = \mathrm{hd}(Params(f) \setminus S)}$$

$$\frac{(\exists \mathsf{C}_j(e_1, \ldots, e_n) \in P) \quad e \Downarrow \mathsf{C}_j, \gamma \quad e_j \Downarrow v, \delta}{x_i^j \Downarrow v, \overline{1[\varepsilon/x_i^j]}}\ {\scriptstyle \mathrm{Binder}(x_i^j) = \mathbf{match}\ e\ \mathbf{with}\ \left\langle \mathsf{C}_i(x_1^i, \ldots, x_{n_i}^i) \rightarrow e_i \right\rangle_{i=1}^{k}}$$

$$\frac{(\exists e_1 e_2 \in P) \quad e_1 \Downarrow *, \gamma}{x \Downarrow *, 1[\varepsilon/x]}\ {\scriptstyle x \text{ is a function parameter}}$$

$$\frac{e \Downarrow *, \gamma}{x \Downarrow *, 1[\varepsilon/x]}\ {\scriptstyle \mathrm{Binder}(x) = \mathbf{match}\ e\ \mathbf{with}\ \left\langle \mathsf{C}_i(x_1^i, \ldots, x_{n_i}^i) \rightarrow e_i \right\rangle_{i=1}^{k}}$$

Program Template Variables

$$\frac{}{X \Downarrow \mathfrak{W}(X), 0}\ {\scriptstyle \text{with valuation } \mathfrak{W}}$$

Figure 4.2: 0CFA: Static Analysis (Evaluation)

Primitive Operators

$$\frac{(\forall j < i) \quad e_j \Downarrow v_j}{op(e_1, \ldots, e_n) \to e_i, 1^*}\ {\scriptstyle i \leq n}$$

Evaluating Closures

$$\frac{}{(\!|f : S|\!) \to Body(f), 1^*}\ {\scriptstyle S = Params(f)}$$

Constructors and Pattern Matching

$$\frac{(\forall j < i) \quad e_j \Downarrow v_j}{\mathsf{C}(e_1, \ldots, e_n) \to e_i, 1^*}\ {\scriptstyle i \leq n}$$

$$\frac{\begin{array}{c}\mathbf{match}\ e\ \mathbf{with}\ \left\langle \mathsf{C}_i(x_1^i, \ldots, x_{n_i}^i) \to e_i \right\rangle_{i=1}^{k} \to e, 1^* \\ e \Downarrow c, \gamma\end{array}}{s = \mathbf{match}\ e\ \mathbf{with}\ \left\langle \mathsf{C}_i(x_1^i, \ldots, x_{n_i}^i) \to e_i \right\rangle_{i=1}^{k} \to e_l, \mathrm{patmatch}_s^l(\gamma)}\ {\scriptstyle c \sqsupseteq \mathsf{C}_l}$$

Conditionals

$$\frac{\begin{array}{c}\mathbf{if}\ e_g\ \mathbf{then}\ e_t\ \mathbf{else}\ e_f \to e_g, 1^* \\ e_g \Downarrow c \sqsupseteq \alpha(\mathbf{true})\end{array}}{\mathbf{if}\ e_g\ \mathbf{then}\ e_t\ \mathbf{else}\ e_f \to e_t, 1^*} \qquad \frac{e_g \Downarrow c \sqsupseteq \alpha(\mathbf{false})}{\mathbf{if}\ e_g\ \mathbf{then}\ e_t\ \mathbf{else}\ e_f \to e_f, 1^*}$$

Function Application

$$\frac{}{e_1 e_2 \to e_1, 1^*} \qquad \frac{e_1 \Downarrow (\!|f : S|\!), \gamma_1}{e_1 e_2 \to e_2, 1^*}\ {\scriptstyle S \subsetneq Params(f)}$$

$$\frac{e_1 \Downarrow (\!|f : S|\!), \gamma_1 \quad e_2 \Downarrow w, \gamma_2}{e_1 e_2 \to Bind\ (\!|f : S|\!)\ w, [\mathrm{app}_x(\gamma_1, \gamma_2)]_1}\ {\scriptstyle \substack{S \subsetneq Params(f) \wedge \\ x = \mathrm{hd}(Params(f) \setminus S)}}$$

Figure 4.3: 0CFA: Static Analysis (Calls)

of the argument much of the detail may be omitted on first reading. We shall not go into such detail for the k-CFA or automata analyses.

Valuations

Recall that we use free variables (say $X_1, \ldots, X_n$) to prove termination of programs given arbitrary inputs to certain functions. We have defined (Definition 3.33, p. 46) control-safety of the static call graph in terms of an *abstract valuation* $\mathfrak{W}$, which approximates the valuation $\mathfrak{V}$ assigning values to free variables.

Throughout this section we assume that $\mathfrak{W}$ and $\mathfrak{V}$ are fixed (but arbitrary), and that $\mathfrak{W}$ is a safe approximation of $\mathfrak{V}$, in the sense that for each free variable X, $\mathfrak{V}(X) \in \gamma(\mathfrak{W}(X))$.

Furthermore, as we have previously noted, such arbitrary assignments of values to free variables create difficulties for 0CFA static analysis. For, closure analysis relies on having the whole program, whereas free variables allow arbitrary (possibly non-constructible) values to be introduced.

As a result, we shall restrict the range of $\mathfrak{W}$ in the following way:

Property 4.17. *For each free variable X appearing in the program, $\mathfrak{W}(X) \in AbstConst$ or $\mathfrak{W}(X) = *$.*

That is, two kinds of values are allowed to appear: either (first-order) constants, which have no bearing on closure analysis; or the special value $*$, indicating that the corresponding concrete value need not be valid under closure analysis.

Reachability and Closure Analysis

Before we can give the proof, it is crucial to remark the following. The dynamic call graph of a program P only contains, by definition, those edges $s \to s'$ where s is a *reachable* state. That is, let **start** be the start state of the program (this is of the form $\mathbf{start} = e : \emptyset$, where e is the main expression of the program). Then we are only concerned with states s such that $\mathbf{start} \to^* s$.

In particular, it is *not* necessarily the case that for *any* state s such that $s \Downarrow v$, there is a corresponding evaluation in the abstract interpretation. Due to our use of closure analysis, the approximation is only correct for reachable s.

We shall therefore define a *reachability* condition $\mathrm{Valid}_{\mathrm{CA}}(s)$ on states. This encodes all the properties of reachable states s that are required in the proof. It will of course be necessary to show that all reachable states do satisfy $\mathrm{Valid}_{\mathrm{CA}}(s)$.

Closure Analysis and Template Variables Closure analysis approximates, for each variable, the set of values which may be bound to this variable within the program text. Whenever a value v is bound to a variable x in a reachable state, we therefore have two cases:

1. *Either* v is bound to x within the program, in which case v can be approximated by considering possible binding sites,

2. *Or* v is bound to x within a value assigned to a program template variable. In this case, the special abstract value $*$ identifies that v cannot be approximated in the same way.

Definition 4.18. Let f be a function and x a parameter of f. We define $\text{values}(f, x) = \{v \mid (\exists e_1 e_2 \in P) e_1 \Downarrow^\alpha (\!|f : S|\!) \wedge e_2 \Downarrow^\alpha v\}$, where S is the set of parameters of f that appear before x, if $(\nexists e_1 e_2 \in P) e_1 \Downarrow^\alpha *$; and $\text{values}(f, x) = \mathit{AbsVal} \cup \{*\}$ otherwise.

In the above definition, we single out the case of a program in which a value approximated by $*$ occurs applied. In this case, as $*$ could represent *any* closure, not necessarily constructible within the program, we cannot assert anything about the value of function parameters[5].

Definition 4.19. Let C be a constructor, and $i \leq \sharp\mathsf{C}$. Then $v \in \text{values}(\mathsf{C}, i)$ iff for some constructor application $\mathsf{C}(e_1, \ldots, e_n)$ in P, $e_i \Downarrow^\alpha v$.

The sets $\text{values}(f, x)$ and $\text{values}(\mathsf{C}, i)$ encode the results of closure analysis for constructed values and function parameters. We may now use the assumption that each variable only occurs bound in one place in the program to define the following shorthand (and extend the definition to variables defined in pattern matching constructs):

Definition 4.20. Let x be a variable. Define $\text{values}(x)$ to be:

1. If $\text{Binder}(x)$ is the definition of function f, then $\text{values}(x) = \text{values}(f, x)$.

2. If $\text{Binder}(x) = \textbf{match}\ e\ \textbf{with}\ \left\langle \mathsf{C}_i(x_1^i, \ldots, x_{n_i}^i) \to e_i \right\rangle_{i=1}^{k}$, and $x = x_j^i$, then we have two cases:

   - If $e \Downarrow^\alpha *$, then $\text{values}(x) = \mathit{AbsVal} \cup \{*\}$.
   - Otherwise, $\text{values}(x) = \text{values}(\mathsf{C}_i, j)$.

We may now define the notion of validity for reachable states. This condition states that all values bound to variables should be approximated by the corresponding sets $\text{values}(x)$. However, whenever $* \in \text{values}(x)$, the value bound to x need not be constructible within the program, which we must take into consideration.

The resulting definition of reachable states is as follows:

Definition 4.21. Let s be a state. We say that s is *closure-analysis valid*, and write $\text{Valid}_{\text{CA}}(s)$, if one of the following holds:

- s is a constant, or

[5]The call graph of a program with $*$ in operator position is thus dramatically imprecise, but such a program is unlikely to be terminating as any function may be substituted for $*$.

- $s = \mathsf{C}(v_1, \ldots, v_n)$ and for each i, $v_i \in \gamma^{\mathbb{P}}(\text{values}(\mathsf{C}, i))$. Furthermore, for each i, if $* \notin \text{values}(\mathsf{C}, i)$, then $\text{Valid}_{\text{CA}}(v_i)$ holds.
- s is an expression or closure state with environment ρ, and for each $x \in \text{dom}\,\rho$, $\rho(x) \in \gamma^{\mathbb{P}}(\text{values}(x))$, and if $* \notin \text{values}(x)$, then $\text{Valid}_{\text{CA}}(\rho(x))$.

We are now in a position to prove control-safety of 0CFA. The proof is shown in full detail for reference, but is straightforward in nature given the appropriate definition of the $\text{Valid}_{\text{CA}}(s)$ predicate. Similar proofs will not be given in full in the remainder of the text.

Lemma 4.22. *Let s be a state such that $Valid_{CA}(s)$. Suppose that $s \Downarrow v$. Then*

- *Either $\alpha(s) \Downarrow^{\alpha} *$, or*
- *$Valid_{CA}(v)$ holds, and whenever $t \sqsupseteq \alpha(s)$, there exists $w \sqsupseteq \alpha(v)$ such that $t \Downarrow^{\alpha} w$.*

Proof. We proceed by induction on the proof of $s \Downarrow v$, and cases on the last rule used in the proof. Let us first note a number of facts that simplify the proof:

1. If s is a state with an empty environment, then $\text{Valid}_{\text{CA}}(s)$.
2. If s is a state, and s' has the same environment as s, then $\text{Valid}_{\text{CA}}(s) \Rightarrow \text{Valid}_{\text{CA}}(s')$.
3. Let s be a state, and $t \sqsupseteq \alpha(s)$, $t \neq *$. If $s = e : \rho$ then $t = e$; if $s = (\!|f : \rho|\!)$ then $t = (\!|f : \text{dom}\,\rho|\!)$; and if $s = \mathsf{C}(v_1, \ldots, v_n)$, then $t = \mathsf{C}$.

We now consider the cases for the last rule used. We only prove $\text{Valid}_{\text{CA}}(v)$ in nontrivial cases (using the above remarks), and assume $t \neq *$ as this case is trivial.

Values If the value rule is used, then $s = v \Downarrow v$, and $s \in \mathit{Value}$. Hence as $t \sqsupseteq \alpha(s)$, $t \in \mathit{AbsVal}$ or $t = *$, so $t \Downarrow^{\alpha} t$, as required.

Function References Let $s = f : \rho$, where f is a function name. Then $s \Downarrow \mathit{Body}(f) : \emptyset = v$. Certainly $\text{Valid}_{\text{CA}}(v)$, and $t = f \Downarrow^{\alpha} \mathit{Body}(f) = \alpha(v)$, as required.

Constants Let $s = c : \rho$ be a reference to a constant. Then $s \Downarrow c$. Now $t = \alpha(s) = c$ (the syntactic constant expression c), and so $t \Downarrow^{\alpha} \alpha(c)$, as required.

Closures Suppose that s is a function closure $s = (\!|f : \rho|\!)$, where $S := \text{dom}\,\rho = \mathit{Params}(f)$. Then $s \Downarrow v$ whenever $s' = \mathit{Body}(f) : \rho \Downarrow v$. As $\text{Valid}_{\text{CA}}(s) \Rightarrow \text{Valid}_{\text{CA}}(s')$ we may apply the inductive hypothesis. Let $t' = \mathit{Body}(f)$. If $t' \Downarrow^{\alpha} *$ then $t = (\!|f : S|\!) \Downarrow^{\alpha} *$ and we are done. Otherwise, by the inductive hypothesis $t' \Downarrow^{\alpha} w \sqsupseteq \alpha(v)$, and $\text{Valid}_{\text{CA}}(v)$ holds. Thus $t \Downarrow^{\alpha} w$, as required.

Primitive Operators Let $s = op(e_1, \ldots, e_n) : \rho$. For each i let $s_i = e_i : \rho$ (this is closure-analysis valid). Now $s_i \Downarrow c_i$ for each i, so by the inductive hypothesis $t_i := e_i \Downarrow^\alpha d_i \sqsupseteq \alpha(c_i)$ for each i. Furthermore $s \Downarrow \mathit{Apply}\ op\ \langle c_1, \ldots, c_n\rangle = c$. Finally, $t = op(e_1, \ldots, e_n) \Downarrow \alpha\mathit{Apply}^\alpha\ op\ \langle d_1, \ldots, d_n\rangle = d$. By safety of Apply^α it follows that (provided $c \neq \mathit{PrimopErr}$) $d \sqsupseteq \alpha(c)$, as required.

Alternation Let $s = \textbf{if}\ e_g\ \textbf{then}\ e_t\ \textbf{else}\ e_f : \rho$, where wlog the guard evaluates to **true**. Let $s^g = e_g : \rho \Downarrow \textbf{true}$. As $\mathrm{Valid}_{\mathrm{CA}}(s) \Rightarrow \mathrm{Valid}_{\mathrm{CA}}(s^g)$, we may apply the inductive hypothesis to deduce that $t^g := e_g \Downarrow^\alpha c \sqsupseteq \alpha(\textbf{true})$.

Then as $\mathrm{Valid}_{\mathrm{CA}}(s) \Rightarrow \mathrm{Valid}_{\mathrm{CA}}(s')$, where $s' = e_t : \rho$, we may apply the inductive hypothesis again. Let $t' = e_t$. Then either $t' \Downarrow^\alpha *$, in which case $t \Downarrow^\alpha *$ as required, or $\mathrm{Valid}_{\mathrm{CA}}(v)$ holds and $t' \Downarrow^\alpha w \sqsupseteq \alpha(v)$. In the latter case $t \Downarrow^\alpha w$, as required.

Constructor Application Suppose that $s = \mathsf{C}(e_1, \ldots, e_n) : \rho$. Then $s \Downarrow \mathsf{C}(v_1, \ldots, v_n) = v$, where $s_i = e_i : \rho \Downarrow v_i$ for each i. As $\mathrm{Valid}_{\mathrm{CA}}(s) \Rightarrow \mathrm{Valid}_{\mathrm{CA}}(s_i)$, we may apply the inductive hypothesis.

We first note that for each i, $t_i = e_i \Downarrow^\alpha w_i \sqsupseteq \alpha(v_i)$ by the inductive hypothesis (where $w_i = *$ possibly), whence $t = \mathsf{C}(e_1, \ldots, e_n) \Downarrow^\alpha \mathsf{C} = \alpha(v)$. It remains to show that $\mathrm{Valid}_{\mathrm{CA}}(v)$ holds.

Fix $i \leq n$. Then as $e_i \Downarrow^\alpha w_i$, $w_i \in \gamma^{\mathbb{P}}(V)$, where $V = \mathrm{values}(\mathsf{C}, i)$. As $\alpha(v_i) \sqsubseteq w_i$, $v_i \in \gamma^{\mathbb{P}}(V)$. Furthermore, suppose that $* \notin V$. Then $e_i \not\Downarrow^\alpha v$, and thus by the inductive hypothesis, since $s_i \Downarrow v_i$, $\mathrm{Valid}_{\mathrm{CA}}(v_i)$ holds, as required.

Pattern Matching Let $s = \textbf{match}\ e\ \textbf{with}\ \left\langle \mathsf{C}_i(x_1^i, \ldots, x_{n_i}^i) \to e_i \right\rangle_{i=1}^k : \rho$. Suppose further that $s^m = e : \rho \Downarrow \mathsf{C}_l(v_1, \ldots, v_{n_l})$ and $s' = e^l : (\rho \oplus \{x_j^l \mapsto v_j\}_{j=1}^{n_l}) \Downarrow v$. We write ρ' for the environment of s'.

As $\mathrm{Valid}_{\mathrm{CA}}(s) \Rightarrow \mathrm{Valid}_{\mathrm{CA}}(s^m)$, we may apply the inductive hypothesis. Then $t^m = e \Downarrow^\alpha u \sqsupseteq \mathsf{C}_l$.

We shall first show that $\mathrm{Valid}_{\mathrm{CA}}(s')$ holds. Certainly for each $x \in \mathrm{dom}\,\rho$, $\rho'(x) = \rho(x)$ satisfies the validity condition, as $\mathrm{Valid}_{\mathrm{CA}}(s)$ holds. We now consider the case of $x = x_j^l \in \mathrm{dom}\,\rho' \setminus \mathrm{dom}\,\rho$.

There are two cases: either $t^m \Downarrow^\alpha *$ or $t^m \not\Downarrow^\alpha *$. If $t^m \Downarrow^\alpha *$, then $\mathrm{values}(x) = \mathit{AbsVal} \cup \{*\}$. Hence certainly $v_j \in \gamma^{\mathbb{P}}(\mathrm{values}(x))$. Now if $t^m \not\Downarrow^\alpha *$, then $u \neq *$, so by the inductive hypothesis, $\mathrm{Valid}_{\mathrm{CA}}(\mathsf{C}(v_1, \ldots, v_{n_l}))$ holds. In particular, $v_j \in \gamma^{\mathbb{P}}(\mathrm{values}(x))$. Furthermore, as $\mathrm{Valid}_{\mathrm{CA}}(\mathsf{C}(v_1, \ldots, v_{n_l}))$ holds and $* \notin \mathrm{values}(\mathsf{C}, j)$, $\mathrm{Valid}_{\mathrm{CA}}(v_j)$ holds, as required.

Having verified that $\mathrm{Valid}_{\mathrm{CA}}(s')$ holds, we are free to apply the inductive hypothesis. Thus $t' = e^l \Downarrow^\alpha w \sqsupseteq \alpha(v)$, whence $t \Downarrow^\alpha w$. If $t' \Downarrow^\alpha *$ then $t \Downarrow^\alpha *$ and we are done; otherwise $w \neq *$ and thus $\mathrm{Valid}_{\mathrm{CA}}(v)$ holds by the inductive hypothesis, as required.

Function Application Suppose that $s = e_1 e_2 : \rho \Downarrow v$, where $s_1 = e_1 : \rho \Downarrow (\!|f : \mu|\!) = s_b$, $s_2 = e_2 : \rho \Downarrow v_x$ and $s' = (\!|f : \mu \oplus \{x \mapsto v_x\}|\!) \Downarrow v$, where x is the first parameter of f not bound in μ.

As $\mathrm{Valid}_{\mathrm{CA}}(s_1)$ and $\mathrm{Valid}_{\mathrm{CA}}(s_2)$ hold, we may apply the inductive hypothesis. Thus $t_1 = e_1 \Downarrow^\alpha t_b \sqsupseteq \alpha(s_b)$, while $t_2 = e_2 \Downarrow^\alpha w_x \sqsupseteq \alpha(v_x)$. Either of t_1 and t_2 may evaluate to $*$, and we distinguish several cases.

Case 1. Suppose first that $t_1 \Downarrow^\alpha *$. Then as $t_2 \Downarrow w_x$, and $\mathit{Bind} * w_x = *$, we have: $t \Downarrow^\alpha w$ whenever $* \Downarrow^\alpha w$. In particular, $t \Downarrow^\alpha *$, and no further proof is required.

Case 2. Now suppose that $t_1 \not\Downarrow^\alpha *$, and $t_2 \Downarrow^\alpha *$. Then $t_b = (\!|f : S|\!)$ necessarily, where $S = \mathrm{dom}\,\mu$. Furthermore, by the inductive hypothesis $\mathrm{Valid}_{\mathrm{CA}}(s_b)$ holds. We must show that $\mathrm{Valid}_{\mathrm{CA}}(s')$ holds. As the environment of s' is that of s with the binding $x \mapsto v_x$ added, it suffices to show that the validity condition holds for x. But $* \in \mathrm{values}(x)$ as $t_2 \Downarrow^\alpha *$, whence $\gamma^{\mathbb{P}}(\mathrm{values}(x)) = \mathit{Value}$ and this is trivial.

Hence $\mathrm{Valid}_{\mathrm{CA}}(s')$ holds and we are free to apply the inductive hypothesis. Hence $t' = (\!|f : S \cup \{x\}|\!) \Downarrow^\alpha w \sqsupseteq \alpha(v)$, and thus $t \Downarrow^\alpha w$. As before, this gives the required result, deducing $\mathrm{Valid}_{\mathrm{CA}}(v)$ whenever $t \not\Downarrow^\alpha *$.

Case 3. Finally, suppose that $t_1 \not\Downarrow^\alpha *$ and $t_2 \not\Downarrow^\alpha *$. By the inductive hypothesis $\mathrm{Valid}_{\mathrm{CA}}(s_b)$ and $\mathrm{Valid}_{\mathrm{CA}}(v_x)$ hold.

We show that $\mathrm{Valid}_{\mathrm{CA}}(s')$ holds. Let $y \in \mathrm{dom}\,\mu'$, where $\mu' = \mu \oplus \{x \mapsto v_x\}$. If $y \neq x$, then $\mu'(x) = \mu(x)$, whence as $\mathrm{Valid}_{\mathrm{CA}}(s)$ holds, the validity condition for $\mu'(x)$ holds. Now consider the value of x. As $e_2 \Downarrow^\alpha w_x \sqsupseteq \alpha(v_x)$, we have $v_x \in \gamma^{\mathbb{P}}(\mathrm{values}(x))$. Furthermore, $\mathrm{Valid}_{\mathrm{CA}}(v_x)$ by the inductive hypothesis, as required.

Hence $\mathrm{Valid}_{\mathrm{CA}}(s')$ holds, and we may deduce the result as in case 2 above.

Variables Let $s = x : \rho \Downarrow \rho(x) = v$. We have two cases: x may be a function parameter or a pattern matching variable.

Case 1. Suppose that x is a parameter, say x is a parameter of function f, and S is the set of parameters occurring before x in the definition of f. Now $\mathrm{Valid}_{\mathrm{CA}}(s)$ holds, and thus $\rho(x) \in \gamma^{\mathbb{P}}(\mathrm{values}(x))$. There are two cases.

If for some $e_1 e_2 \in P$, $e_1 \Downarrow^\alpha *$, then $x \Downarrow^\alpha *$ and we are done.

Otherwise, as $\rho(x) \in \gamma^{\mathbb{P}}(\mathrm{values}(x))$, there exists $e_1 e_2 \in P$ where $e_1 \Downarrow^\alpha (\!|f : S|\!)$, and $e_2 \Downarrow^\alpha w \sqsupseteq \alpha(\rho(x))$.

If $e_2 \Downarrow^\alpha *$ for some application site $e_1 e_2$ with $e_1 \Downarrow^\alpha (\!|f : S|\!)$, then $x \Downarrow^\alpha *$ and we are done. Otherwise, $* \notin \mathrm{values}(x)$, and so as $\mathrm{Valid}_{\mathrm{CA}}(s)$ holds, $\mathrm{Valid}_{\mathrm{CA}}(\rho(x))$ necessarily holds. Furthermore, $x \Downarrow^\alpha w \sqsupseteq \alpha(\rho(x))$, as required.

Case 2. Now suppose that x is bound in a pattern matching expression, say of the form $\mathbf{match}\ e\ \mathbf{with}\ \langle \mathsf{C}_i(x_1^i, \ldots, x_{n_i}^i) \to e^i \rangle_{i=1}^n$.

First suppose that $e \Downarrow^\alpha *$. Then $x \Downarrow^\alpha *$ and we are done.

Otherwise, as $\mathrm{Valid}_{\mathrm{CA}}(s)$ holds, $v = \rho(x) \in \mathrm{values}(x) = \mathrm{values}(\mathsf{C}_i, j)$, where $x = x_i^j$. Hence there exists a constructor application $\mathsf{C}_i(e_1, \ldots, e_n)$ in P such that $e_j \Downarrow^\alpha w \sqsupseteq \alpha(v)$.

If for some such constructor application, $e_j \Downarrow^\alpha *$, then $x \Downarrow^\alpha *$ and we are done.

Otherwise, $* \notin \text{values}(x) = \text{values}(\mathsf{C}_i, j)$ and thus as $\text{Valid}_{\text{CA}}(s)$ holds, $\text{Valid}_{\text{CA}}(\rho(x))$ holds. Finally, $x \Downarrow^\alpha w \sqsupseteq \alpha(v)$.

Program Template Variables Suppose finally that $s = X : \rho \Downarrow \mathfrak{V}(X) = v$, where X is a program template variable. Then $t = X \Downarrow^\alpha \mathfrak{W}(X) = w$. Now as $\mathfrak{W}$ is compatible with $\mathfrak{V}$, $w \sqsupseteq \alpha(v)$, as required. Furthermore, by assumption either $w = *$ (in which case we are done), or $w \in \mathit{AbstConst}$. In the latter case, $\text{Valid}_{\text{CA}}(v)$ holds trivially, completing the proof. □

Lemma 4.23. *Let s be a state such that $\mathit{Valid}_{CA}(s)$. Suppose that $s \to s'$. Then $\mathit{Valid}_{CA}(s')$, and whenever $t \sqsupseteq \alpha(s)$, there exists $t' \sqsupseteq \alpha(s')$ such that $t \to^\alpha t'$.*

Proof. The proof of this result is an immediate consequence of the proof of Lemma 4.22. For, the states s' such that $s \to s'$ are just those states that the proof of $s \Downarrow v$ depends on. We have therefore shown that $\text{Valid}_{\text{CA}}(s')$ for each such state s' in the previous proof. The fact that $t \to^\alpha t'$ for some $t' \sqsupseteq \alpha(s')$ whenever $t \sqsupseteq \alpha(s)$ is trivial by definition of the $\to^\alpha$ relation, given Lemma 4.22. □

It is easy to deduce from Lemmata 4.22 and 4.23 that the 0CFA analysis is control-safe:

Corollary 4.24 (Control-Safety of 0CFA)**.** *Let P be a program and **start** the start state of P. Suppose that **start** $\to^* s$, and let $t \sqsupseteq \alpha(s)$. Then if $s \Downarrow v$ (resp. $s \to s'$) there exists $t' \sqsupseteq \alpha(s')$ (resp. $w \sqsupseteq \alpha(v)$) such that $t \to^\alpha t'$ (resp. $t \Downarrow^\alpha w$).*

Proof. The start state **start** is of the form $e : \emptyset$. Hence as the environment is empty, $\text{Valid}_{\text{CA}}(\textbf{start})$ is trivial. Therefore by Lemma 4.23 and induction on the length of the call sequence, whenever **start** $\to^* s$, then $\text{Valid}_{\text{CA}}(s)$ holds. We may now apply Lemmata 4.22 and 4.23 to obtain the result. □

Dataflow Safety We have shown that the $\Downarrow^\alpha$ and $\to^\alpha$ relation are *control-safe* and thus approximate the control flow of the program correctly. It now remains to show dataflow safety. That is, we must show that whenever $t \to^\alpha t'$, if s and s' are such that $s \in \gamma(t)$, $s' \in \gamma(t')$ and $s \to s'$, then some $\delta \in G(t \to^\alpha t')$ is safe for (s, s'). However, this property is entirely immediate given the results of Section 3.3 — the graph constructions that appear in the abstract interpretation are of course chosen to approximate the dataflow graphs defined then. We shall therefore not come back to dataflow safety for the alternative static analyses to follow.

4.4.4 A Worked Example

Let us now make the discussion of the 0CFA-based size-change termination analysis more concrete with an example. We shall use a folklore program for computing the standard

fold-left function on lists as an instance of *fold-right* building a function. Let us first recall, informally:

$$\begin{aligned} \mathit{foldr}\ (\oplus)\ e\ [x_1;\ldots;x_n] &= x_1 \oplus (x_2 \oplus \cdots (x_{n-1} \oplus (x_n \oplus e))) \\ \mathit{foldl}\ (\oplus)\ e\ [x_1;\ldots;x_n] &= (((e \oplus x_1) \oplus x_2) \oplus \cdots x_{n-1}) \oplus x_n \end{aligned}$$

Thus the fold functions compute the "sum" of elements of a list (with the given operator $\oplus$), bracketing to the right and left respectively. If the $\oplus$ operator is associative, the two functions are equivalent.

We now observe that:

$$\begin{aligned} & \mathit{foldl}\ (\oplus)\ e\ [x_1;\ldots;x_n] \\ = \ & (((e \oplus x_1) \oplus x_2) \oplus \cdots x_{n-1}) \oplus x_n \\ = \ & \big((\oplus x_n) \circ (\oplus x_{n-1}) \circ \cdots \circ (\oplus x_2) \circ (\oplus x_1)\big)\ e \end{aligned}$$

where $\circ$ denotes function composition, and $(\oplus x)$ is the operator section $\lambda y.y \oplus x$.

This suggests an alternative way of computing fold-left: compute the composition of the n operator sections $(\oplus x_i)$ as above, and apply the resulting function to e. As function composition is associative we may use fold-left or fold-right to compute the composition; we will of course choose fold-right. We obtain the following program (in OCaml syntax):

```
let rec foldr f e =
  function
      [] → e
    | x :: xs → f x (foldr f e xs)

let foldl' f e xs =
  let step a g x = g (f x a)
  and id x = x in
    foldr step id xs e
```

The definition of *foldr* is standard. The definition of *foldl'* uses the *step* function to add another term to the function composition. The *id* function represents the base case.

We shall show the 0CFA size-change termination analysis of this program. First, we must convert this OCaml program into our core language, using λ-lifting to eliminate local declarations (we also rename variables to ensure unique names):

```
foldr h e xs =
  match xs with
      [] → e
    | y :: ys → h y (foldr h e ys)
```

Binding Construct	Parameter	Values
foldr	h	$(\!\|step : \{f\}\|\!)$
	e	$\top_c$
	xs	$*$
foldr.**match**	y	$*$
	ys	$*$
step	f	$(\!\|sum : \emptyset\|\!)$
	a	$*$
	g	$(\!\|id : \emptyset\|\!)$ $(\!\|step : \{f, g\}\|\!)$
	x	$\top_c$
id	z	$\top_c$
foldl'	f	$(\!\|sum : \emptyset\|\!)$
	b	$\top_c$
	zs	$*$

Figure 4.4: Analysis of *foldl' sum* $\langle * \rangle$ $\langle * \rangle$: Closure Analysis

```
step f a g x = g (f x a)

id z = z

foldl' f b zs = foldr (step f) id zs b
```

In order to prove termination of *foldl'*, we must provide an expression to evaluate (defining the meaning of this program). Intuitively *foldl'* terminates for any base value b, and list zs and any terminating function f. However, our analysis cannot handle arbitrary parameters of function type, so we must fix a specific function f (say the sum function for definiteness). We therefore add the following:

```
sum n m = n + m

foldl' sum <X> <Y>
```

We shall choose valuation $\{X \mapsto \top_c, Y \mapsto *\}$.

The results of closure analysis on this program are given in Figure 4.4. This maps each variable in the program to a set of abstract values approximating the set of values that this variable might be bound to. The most significant set of values is that of the g parameter of *step*. To find its possible values, we look for all expressions $e_1 e_2$, where $e_1 \Downarrow^{\alpha} (\!|step : \{f, a\}|\!)$. By inspection of the program we can see that the only textual occurence of *step* occurs as an argument to *foldr*, and as a result the only such binding site is the right-hand side of the $y :: ys$ case of *foldr*. The values of g are thus deemed to be all possible values of *foldr f e ys*. By inspection again we can convince ourselves that these values are $(\!|id : \emptyset|\!)$ (corresponding to the base case) and $(\!|step : \{f, a, g\}|\!)$ (for the recursive step).

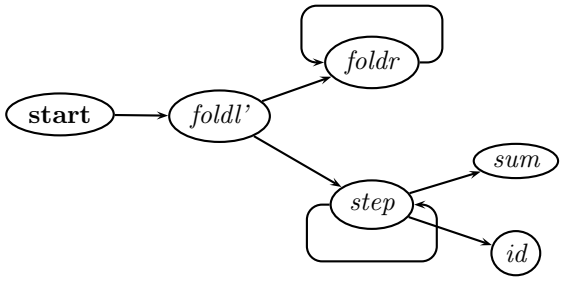
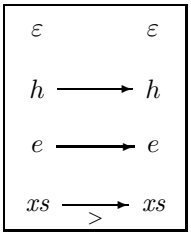


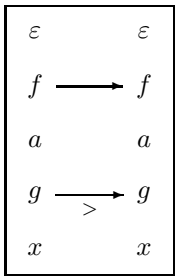
Figure 4.5: Analysis of *foldl' sum* $\langle * \rangle$ $\langle * \rangle$: Call Graph (excerpt)

The results of closure analysis for g are significant because the possible value $(\!|step : \{f, a, g\}|\!)$ leads to a self-call: *step* applies g, which may itself be an instance of *step*. This self-call is of course not textually apparent in the program.

Given the results of closure analysis, we may compute the abstract call graph of this program. A simplified outline of this call graph is shown in Figure 4.5. As noted before this shows a self-loop at the *step* function in addition to the (obvious) recursive call from *foldr* to itself. The **start** node denotes the expression defining the meaning of the program.

To conclude this example let us examine the size-change termination properties of the two self-loops identified. The size-change graphs are shown below:





The size-change graph for the *foldr* recursive loop is unsurprising: *foldr* is defined by primitive recursion over a list, and hence the list decreases in size at each self-call, giving size-change termination of *foldr*.

The *step* self-loop is of more interest. As the size-change graph for this loop shows, this loop is guaranteed to terminate because of a decrease in the g parameter. The decrease occurs because this self-loop is caused by the application of parameter g — as illustrated in Example 3.16 (p. 35), the use of the subtree order on higher-order values is justified by this useful property. Another way to see this is by observing the similarity with the value shown in Figure 4.1 (p. 57). Each recursive call decreases the size of the subtree representing the g parameter.

The *foldl'* example is therefore size-change terminating under the 0CFA analysis. It should be noted that our presentation of this example was chosen for illustrative purposes,

and does not reflect the full behaviour of the analysis. In particular, closure analysis and call graph construction are not successive phases, but rather are interdependent.

4.4.5 Discussion

Cost

Let us estimate the cost of the 0CFA-based analysis. Let $\mathcal{E}$ be the set of subexpressions of the program (so that $|\mathcal{E}|$ is the size of the program), and $\mathcal{F}$ be the set of function definitions in the program. Let $\mathcal{T}$ be the set of type declarations in the program, and $\mathcal{C}$ be the set of constructors defined in these.

The abstract domain consists of: expression states e, function closure states $(\!|f : S|\!)$ (where $S \subseteq \mathit{Params}(f)$), primitive constant values $c \in \mathit{AbstConst}$, constructor values C and the special arbitrary value $*$.

The number of expression states is $|\mathcal{E}|$, while the number of closure states is given by $\sum_{f \in \mathcal{F}}(|\mathit{Params}(f)| + 1)$. Indeed, for each function f, the only closures $(\!|f : S|\!)$ that may be constructed are those in which S is a prefix of the (ordered) list of parameters of f; there are $|\mathit{Params}(f)| + 1$ of these. The number of constructor states is just $|\mathcal{C}|$.

Expression states and fully applied closure states never represent values, while all other states always represent values. The size of the set of *abstract values* is therefore:

$$|\text{Abstract Values}| = \sum_{f \in \mathcal{F}} |\mathit{Params}(f)| + |\mathcal{C}| + 1 + |\mathit{AbstConst}|$$

Let us examine this more closely. Note first that we expect $|\mathit{AbstConst}|$ to be small in any reasonable abstract domain for SCT. The number of states is dominated by the term $\sum_{f \in \mathcal{F}} |\mathit{Params}(f)|$. This is not strictly linear in the size of the program (in the worst case it is quadratic). However, we appeal to the observation that most functions have a small number of arguments to argue that in practice this term is linear in the size of the program[6]. The number of constructors $|\mathcal{C}|$ is similarly expected to grow sublinearly with the size of the program.

The number of values is therefore almost linear in the size of the program. Let us now look at the number of proper states in the abstract domain. This is more relevant to the efficiency of the analysis, as values are terminal nodes in the call graph, whereas states may have many call edges. The proper states are just expression and full closure states:

$$|\text{Proper States}| = |\mathcal{E}| + |\mathcal{F}|$$

This is, of course, linear in the size of the program.

[6] It is certainly the case that human-written programs typically have a small, bounded number of arguments per function. Our use of λ-lifting may, however, increase the maximum number of function parameters substantially. This may be an argument against the use of λ-lifting prior to analysis, though it is possible to mitigate this problem somewhat by treating parameters added by λ-lifting as a special case.

The number of nodes in the call graph is bounded above by the sum of the sizes of the sets of values and proper states above. This is again almost linear in the size of the program. The number of nodes in the call graph is expected to be near this maximum (in particular, in the absence of dead code all expression and closure states should be in the call graph.

We may hope however that in non-pathological cases the call graph is sparse, though in general the size (number of edges) of the call graph is quadratic.

Precision

The 0CFA analysis is *context-insensitive*, so that each function body is only analysed once. As shown above this translates into an inexpensive analysis. An important question that arises is whether the lack of precision of 0CFA is indeed problematic in practice. In the following we highlight a number of salient precision issues in 0CFA.

Commonly-Used Higher-Order Functions Let us first consider the case of functions such as *map* (or, as shown before, the fold functions). It should be clear that these are size-change terminating under 0CFA, in the sense that if f is a (size-change) terminating function, the program evaluation *map f xs* is size-change terminating. However, this property does not carry over to programs containing *multiple calls* to *map*.

Consider a program containing a number of calls to map, say with explicit function arguments for simplicity, and assume that the calls are *map* f_1 e_1 through *map* f_n e_n (where the e_i are terminating expressions). Assume also that each f_i has a single parameter. Then let A_i be the expression denoting the i^{th} call to *map*. By soundness we must necessarily have $A_i \rightarrow^{\alpha} (\!|map : S|\!)$, where $S = Params(map)$. Furthermore, the i^{th} call to *map* (in the exact semantics) calls a state of the form $S_i := (\!|f_i : Params(f_i)|\!)$. By soundness we must therefore have $(\!|map : S|\!) \rightarrow^{\alpha *} S_i$. Thus for each i, j, $A_i \rightarrow^{\alpha *} S_j$.

Taking a step back we find that each of the applied occurrences of *map* call *any* of the functions that appear as arguments to *map*. This degrades the precision of the call graph substantially, and indeed it is not too difficult to construct an example where this prevents a terminating program from being recognised as such (we omit the definition of *map*):

```
id x = x

k us = map id (1 :: us)

h vs = map k vs

h <X>
```

This artificial example is a terminating OCaml program; it takes the list $[[1];[2];[3]]$ to $[[1;1];[1;2];[1;3]]$. However, this is *not* size-change terminating in the 0CFA analysis. For, the k function calls *map*; in turn *map* calls k (as k is a parameter to *map*). However, there is no

size-change information in the $k \to k$ call (this is the purpose of calling *map* with argument $1 :: us$ in k, as this prevents what is otherwise a decrease in the list value).

This property points to a lack of *compositionality* in 0CFA analysis — though each individual call to *map* is size-change terminating, a program with many calls to *map* may not be. This is not in itself surprising, as the size-change termination property is not naturally compositional (though termination is, to an extent). Indeed, none of the analyses that we will detail next will be compositional. The 0CFA analysis does fail rather too readily in such circumstances, which leads to poor results in practice.

Imprecision of Closure Analysis The previous discussion described a problem that occurs with many commonly-used functions, due to the nature of the abstract domain (specifically, the fact that there is only one instance of each function). In this section we shall introduce a different kind of problem, caused by an inherent imprecision of the closure analysis mechanism.

The issue here is with *repeated* occurrences of function parameters inside a function body. Consider a function g with (functional) parameter f, and say that g applies f twice. The value of f is approximated by closure analysis, leading to a set of values (in general, not a singleton set). At each application of f, all values from this set are taken as possible values for f.

This leads to a loss of precision: if the set of values of f is $\{(\!|f_1 : S_1|\!), (\!|f_2 : S_2|\!)\}$, then we know that either both occurrences of f call f_1, or they both call f_2. However, the 0CFA analysis does not rule out the cases where one occurrence calls f_1 and the other calls f_2.

The reason for this behaviour is straightforward: we require that each function body be analysed once. However, if variable values are to be coerced to be consistent, each function body must be analysed in a *context* giving values to variables, and thus should be analysed for each context. For a function with n parameters x_i, each of which has values in S_i, there are $\prod_i |S_i|$ contexts. In general this is exponential in the size of the program, justifying the choice of the simpler, but less precise, analysis.

As an example of this, let us consider the following program:

```
id x = x

f h = h id

g a b = g a b

test p = p (p id)

if <X : bool> then test f else test g
```

Then there are two calls to *test*, with p either representing f or g. The key point is that

f (f id) terminates with value $(\!|id : \emptyset|\!)$; and g (g id) terminates with value represented by $(\!|g : \{a\}|\!)$. However, f (g id) causes the result of g id to be applied, entailing evaluation of the body of g. This is of course nonterminating.

Accordingly this example is *not* size-change terminating under 0CFA. That this is due to closure analysis is demonstrated by the fact that if the *test* function is inlined, so that the expression that the program evaluates is

```
if <X : bool> then f (f id) else g (g id)
```

then the program becomes size-change terminating under 0CFA.

It is therefore indeed the case that the fact that 0CFA ignores relationships between variables, and furthermore ignores relationships between occurrences of the same variable, strictly reduces the expressive power of the analysis. However, unlike the previous source of imprecision, we contend that this is unlikely to be relevant in practice — indeed, the author had some difficulty in finding the above example, and as it stands this is not even typable.

4.5 A Generalisation: k-bounded CFA

4.5.1 Motivation

In the previous section, we have described 0CFA, a context-insensitive flow analysis. In 0CFA each function body is only analysed once, which leads to an inexpensive analysis, but one that is rather imprecise. Indeed we have shown an example of a natural program that fails to be size-change terminating under 0CFA due to the coarse call graph constructed.

In this section we shall introduce a generalisation of 0CFA, namely k-bounded CFA. Where in 0CFA we discarded all information from environments, in k-bounded CFA we shall keep environment information up to a fixed depth k. Of course, 0CFA arises as a special case of k-bounded CFA with $k = 0$, but it is worthwhile to introduce 0CFA independently, as it is simpler and possesses some unique properties, such as context-insensitivity.

The Abstract Domain

As we have remarked previously, states are represented as trees of bounded branching factor and arbitrary depth. The branching factor (maximum number of successors of any node) is bounded by the maximum arity of a function, constructor or primitive operator appearing in the program. Furthermore, internal nodes are labelled with function names and primitive operators, so that there are only finitely many of these. Terminal nodes, on the other hand, may be labelled with constants, of which there are infinitely many.

The abstract domain in k-bounded CFA is a finite approximation of this infinite set of trees. As before primitive constants are mapped to abstract constants, guaranteeing finiteness. The key difference is that we now keep k levels of the tree in the abstract state.

As an example, let us define the k-bounded CFA abstraction function α_k for closure states. This is defined by recursion on k, as follows:

$$\begin{aligned} \alpha_0(\langle\!|f : \rho|\!\rangle) &= \langle\!|f : \operatorname{dom}\rho|\!\rangle \\ \alpha_{k+1}(\langle\!|f : \rho|\!\rangle) &= \langle\!|f : \{x \mapsto \alpha_k(v) \mid \rho(x) = v\}|\!\rangle \end{aligned}$$

To wit, the base case is just 0CFA — no information is kept beyond the domain of the environment. In the inductive case ($(k+1)$-bounded CFA) we apply k-bounded CFA to all the values in the environment. The effect of this is to cut off all environment paths after k nodes.

Rationale

Before proceeding to define k-bounded CFA formally let us explore the intuition behind this abstraction further. In the case of 0CFA we have observed that if a program contains many calls to *map*, the call graph contains calls from *map* to each function that was passed as a parameter to *map* somewhere in the program. We have further shown that this can lead to spurious counterexamples to termination.

We expect, however, that it is common for programs to use functions such as *map* many times in many unrelated contexts. The effect of 0CFA is thus to create spurious calls between otherwise unrelated parts of the program, with a negative impact on our ability to prove termination. It is therefore probable that the likelihood of 0CFA proving termination *decreases* with the size of the program, even if the program does not otherwise increase in complexity.

The k-bounded CFA analysis attempts to deal with precisely this problem. In 1-bounded CFA, for each call to *map* with a distinct function f, a new abstract state is created. We argue that this is often enough to prevent precision from degrading as the program grows. For, closure states of high depth are rare in practice, and thus we may expect k-bounded CFA for low values of k ($k = 1, 2$) to recover much of the use of higher-order functions accurately.

4.5.2 Static Analysis

The Abstract Domain

In k-bounded CFA, the abstraction function maps a state to the tree obtained by cutting off all environment paths at length at most k, and replacing primitive constants with their abstract equivalents. The abstraction function is readily defined, where α_k is the abstraction function for k-bounded CFA. The base case is just 0CFA (which we shall not repeat here),

while for $k > 0$ we apply $(k-1)$-bounded CFA to substates:

$$\begin{aligned}
\alpha_{k+1}(e^l : \rho) &= e^l : \alpha_k(\rho) \\
&\quad \textbf{where } \alpha_k(\rho) = \{x \mapsto \alpha_k(v) \mid x \in \operatorname{dom}\rho \wedge \rho(x) = v\} \\
\alpha_{k+1}(\langle\!|f : \rho|\!\rangle) &= \langle\!|f : \alpha_k(\rho)|\!\rangle \\
\alpha_{k+1}(\mathsf{C}(v_1, \ldots, v_n)) &= \mathsf{C}(\alpha_k(v_1), \ldots, \alpha_k(v_n))
\end{aligned}$$

Finally, $\alpha_k(c) = \alpha^C(c)$ for all k: abstracting constant values is, as before, achieved using α^C.

As before, we further introduce the special value $*$, to model arbitrary values in the analysis. This may also appear as a substate of other k-bounded CFA states.

The abstract domain $AbsState_k$ of k-bounded CFA may be formally defined as follows:

$$\begin{aligned}
AbsState_0 &= AbstConst \cup \{e^l \mid e^l \in P\} \cup \{\langle\!|f : S|\!\rangle \mid S \subseteq Params(f)\} \\
&\quad \cup \{\mathsf{C} \in Constr\} \cup \{*\} \\
AbsState_{k+1} &= AbstConst \\
&\quad \cup \{e^l : \rho \mid e^l \in P \wedge \operatorname{dom}\rho \subseteq \operatorname{Scope}(e^l) \wedge (\forall x \in \operatorname{dom}\rho)\rho(x) \in AbsState_k\} \\
&\quad \cup \{\langle\!|f : \rho|\!\rangle \mid \operatorname{dom}\rho \subseteq Params(f) \wedge (\forall x \in \operatorname{dom}\rho)\rho(x) \in AbsState_k\} \\
&\quad \cup \{\mathsf{C}(v_1, \ldots, v_n) \mid n = \sharp\mathsf{C} \wedge (\forall i)v_i \in AbsState_k\} \cup \{*\}
\end{aligned}$$

This just corresponds to the set of trees of depth at most k, with nodes labelled with expressions, closures and abstract constants.

Ordering Abstract States To conclude the description of the abstract domain, we define the order on k-bounded CFA states. This is the pointwise order generated by the following:

1. Primitive constants are compared with $\sqsubseteq^C$
2. Whenever v is an abstract state denoting a set of *values*, $v \sqsubseteq *$.

The second case relies on the observation that, as in the 0CFA case, each abstract state unambiguously denotes *either* a set of values, *or* a set of states, none of which are values. We let $AbsVal$ denote the set of abstract states representing values.

More precisely,

$$\begin{aligned}
\langle\!|f : \rho|\!\rangle \sqsubseteq \langle\!|f : \mu|\!\rangle \quad &\text{iff} \quad (\forall x)\ \rho(x) \sqsubseteq \mu(x) \\
e^l : \rho \sqsubseteq e^l : \mu \quad &\text{iff} \quad (\forall x)\ \rho(x) \sqsubseteq \mu(x) \\
\mathsf{C}(v_1, \ldots, v_n) \sqsubseteq \mathsf{C}(w_1, \ldots, w_n) \quad &\text{iff} \quad (\forall i)v_i \sqsubseteq w_i \\
v \sqsubset * \quad &\text{iff} \quad v \in AbsVal \\
c \sqsubseteq c' \quad &\text{iff} \quad c \sqsubseteq^C c'
\end{aligned}$$

Graph Bases

In 0CFA, the graph basis of a state was the *1-limited* graph basis: this contained environment paths of depth at most 1. By analogy, in k-bounded CFA the graph basis of an abstract state is the $(k+1)$-limited graph basis. For, the tree representing a k-bounded CFA state precisely encodes environment paths up to depth k, and we may deduce the environment paths of length $k+1$ as the immediate parameters of leaf nodes are known (for instance, for a leaf node of the form $(\!|f : S|\!)$, the parameters are just the members of S).

We may define this more precisely as follows: the graph basis of a k-bounded CFA state is defined recursively, with

$$\begin{aligned}
\mathrm{gb}^\alpha(c) &= \{\varepsilon\} \\
\mathrm{gb}^\alpha(e^l : \rho) = \mathrm{gb}^\alpha((\!|f : \rho|\!)) &= \{\varepsilon\} \cup \{x : p \mid x \in \mathrm{dom}\,\rho \wedge p \in \mathrm{gb}^\alpha(\rho(x))\} \\
\mathrm{gb}^\alpha(\mathsf{C}(v_1, \ldots, v_n)) &= \{\varepsilon\} \cup \{x_i : p \mid 1 \leq i \leq \sharp\mathsf{C} \wedge p \in \mathrm{gb}^\alpha(v_i)\} \\
\mathrm{gb}^\alpha(*) &= \{\varepsilon\}
\end{aligned}$$

The base case is just the graph basis of a 0CFA state:

$$\begin{aligned}
\mathrm{gb}^\alpha(e^l) &= \{\varepsilon\} \cup \{\langle x \rangle \mid x \in \mathrm{Scope}(e^l)\} \\
\mathrm{gb}^\alpha((\!|f : S|\!)) &= \{\varepsilon\} \cup \{\langle x \rangle \mid x \in S\} \\
\mathrm{gb}^\alpha(\mathsf{C}) &= \{\varepsilon\} \cup \{\langle x_i \rangle \mid 1 \leq i \leq \sharp\mathsf{C}\}
\end{aligned}$$

Closure Analysis

In 0CFA, closure analysis was used to recover (approximate) values for bound variables, as 0CFA states include no environment information. In a k-bounded CFA state, each variable x is bound in the environment to a value v of depth at most $k-1$. This may therefore *either* be an exact value (if the depth of the value represented by v was less than $k-1$), in which case it can be recovered directly, *or* it is a depth $k-1$ approximation to a value.

The problem of closure analysis in k-bounded CFA is therefore the following: given a state with environment ρ and a variable $x \in \rho$, what is the set of (depth k) values that the depth $k-1$ values $\rho(x)$ can represent? That is, we must expand a tree of depth $k-1$ into the set of depth k trees that it could represent.

In order to find the set of depth-k trees represented by a tree of depth $k-1$, we may reuse the closure analysis operation from 0CFA, applying it to the *leaves* of the tree. For each variable x, we have defined the set values(x) to be the set of (depth-0) values found in closure analysis. We may expand a tree to a set of trees of higher depth by expanding leaf nodes to include all possible choices of bindings from values(x), where x is a variable bound in a leaf node.

More precisely, we define a relation $s \ll t$ (s expands to t) to encode this process. This is defined as follows. The base case applies 0CFA closure analysis to leaf nodes, turning a leaf node (for instance $(\!|f : S|\!)$) into a set of subtrees of depth 1:

$$\begin{aligned} c &\ll c \\ e^l \ll e^l : \{x \mapsto v_x \mid x \in \text{Scope}(l)\} \quad &\text{iff} \quad (\forall x)\ v_x \in \text{values}(x) \\ (\!|f : S|\!) \ll (\!|f : \{x \mapsto v_x \mid x \in S\}|\!) \quad &\text{iff} \quad (\forall x)\ v_x \in \text{values}(x) \\ \mathsf{C} \ll \mathsf{C}(v_1, \ldots, v_n) \quad &\text{iff} \quad (\forall i)\ v_i \in \text{values}(\mathsf{C}, i) \\ * &\ll * \end{aligned}$$

For the inductive case (internal nodes), we recursively expand each of the subtrees:

$$\begin{aligned} e^l : \rho \ll e^l : \mu \quad &\text{iff} \quad (\forall x)\ \rho(x) \ll \mu(x) \\ (\!|f : \rho|\!) \ll (\!|f : \mu|\!) \quad &\text{iff} \quad (\forall x)\ \rho(x) \ll \mu(x) \\ \mathsf{C}(v_1, \ldots, v_n) \ll \mathsf{C}(w_1, \ldots, w_n) \quad &\text{iff} \quad (\forall i)\ v_i \ll w_i \end{aligned}$$

Abstract Interpretation

The abstract interpretation for k-bounded CFA is defined in Figures 4.6 and 4.7 (for the evaluation judgement $\Downarrow^\alpha$), and Figure 4.8 (for the call relation $\rightarrow^\alpha$). We have separated rules that are analogues of the 0CFA equivalents (in Figure 4.6) from rules that affect the environment and therefore require a different treatment in k-bounded CFA (in Figure 4.7).

We must introduce a final item of notation. Consider a function application, in which (say) the operator evaluates to $(\!|f : \rho|\!)$ and the operand to v. This triggers evaluation of the state $(\!|f : \rho \oplus \{x \mapsto v\}|\!)$ for some x. However, if v is a depth-k state, the resulting state may have depth $k + 1$. That is, evaluating function applications may increase the depth of states (constructor applications likewise fall in this category). To ensure finiteness of the abstract domain, states must be limited to depth k. We therefore introduce the *k-restriction* of a state to be the depth-k approximation to a possibly higher depth state.

The defininition of the restriction operator $s \mapsto [s]_k$ is entirely as expected and thus omitted. This is equivalent to applying the abstraction map α_k to s, with only difference that s is here an abstract state.

4.5.3 Soundness

Let us now prove the k-bounded CFA analysis sound. As before we proceed by showing control-safety (dataflow-safety is once more straightforward and omitted). The soundness proof for k-bounded CFA is more direct than in the 0CFA case, as the safety properties of closure analysis have already been established.

Let us first prove that the operation of expanding a depth-$(k-1)$ state to a set of depth-k

Values and Function Names

$$\frac{}{v \Downarrow v, 1}\ {\scriptstyle v \in AbsVal} \qquad \frac{}{f : \rho \Downarrow (\!|f : \emptyset|\!), 0} \qquad \frac{}{c : \rho \Downarrow \alpha(c), 0}$$

Primitive Operators

$$\frac{(\forall i) e_i : \rho \Downarrow c_i, \gamma_i \quad \mathit{Apply}^{\alpha}\ \mathit{op}\ \langle c_1, \ldots, c_n \rangle = c}{\mathit{op}(e_1, \ldots, e_n) : \rho \Downarrow c, \mathit{OpGraph}\ \mathit{op}\ \langle (c_1, \gamma_1), \ldots, (c_n, \gamma_n) \rangle}$$

Evaluating Closures

$$\frac{\mathit{Body}(f) : \rho \Downarrow v, \gamma}{(\!|f : \rho|\!) \Downarrow v, \gamma^*}\ {\scriptstyle \mathrm{dom}\,\rho = \mathit{Params}(f)}$$

Conditionals

$$\frac{e_g : \rho \Downarrow c \sqsupseteq \alpha(\mathbf{true}), \gamma_g \quad e_t : \rho \Downarrow v, \gamma}{\mathbf{if}\ e_g\ \mathbf{then}\ e_t\ \mathbf{else}\ e_f : \rho \Downarrow v, \gamma^*} \qquad \frac{e_g : \rho \Downarrow c \sqsupseteq \alpha(\mathbf{false}), \gamma_g \quad e_f : \rho \Downarrow v, \gamma}{\mathbf{if}\ e_g\ \mathbf{then}\ e_t\ \mathbf{else}\ e_f : \rho \Downarrow v, \gamma^*}$$

Program Template Variables

$$\frac{}{X \Downarrow \mathfrak{W}(X)}\ {\scriptstyle \text{with valuation } \mathfrak{W}}$$

Figure 4.6: k-bounded CFA: Static Analysis (Evaluation, part 1)

Constructors and Pattern Matching

$$\frac{e_1 : \rho \Downarrow v_1, \gamma_1 \quad \cdots \quad e_n : \rho \Downarrow v_n, \gamma_n}{\mathsf{C}(e_1, \ldots, e_n) : \rho \Downarrow [\mathsf{C}(v_1, \ldots, v_n)]_k, [\mathrm{constr}(\gamma_1, \ldots, \gamma_n)]_{k+1}}$$

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

$$\frac{e : \rho \Downarrow * \quad e_l : \rho \oplus \{x_j^l \mapsto *\}_{j=1}^{n_l} \Downarrow v, \delta}{s = \mathbf{match}\ e\ \mathbf{with}\ \left\langle \mathsf{C}_i(x_1^i, \ldots, x_{n_i}^i) \to e_i \right\rangle_{i=1}^{k} \Downarrow v, \mathrm{patmatch}_s^l(\gamma); \delta}$$

Function Application

$$\frac{e_1 : \rho \Downarrow (\!|f : \mu|\!), \gamma_1 \quad e_2 : \rho \Downarrow w, \gamma_2 \quad [\mathit{Bind}\ (\!|f : \mu|\!)\ w]_k \Downarrow v, \delta}{e_1 e_2 : \rho \Downarrow v, [\mathrm{app}_x(\gamma_1, \gamma_2)]_{k+1}; \delta}\ {\scriptstyle S = \mathrm{dom}\,\mu \subsetneq \mathit{Params}(f) \wedge \atop x = \mathrm{hd}(\mathit{Params}(f) \setminus S)}$$

Closure Analysis

$$\frac{\rho(x) \ll v}{x : \rho \Downarrow v, \overline{1[\varepsilon/x]}}$$

Figure 4.7: k-bounded CFA: Static Analysis (Evaluation, Environment rules)

Primitive Operators

$$\frac{(\forall j < i) \quad e_j : \rho \Downarrow v_j}{op(e_1, \ldots, e_n) : \rho \to e_i : \rho, 1^*}\ i \leq n$$

Evaluating Closures

$$\frac{}{(\!|f : \rho|\!) \to Body(f) : \rho, 1^*}\ \mathrm{dom}\,\rho = Params(f)$$

Constructors and Pattern Matching

$$\frac{(\forall j < i) \quad e_j : \rho \Downarrow v_j}{\mathsf{C}(e_1, \ldots, e_n) : \rho \to e_i : \rho, 1^*}\ i \leq n$$

$$\frac{}{\begin{array}{c}\textbf{match}\ e\ \textbf{with}\ \left\langle \mathsf{C}_i(x_1^i, \ldots, x_{n_i}^i) \to e_i \right\rangle_{i=1}^{k} : \rho \to e : \rho, 1^* \\ e : \rho \Downarrow \mathsf{C}_l(v_1, \ldots, v_{n_l}), \gamma\end{array}}$$

$$\frac{}{\begin{array}{c}s = \textbf{match}\ e\ \textbf{with}\ \left\langle \mathsf{C}_i(x_1^i, \ldots, x_{n_i}^i) \to e_i \right\rangle_{i=1}^{k} \to e_l : \rho \oplus \{x_j^l \mapsto v_j\}_{j=1}^{n_l}, \mathrm{patmatch}_s^l(\gamma) \\ e : \rho \Downarrow *\end{array}}$$

$$\frac{}{s = \textbf{match}\ e\ \textbf{with}\ \left\langle \mathsf{C}_i(x_1^i, \ldots, x_{n_i}^i) \to e_i \right\rangle_{i=1}^{k} \to e_l : \rho \oplus \{x_j^l \mapsto *\}_{j=1}^{n_l}, \mathrm{patmatch}_s^l(\gamma)}$$

Conditionals

$$\frac{}{\begin{array}{c}\textbf{if}\ e_g\ \textbf{then}\ e_t\ \textbf{else}\ e_f : \rho \to e_g : \rho, 1^* \\ e_g : \rho \Downarrow c \sqsupseteq \alpha(\textbf{true})\end{array}} \qquad \frac{}{\begin{array}{c}\textbf{if}\ e_g\ \textbf{then}\ e_t\ \textbf{else}\ e_f : \rho \to e_t : \rho, 1^*\end{array}}$$

$$\frac{}{\begin{array}{c}\textbf{if}\ e_g\ \textbf{then}\ e_t\ \textbf{else}\ e_f : \rho \to e_f : \rho, 1^*\end{array}}$$

Function Application

$$\frac{}{\begin{array}{c}e_1 e_2 : \rho \to e_1 : \rho, 1^* \\ e_1 : \rho \Downarrow (\!|f : \mu|\!), \gamma_1\end{array}}\ \mathrm{dom}\,\mu \subsetneq Params(f)$$

$$\frac{}{\begin{array}{c}e_1 e_2 : \rho \to e_2 : \rho, 1^* \\ e_1 : \rho \Downarrow (\!|f : \mu|\!), \gamma_1 \quad e_2 : \rho \Downarrow w, \gamma_2\end{array}}\ S = \mathrm{dom}\,\mu \subsetneq Params(f) \wedge x = \mathrm{first}(Params(f) \setminus S)$$

$$e_1 e_2 \to [Bind\ (\!|f : \mu|\!)\ w]_k\ , [\mathrm{app}_x(\gamma_1, \gamma_2)]_{k+1}$$

Figure 4.8: k-bounded CFA: Static Analysis (Calls)

states is sound:

Lemma 4.25. *Let s be a state such that $\mathit{Valid}_{CA}(s)$ holds. Then if $t \sqsupseteq \alpha_{k-1}(s)$, then $u \sqsupseteq \alpha_k(s)$ for some u such that $t \ll u$. .*

Proof. The proof is by induction on k. We assume wlog that s is a closure state $(\!|f : \rho|\!)$, as other cases are similar. If $t = *$ the result is trivial.

For the base case ($k = 1$), $t = \alpha_0(s) = (\!|f : S|\!)$ where $S = \operatorname{dom}\rho$ necessarily. Now as $\mathrm{Valid}_{\mathrm{CA}}(s)$ holds, for each $x \in S$, $\rho(x) \in \gamma^{\mathbb{P}}(\mathrm{values}(x))$. That is, for each x there exists $v_x \in \mathrm{values}(x)$ such that $v_x \sqsupseteq \alpha_0(\rho(x))$. Let $u = (\!|f : \{x \mapsto v_x \mid x \in S\}|\!)$. Then $t \ll u$. Furthermore, $\alpha_1(s) = (\!|f : \{x \mapsto \alpha_0(\rho(x)) \mid x \in S\}|\!) \sqsubseteq u$, as $\alpha_0(\rho(x)) \sqsubseteq v_x$ for each x.

Now consider the inductive case. Let $t = (\!|f : \mu|\!) \sqsupseteq \alpha_{k-1}(s)$. Then for each $x \in S$, $\mu(x) \sqsupseteq \alpha_{k-2}(\rho(x))$. Hence by the inductive hypothesis, for each x there exists a values v_x such that $\mu(x) \ll v_x$ and $v_x \sqsupseteq \alpha_{k-1}(\rho(x))$. Let $u = (\!|f : \{x \mapsto v_x \mid x \in S\}|\!)$. Then $t \ll u$, and $u \sqsupseteq \alpha_k(s)$, as required. □

The dual of this result shows that restriction (of a depth-k state to a depth $k-1$ state) is likewise safe:

Lemma 4.26. *Let s be a state, and suppose that $\alpha_{k+1}(s) \sqsubseteq t$. Then $\alpha_k(s) \sqsubseteq [t]_k$.*

Proof. The proof is by induction on k, where as before we may assume wlog that $s = (\!|f : \rho|\!)$ and $t \neq *$.

For the base case, $t = (\!|f : \mu|\!)$, where $\operatorname{dom}\mu = \operatorname{dom}\rho = S$ say. Then $[t]_0 = (\!|f : S|\!) = \alpha_0(s)$, as required.

For the inductive case, let $t = (\!|f : \mu|\!)$, where for each $x \in S$, $\mu(x) \sqsupseteq \alpha_k(\rho(x))$. Then define ν by $\nu(x) = [\mu(x)]_{k-1}$ for each $x \in S$. By the inductive hypothesis $\nu(x) \sqsupseteq \alpha_{k-1}(\rho(x))$ for each $x \in S$, so that $[t]_k = (\!|f : \nu|\!) \sqsupseteq \alpha_k(s)$, as required. □

We may now prove soundness of k-bounded CFA:

Lemma 4.27. *Let s be a state such that $\textbf{\textit{start}} \rightarrow^* s$, and suppose that $s \Downarrow v$. Then whenever $t \sqsupseteq \alpha_k(s)$, there exists $w \sqsupseteq \alpha_k(v)$ such that $t \Downarrow^\alpha w$ (in k-bounded CFA).*

Proof. The proof is by induction on the proof tree for $s \Downarrow v$, and cases on the last rule used. As this is straightforward we omit details and only deal with a few illustrative cases: constructor application, pattern matching and variable lookup.

We note that by Lemma 4.23, whenever $\mathbf{start} \rightarrow^* s$, $\mathrm{Valid}_{\mathrm{CA}}(s)$ necessarily holds.

Constructor Application Let $s = \mathsf{C}(e_1, \ldots, e_n) : \rho \Downarrow \mathsf{C}(v_1, \ldots, v_n) = v$, where for each i $s_i = e_i : \rho \Downarrow v_i$. Now $s \rightarrow s_i$ for each i, so the inductive hypothesis applies.

Let $t = \mathsf{C}(e_1, \ldots, e_n) : \mu$, where $\mu(x) \sqsupseteq \alpha_{k-1}(\rho(x))$ for each x. Then $t_i = e_i : \mu \sqsupseteq \alpha_k(s_i)$, so by the inductive hypothesis $t_i \Downarrow^\alpha w_i \sqsupseteq \alpha_k(v_i)$ for each i. Let $u_i = [w_i]_{k-1}$ for each i. By Lemma 4.26, $u_i \sqsupseteq \alpha_{k-1}(v_i)$ for each i.

Hence $w := \mathsf{C}(u_1, \ldots, u_n) \sqsupseteq \alpha_k(v)$, as required as $t \Downarrow^\alpha w$.

Pattern Matching Let $s = \textbf{match}\ e\ \textbf{with}\ \left\langle \mathsf{C}_i(x_1^i, \ldots, x_{n_i}^i) \to e_i \right\rangle_{i=1}^{k} : \rho = e_m : \rho \Downarrow v$ say. Then $t = e_m : \mu$, where $\mu(x) \sqsupseteq \alpha_{k-1}(\rho(x))$ for each x.

Now $s_m = e : \rho \Downarrow \mathsf{C}_l(v_1, \ldots, v_{n_l}) = c$ for some l, and $s' = e_l : \rho \oplus \{x_j^l \mapsto v_j\}_j \Downarrow v$. As $s \to s_m$ and $s \to s'$, the inductive hypothesis applies.

Let $t_m = e : \mu$. Then $t_m \Downarrow d \sqsupseteq \alpha_k(c)$ by the inductive hypothesis. There are two cases: either $t_m = *$ or $t_m = \mathsf{C}_l(w_1, \ldots, w_{n_l})$ where $w_j \sqsupseteq \alpha_{k-1}(v_j)$ for each j.

In the first case, $t' = e_l : \mu \oplus \{x_j^l \mapsto *\} \sqsupseteq \alpha_k(s')$ certainly, so by the inductive hypothesis $t' \Downarrow w \sqsupseteq \alpha_k(v)$, as required.

In the second case, $t' = e_l : \mu \oplus \{x_j^l \mapsto w_j\} \sqsupseteq \alpha_k(s')$, as $w_j \sqsupseteq \alpha_{k-1}(v_j)$ for each j. Thus we may apply the inductive hypothesis again to complete the proof.

Variable Lookup Let $s = x : \rho \Downarrow \rho(x) = v$. Then $t = x : \mu$, where in particular $\mu(x) \sqsupseteq \alpha_{k-1}(\rho(x))$. Therefore by Lemma 4.25 there exists w such that $\mu(x) \ll w$ and $w \sqsupseteq \alpha_k(\rho(x)) = \alpha_k(v)$. But then $t \Downarrow^\alpha w$, as required. □

Given Lemma 4.27, it is straightforward to conclude control-safety of k-bounded CFA, as in the 0CFA case. We merely state the result:

Corollary 4.28 (Control-Safety of k-bounded CFA)**.** *Let P be a program and **start** the start state of P. Suppose that **start** $\to^* s$, and let $t \sqsupseteq \alpha_k(s)$. Then if $s \Downarrow v$ (resp. $s \to s'$) there exists $t' \sqsupseteq \alpha(s')$ (resp. $w \sqsupseteq \alpha(v)$) such that $t \to^\alpha t'$ (resp. $t \Downarrow^\alpha w$) in the k-bounded CFA interpretation.*

4.5.4 A Worked Example

Let us now illustrate the k-bounded CFA analysis with an example. We shall use an artificial example to showcase what is gained by increasing precision in the approximation of environments. This illustrates the improvement of k-bounded CFA ($k > 0$) over 0CFA, which is increasingly visible as program size grows (for natural programs).

We recall the following example program from the previous section:

```
map f xs =
  match xs with
      [] -> []
    | z::zs -> f z :: map f zs

id x = x

k us = map id (1 :: us)

h vs = map k vs
```

Binding Construct	Parameter	Values
map	*f*	$(\!\|id : \emptyset\|\!)$ $(\!\|k : \emptyset\|\!)$
	xs	$*$
map.**match**	*z*	$*$
	zs	$*$
id	*x*	$*$
k	*us*	$*$
h	*vs*	$*$

Figure 4.9: Example Program: Closure Analysis

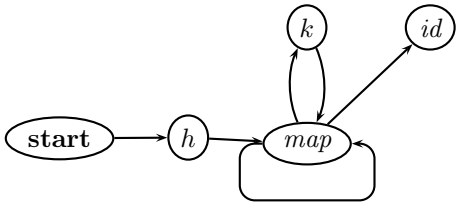
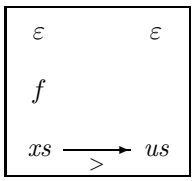


Figure 4.10: Example Program: 0CFA Call Graph

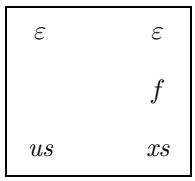
h <X>

We have claimed that this is not size-change terminating under 0CFA. Let us justify that claim. The results of closure analysis for this program are shown in Figure 4.9. From these it is a straightforward matter to deduce the 0CFA call graph of this program (Figure 4.10).

The interesting loop in the 0CFA call graph is the indirect $map \to k \to map$ loop. The size-change graphs for these calls are shown below:



$map \to k$



$k \to map$

The composition of these graphs is empty: no size-change information is available for the $k \to^+ k$ self-loop. As a result, this program is not 0CFA-terminating.

However, this program does terminate. The loop found in the call graph is in fact *spurious*: there is no corresponding sequence of calls in the dynamic call graph (for any input). Indeed, function k never (directly or indirectly) calls itself.

The loss of precision is due to the imprecision of 0CFA: there are two unrelated uses of *map* in this program, which are not differentiated, leading to the spurious self-loop.

In constrast, this program *is* size-change terminating under 1-bounded CFA. The 1-bounded CFA call graph is shown in Figure 4.11.

This call graph shows that the two occurrences of *map* are indeed disambiguated. The

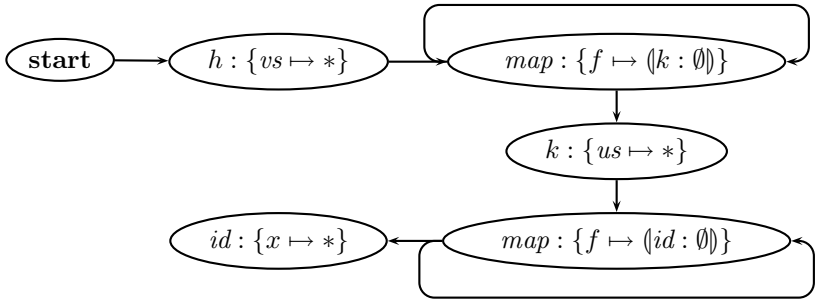


Figure 4.11: Example Program: 1-bounded CFA Call Graph

only self-loops in the call graph correspond to the recursion in the *map* function, which is naturally size-change terminating. The 1-bounded CFA call graph further reveals that the k function indeed cannot call itself recursively, as expected. It is frequently the case that spurious loops in the call graph result in a program being found nonterminating: no sensible size-change information can be found about loops that can never occur, and so no termination proof can be given.

4.5.5 Discussion

Cost

Let us first evaluate the size of the abstract domain for k-bounded CFA. Let V_k be the set of abstract values (abstract states which only represent values) and S_k the set of abstract proper states (all other states) at depth k.

As before, let $\mathcal{F}$ be the set of function definitions in P and $\mathcal{C}$ the set of constructors. Then:

$$\begin{aligned} |V_0| &= \sum_{f \in \mathcal{F}} |Params(f)| + |\mathcal{C}| + 1 + |AbstConst| \\ &\leq B \cdot |\mathcal{F}| + |\mathcal{C}| + 1 + |AbstConst| \end{aligned}$$

where B is one plus the maximum number of parameters of functions. The respective terms in this expression correspond to: closure values, constructor values, the special $*$ value and primitive constants.

The size of the set V_k of k-bounded CFA values can then be found as follows: a depth-k value is a tree with root labelled either with a closure, a constructor or a constant, and with

depth-$(k-1)$ values as subtrees. This gives:

$$\begin{aligned} |V_k| &= \sum_{f \in \mathcal{F}} \sum_{i=0}^{|Params(f)|} |V_{k-1}|^i + \sum_{\mathsf{C} \in \mathcal{C}} |V_{k-1}|^{\sharp \mathsf{C}} + 1 + |AbstConst| \\ &\leq \sum_{f \in \mathcal{F}} |V_{k-1}|^{\sharp f+1} + \sum_{\mathsf{C} \in \mathcal{C}} |V_{k-1}|^{\sharp \mathsf{C}} + 1 + |AbstConst| \\ &\leq (|\mathcal{F}| + |\mathcal{C}|) \cdot |V_{k-1}|^B + |AbstConst| + 1 \end{aligned}$$

Let $A = |V_0|$, $C = |\mathcal{F}| + |\mathcal{C}|$ and $D = |AbstConst| + 1$. As we have argued previously, in practice A is linear in the size of the program, while C is at most linear in the size of the program. It is reasonable to consider that D is a constant.

Then (an overestimate of) the size of V_k, say n_k, is given by:

$$\begin{aligned} n_0 &= A \\ n_{k+1} &= C n_k^B + D \end{aligned}$$

Let us estimate the asymptotic behaviour of n_k. First, $n_k \geq m_k$, where

$$m_0 = A \qquad \text{and } m_{k+1} = C m_k^B$$

That is,

$$n_k \geq A^{B^k} \cdot C^{1+B+\cdots+B^{k-1}}$$

On the other hand,

$$n_k = O\left((AC)^{B^k}\right)$$

This may be proved by considering the recurrence for $n_k/(AC)^{B^k}$.

It appears from this upper bound that the size of the state space for k-bounded CFA is high, and increases quickly (doubly exponentially) with k. Indeed, taking $B = 4$[7], we find that n_1 lies between CA^4 and $(AC)^4$, while n_2 lies between C^5A^{16} and $(AC)^{16}$ — a rather frightening complexity! To an extent this is confirmed in practice, in that that k-bounded CFA for $k > 2$ is rarely tractable.

However, we argue that the upper bound can only be reached in pathological cases, and indeed that the reachable state space of a program under k-bounded CFA is a fraction of the total state space. This is in contrast to 0CFA, where in a program with no dead code the entire state space is reachable.

As an example, consider a program which calls the *map* function n times say with distinct values $f_1, \ldots, f_n$ for parameter f. The only reachable states of the form $(\!|map : \{f \mapsto v, xs \mapsto w\}|\!)$ are those where v represents one of the functions f_1 through f_n. This is

[7]The average of four parameters for each function appears reasonable, and the precise value is of little import here, though λ-lifting may complicate the issue.

clearly likely (in non-pathological cases) to be much smaller than the set of functions in the program.

We may generalise this: the crucial point is that all reachable states s must satisfy the $\text{Valid}_{\text{CA}}(s)$ predicate. This is a substantial restriction on the size of the reachable state space in practice. We may therefore conclude that experimental evidence is indispensible to evaluate the cost of k-bounded CFA accurately.

k-Limited Environment Paths

We have defined the k-bounded CFA analysis to produce $(k+1)$-*limited size-change graphs*: size change graphs tracking dataflow between all substates lying at environment paths of length at most $k+1$. Safety of these derives from the fact that a depth-k state uniquely encodes environment paths up to depth $k+1$ (excluding the special $*$ state). For instance, the 1-bounded CFA state:

$$(\!|map : \{f \mapsto (\!|map : \{f\}|\!), xs \mapsto *\}|\!)$$

denotes all states (trees) whose root and immediate children are described by the given bindings. As a result, the graph basis of such a state necessarily contains the 2-limited basis $\{\varepsilon, \langle f \rangle, \langle f, f \rangle, \langle xs \rangle\}$.

An alternative strategy would be to produce 1-limited graphs, as in 0CFA. While this has no impact on the size of the state space (and thus the complexity of call graph construction), it *does* impact the size-change termination algorithm. Recall (Section 3.5) that this takes an annotated static call graph and checks the SCT condition. The complexity of this algorithm increases with the size of graph bases in the call graph, and therefore the choice of using $(k{+}1)$-limited graphs rather than less precise 1-limited graphs does impact the overall complexity.

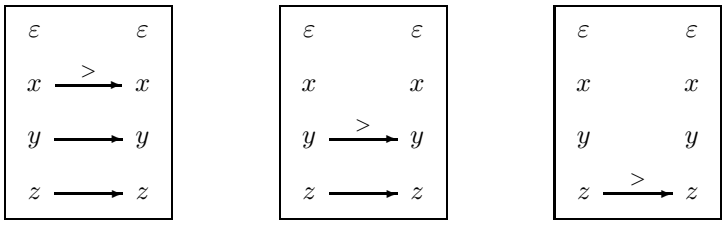
We must therefore justify this choice by showing that precision of the analysis is strictly increased. Consider first this simple example[8]:

```
hydra x y z =
  if x = 0 && y = 0 && z = 0 then 1 else
  if x > 0 then hydra (x-1) y z else
  if y > 0 then hydra (x+2) (y-1) z else
  hydra (x+2) (y+2) (z-1)
```

This is a first-order program, thus call graph construction is straightforward. The size-change

[8]The author is grateful to Richard Bird for suggesting the *hydra* example. This is an instance of the wider class of Hydra problems.

graphs produced by the analysis (0CFA) are as follows:



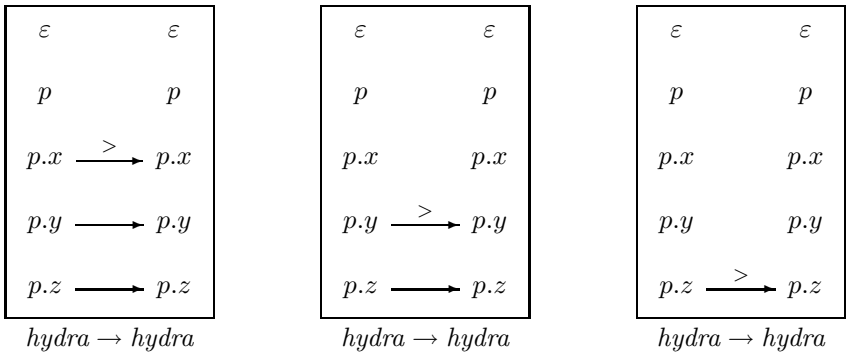
This program is size-change terminating — this can be checked either by arguing that any infinite sequence of calls leads to an infinitely decreasing thread, or by applying the SCT algorithm.

However, what of the *uncurried* version of this program?

```
hydra (x, y, z) =
  if x = 0 && y = 0 && z = 0 then 1 else
  if x > 0 then hydra (x-1, y, z) else
  if y > 0 then hydra (x+2, y-1, z) else
  hydra (x+2, y+2, z-1)
```

The notation in the above is really syntactic sugar for pattern matching the single parameter of *hydra* (say p) against a tuple pattern. Tuples are naturally defined as algebraic datatypes.

In 1-bounded CFA we may produce the following size-change graphs:



This is again size-change terminating, as before. However, the 1-limited graphs (excluding the $p.x$, $p.y$ and $p.z$ environment paths) do *not* admit this termination argument. In fact there is no size-change in the value of p (the only parameter at environment depth 1) in the recursive calls.

This example is therefore only found to terminate under 1-bounded CFA using 2-limited size-change graphs. It should be pointed out that uncurried functions represent an important special case, which an implementation of the analysis might well treat specially (for instance by currying functions prior to analysis). However, the above argument generalises to many

more situations (values within data structures such as lists, or indeed parameters of functions bound in the environment).

We can therefore conclude that general k-limited graphs do increase precision of our termination analysis by allowing size-changes to be tracked between values that occur deep in the environment, justifying this choice. Naturally an implementation might offer the choice of more precise, but potentially more expensive, k-limited graphs; or 1-limited graphs.

Environments and Stacks

The idea of our k-bounded CFA control flow analysis was to increase the precision of the approximation of environments. In k-bounded CFA, we record the environment (a tree) precisely up to a predetermined depth k. This seems to be a natural choice, given our development of the framework for computing the static call graph.

Furthermore, as we have remarked above, recording k levels of the environment allows dataflow to be tracked between values occurring at depth up to $k+1$ in the environment, increasing the set of programs found to terminate in the analysis.

Our notion of k-bounded CFA is related to the idea of k-CFA, introduced by Shivers [Shi91]. Rather than focus on approximating the *environments* of states, in k-CFA we approximate the values in the *call stack* upon reaching a particular point in the execution of the program.

In this interpretation, a state in the static call graph is a pair (s, p), where p is a program point and s is a representation of a set of call stacks. This allows us to distinguish different occurrences of the same function called from different locations — thus this is again a context-sensitive analysis. In k-CFA, we choose to represent the call stack by its top k elements. The representation of a state thus has type $PP^k \times AbsState_0$, where $AbsState_0$ is the set of 0CFA states and PP denotes the set of program points.

Let us illustrate this with an example. Consider the following program fragment:

```
f1 xs = map g1 xs

f2 xs = map g2 xs
```

where *g1* and *g2* are distinct functions. In 1-bounded CFA, this gives rise to two distinct states corresponding to the *map* function: one with the f parameter of *map* bound to *g1* and the other with the f parameter bound to *g2*.

In 1CFA, this likewise gives rise to two different *map* states, but these are differentiated by the topmost program point on the call stack: these states are of the form $(\langle f1 \rangle, map)$ and $(\langle f2 \rangle, map)$. These denote occurrences of *map* called from *f1* and *f2* respectively.

We shall not expand on k-CFA in detail, but wish to point out important features in the choice of k-bounded CFA or k-CFA. In our framework, k-bounded CFA is more natural: as we are tracking dataflow, the ability to consider values occurring deep in the environment

(in k-bounded CFA but not k-CFA) is valuable. Let us further give an example to show that our take on k-bounded CFA can increase precision (though at some cost in complexity).

We shall give an example program which is recognised as terminating under k-bounded CFA (indeed, 1-bounded CFA) but not under k-CFA (we shall show it is not terminating under 1CFA, but the argument applies to any k). In fact, our motivating example for 1-bounded CFA fits the bill. This example was described in Section 4.5.4, and we observed that this is found to terminate under 1-bounded CFA but not 0CFA, as the two occurrences of *map* must be separated — one calls the *id* function, the other calls k.

What is the behaviour of this program under 1CFA? There are two call sites for *map*, leading to two different states $s_h = (\langle h \rangle, map)$ and $s_k = (\langle k \rangle, map)$. However, the *map* function calls itself recursively, and thus both s_h and s_k call the state $s_m = (\langle map \rangle, map)$ corresponding to any instance of *map* called from within *map*.

As the value of parameter f in the state s_m can be either $(\!|k : \emptyset|\!)$ or $(\!|id : \emptyset|\!)$, state s_m must necessarily call both k and *id*. As a result, we may exhibit the following call sequence:

$$\textbf{start} \to (\langle\rangle, h) \to s_h \to s_m \to (\langle map \rangle, k) \to s_k \to s_m$$

This yields a spurious call from *map* (here, s_m) to itself *via* the function k. As in 0CFA, this prevents termination from being detected.

The key feature of this example is the use of recursion in *map* — as a result, no finite amount of information about the stack can help here, and k-CFA is unable to prove termination of this program for any k.

4.6 Unbounded Environments and Tree Automata

4.6.1 Motivation

Unbounded Environments

We have thus far described two flow analyses to support our termination check. First, the 0CFA analysis is a simple, context-insensitive analysis, with good efficiency but questionable precision. To increase precision we have introduced the k-bounded CFA analysis, in which information about environments is kept up to a fixed depth k.

We have argued that k-bounded CFA is a natural abstraction in many contexts. In particular, for programs that make heavy use of "simple" higher-order functions such as *map* or *fold*, k-bounded CFA (for low k) is a good way to disambiguate unrelated uses of these functions.

In general however, k-bounded CFA does not appear appropriate for arbitrary programs. In this section we shall essentially be concerned with improving precision in the analysis of programs that create environment trees of *unbounded* depth. Examples of such programs have been shown already (Church numerals and the higher-order *foldl* function), but in these

simple examples 0CFA was enough to show termination.

Given such a program, where (depending on the input) environment trees of arbitrary depths may be created, k-bounded CFA is no longer a natural way of improving precision. For, if trees of arbitrary depth are created, keeping any finite number of levels is unlikely to disambiguate enough states. Though k-bounded CFA ($k > 0$) can in fact succeed on such programs where 0CFA failed, this should be regarded as a (happy) coincidence. We shall be concerned with a more principled approach to dealing with such programs in this section.

4.6.2 Tree Automata

The analysis that we shall present in the remainder of this section is based on the use of *tree automata* to represent abstract states. Tree automata are a well-known finite representation of sets of trees, analogous to finite automata representing regular sets of strings. The justification for this approach is our observation that states correspond to finite trees — therefore abstract states represent sets of trees.

We shall first briefly introduce the basic definitions and facts we require in relation to tree automata, and subsequently show how these may be used for our purposes.

Background

Let us define tree automata formally. Note that there are several flavours of tree automata (top-down or bottom-up, deterministic or not), but we shall only be concerned with nondeterministic, top-down tree automata.

The alphabet of a tree automaton defines the set of node labels that may occur in a tree. Throughout this section we shall fix such an alphabet $\mathcal{F}$ say. Then $\mathcal{F}$ is a finite set of symbols, together with a map $\sharp : \mathcal{F} \to \mathbb{N}$. For each $a \in \mathcal{F}$, $\sharp a$ is the *arity* of a — the number of successors of a node with label a.

The set of trees over $\mathcal{F}$ is then defined as:

$$\mathrm{Trees}(\mathcal{F}) = \{a(t_1, \ldots, t_n) \mid a \in \mathcal{F} \wedge \sharp a = n \wedge (\forall i) t_i \in \mathrm{Trees}(\mathcal{F})\}$$

Furthermore, it is convenient to assume a set $\mathcal{V}$ of *variables*, disjoint from $\mathcal{F}$. We shall use φ, ψ, ... without comment to denote elements of $\mathcal{V}$.

Definition 4.29. A (top-down, nondeterministic) tree automaton is a quadruple $(Q, \mathcal{F}, I, \Delta)$ where: Q is a finite set of *states*, $\mathcal{F}$ is the alphabet, I is a set of *initial states* and Δ is a set of *rewrite rules* of the form

$$q/f(\varphi_1, \ldots, \varphi_n) \longrightarrow f(q_1/\varphi_1, \ldots, q_n/\varphi_n)$$

where q and the q_i are states, $f \in \mathcal{F}$ and $\sharp f = n$, and the φ_i are variables.

The set of trees recognised by such an automaton can be characterised as follows. Start with an initial state $q \in I$, and pick a transition at state q. The right-hand side of this transition defines the root node of the tree, and the states q_i of the automaton for each subtree. We can then proceed in this way repeatedly, expanding the tree. This process stops when we pick a rewrite rule introducing a symbol of arity 0 (no children).

Example 4.30. Consider the following automaton, over the alphabet $\{a, b, c\}$, where $\sharp a = 2$ and $\sharp b = \sharp c = 0$. The automaton has four states q_i $(1 \leq i \leq 4)$, where q_1 is the only initial state, and rewrite rules:

$$\begin{aligned} q_1/a(\varphi, \psi) &\rightarrow a(q_3/\varphi, q_2/\psi) \\ q_2/a(\varphi, \psi) &\rightarrow a(q_4/\varphi, q_1/\psi) \\ q_2/c &\rightarrow c \\ q_3/b &\rightarrow b \\ q_4/c &\rightarrow c \end{aligned}$$

Then a run of this automaton is shown below:

$$\begin{aligned} & q_1/a(\varphi, \psi) \\ \rightarrow\ & a(q_3/\varphi, q_2/\psi) \\ \rightarrow\ & a(b, a(q_4/\varphi, q_1/\psi)) \\ \rightarrow\ & a(b, a(c, a(q_3/\varphi, q_2/\psi))) \\ \rightarrow\ & a(b, a(c, a(b, c))) \end{aligned}$$

It is straightforward to define the set of trees produced by an automaton in general. Fix an automaton $(Q, \mathcal{F}, I, \Delta)$. The set of trees produced at state $q \in Q$ by this automaton is $\Phi(q)$, where:

$$\Phi(q) = \{f(t_1, \ldots, t_n) \mid (\exists q/f(\varphi_1, \ldots, \varphi_n) \rightarrow f(q_1/\varphi_1, \ldots, q_n/\varphi_n) \in \Delta) \wedge (\forall i) t_i \in \Phi(q_i)\}$$

The set of trees defined by the automaton is then the set of trees that can be produced from an initial state:

$$\Phi(\mathcal{A}) = \Phi(Q, \mathcal{F}, I, \Delta) = \bigcup_{q \in I} \Phi(q)$$

Tree Automata as Abstract States

We shall use such tree automata to describe abstract states. These will describe finite trees with nodes labelled by the usual cases: expressions, constructors, closures and constants. There are two points to note here. First, the alphabet for tree automata representing states contains *abstract constants*, rather than exact constants. This is necessary to ensure that it is possible to construct a finite abstract domain, but means that the trees produced by such

automata are not strictly speaking states. Second, it is necessary for technical reasons to include the *domain* of environments in closure nodes — so that closure node labels are of the form $(\!|f : S|\!)$, where $S \subseteq \mathit{Params}(f)$. This ensures that each label has a well-defined arity.

Rather than insisting on defining the sets of node labels and their arities precisely we shall be content to use this informally. We will also make use of the following shorthand, using notation similar to our existing notation for states within rewrite rules. For instance,

$$q/e : \{x \mapsto \varphi, y \mapsto \psi\} \to e : \{x \mapsto q_1/\varphi, y \mapsto q_2/\psi\}$$

The translation from this form to a *bona fide* rewrite rule could be formalised.

Example 4.31. We consider the Church Numerals example of Section 4.1 (p. 56). This program defines a function *toChurch* taking a natural number n and returning the n^{th} Church numeral. The interest lies in the fact that this program computes a state of depth which increases unboundedly with n, depicted in Figure 4.1 (p. 57).

We can define a tree automaton producing this set of values (precisely). Two states suffice, say q (the initial state) and r. The rewrite rules are:

$$\begin{aligned} q/(\!|\mathit{compose} : \{f \mapsto \varphi, g \mapsto \psi\}|\!) &\to (\!|\mathit{compose} : \{f \mapsto r/\varphi, g \mapsto q/\psi\}|\!) \\ q/(\!|\mathit{id} : \emptyset|\!) &\to (\!|\mathit{id} : \emptyset|\!) \\ r/(\!|f : \emptyset|\!) &\to (\!|f : \emptyset|\!) \end{aligned}$$

Given a tree automaton $\mathcal{A}$ as defined here, $\Phi(\mathcal{A})$ is a set of trees with nodes labelled with expressions, closures, constructors and abstract constants. This is intented to represent any concrete state (tree) in which exact constants replace abstract constants.

To make this precise, let $\multimap$ be the relation defined by:

1. $c \multimap d$ whenever $c \in \mathit{AbstConst}$, $d \in \mathit{PrimConst}$ and $d \in \gamma(c)$, and
2. $\multimap$ is extended pointwise to trees. For instance, $\mathsf{C}(v_1, \ldots, v_n) \multimap \mathsf{C}(w_1, \ldots, w_n)$ iff $v_i \multimap w_i$ for each i.

That is, $t \multimap s$ iff s can be obtained from t by replacing abstract constants by concrete constants.

We shall let $\Xi(\mathcal{A}) = \{t \mid (\exists s \in \Phi(\mathcal{A}))s \multimap t\}$ denote the set of concrete states represented by $\mathcal{A}$.

We shall make a simplifying assumption on tree automata used to represent states in the sequel. We shall require that *all* trees $t \in \Xi(\mathcal{A})$ share the *same* root node label. Note that this does not hold of the automaton $\mathcal{A}$ shown above — the root labels of trees in $\Xi(\mathcal{A})$ can be either $(\!|\mathit{compose} : \{f, g\}|\!)$ or $(\!|\mathit{id} : \emptyset|\!)$. This would therefore have to be represented as a *set* of two tree automata to cover all cases with this restriction.

The upshot of requiring this uniqueness property is that we may define $\zeta(\mathcal{A})$ to be the root label of any state described by $\mathcal{A}$. This allows us to define the calls originating at an abstract state by pattern matching on $\zeta(\mathcal{A})$ as before, simplifying subsequent developments.

4.6.3 Operations on States

We shall now introduce the principles of our control-flow analysis using tree automata. The key step is to identify the operations on states and values that are used in the semantics (Chapter 2), so that these may be approximated in the abstract interpretation. We shall focus on constructions that affect the environments of states, as these are most relevant to the construction of the static call graph.

Binding and Retrieving Values The most fundamental operations on environments are: binding a value to a variable in the environment, and retrieving the value of a variable.

Overloading previous notation, we shall write *Bind* s x v for the state obtained from s by binding the value v to x in the environment of s. Similarly, we write *Lookup* s x for the value bound to parameter x in state s.

A few remarks apply here. First, both the *Bind* and *Lookup* operations are partial. Certainly *Lookup* s x is only defined if x is bound in the environment of s. Furthermore, if $s = (\!|f : \rho|\!)$ say, then *Bind* s x v is only defined if x is the first parameter of f not bound in ρ. For that reason, *Bind* s x v requires s to be a proper *value*, not a state that may be evaluated further.

We may give these operations the following types:

$$\begin{aligned} \mathit{Bind} &: \mathit{Value} \to \mathit{VarName} \to \mathit{Value} \to \mathit{State} \\ \mathit{Lookup} &: \mathit{State} \to \mathit{VarName} \to \mathit{Value} \end{aligned}$$

As *Bind* s x v is only ever defined for (at most) one variable x, we may define (as before) a shorthand *Bind* s v, leaving out the variable. It is sometimes convenient to make the variable name explicit, however.

Subject to the constraints outlined above, the fundamental property of these operations is:

$$\mathit{Lookup}\ (\mathit{Bind}\ s\ x\ v)\ y = \begin{cases} v & \text{if } x = y \\ \mathit{Lookup}\ s\ y & \text{otherwise} \end{cases}$$

Constructing Algebraic Values A further operation on environments which cannot directly be expressed in terms of the operations defined above is the construction of an algebraic value (constructor application): given values $v_1, \ldots, v_n$, construct the value $\mathsf{C}(v_1, \ldots, v_n)$.

It is clear that this is similar to the *Bind* operation defined previously, but requires binding

n values *simultaneously*, as constructors cannot be partially applied. The resulting type is:

$$Build : Constr \rightarrow Value\ list \rightarrow Value$$

Of course, *Build* C *vs* is only defined when $|vs| = \sharp\mathsf{C}$.

Changing Program Point The last operation on states is rather straightforward: changing the current program point without affecting the environment. For instance, evaluating a function application $s = e_1 e_2 : \rho$ requires evaluation of $s_1 = e_1 : \rho$. This can be modelled as follows:

$$Replace : State \rightarrow Expr \rightarrow State$$

To wit, *Replace* $s\ e^l$ is the state obtained from s by keeping the environment unchanged, and setting the program point to l.

Abstract Interpretation

We have now identified the operations on environments that are relevant to the semantics. An approximation to the semantics, based on tree automata, can now be obtained by giving approximations to these constructions. To this end, we require abstract equivalents of the previously defined operations. The types of abstract operations are given below:

$$\begin{aligned} Bind^\alpha &: AbsVal \rightarrow VarName \rightarrow AbsVal \rightarrow AbsState \\ Lookup^\alpha &: AbsState \rightarrow VarName \rightarrow AbsVal\ set \\ Build^\alpha &: Constr \rightarrow AbsVal\ list \rightarrow AbsVal \\ Replace^\alpha &: AbsState \rightarrow Expr \rightarrow AbsState \end{aligned}$$

It should be noted that $Lookup^\alpha$ returns a *set* of abstract values (that is, a member of the *lifted* domain of abstract interpretation as defined in Section 4.3), whereas all other operations return individual abstract values. The reason for this is our restriction that the root node of an abstract value, $\zeta(v)$, be well-defined. The result of retrieving the value of a parameter (say the f parameter of *map*) may be a set of states representing different functions. This cannot be represented by a single automaton with the single-root property. On the other hand, all other operations return a set of values with the single-root property, so that a single automaton may be returned.

The abstract operations should be *safe*, in the expected sense — these should provide a conservative approximation to the exact operations. More precisely,

1. Whenever $s \in \Xi(t)$ and $v \in \Xi(w)$, $Bind\ s\ x\ v \in \Xi(Bind^\alpha\ t\ x\ w)$.
2. Whenever $s \in \Xi(t)$, $Lookup\ s\ x \in \Xi(w)$ for some $w \in Lookup^\alpha\ t\ x$.
3. Whenever $v_i \in \Xi(w_i)$ for each i, $Build\ \mathsf{C}\ \langle v_1, \ldots, v_n \rangle \in \Xi(Build^\alpha\ \mathsf{C}\ \langle w_1, \ldots, w_n \rangle)$

4. Whenever $s \in \Xi(t)$, $\mathit{Replace}\ s\ e \in \Xi(\mathit{Replace}^\alpha\ t\ e)$

Given such safe operations we may derive the static call graph construction rules. This is fairly straightforward, and analogous to the static call graph constructions we have outlined for 0CFA and k-CFA. As a result we shall be content with giving two illustrative examples: evaluating variables and function applications. The rule for variables is as follows:

$$\frac{v \in \mathit{Lookup}^\alpha\ s\ x}{s \Downarrow v, \overline{1[\varepsilon/x]}}\ \zeta(s)=x$$

as expected, though notationally cluttered. Likewise, the rule for function applications is similar to the exact semantics rule, but using the abstract constructions:

$$\frac{\mathit{Replace}^\alpha\ s\ e_1 \Downarrow s_b, \gamma_1 \quad \mathit{Replace}^\alpha\ s\ e_2 \Downarrow w, \gamma_2 \quad \mathit{Bind}^\alpha\ s\ w \Downarrow v, \delta}{s \Downarrow v, [\mathrm{app}_x(\gamma_1, \gamma_2)]_1 ; \delta}\ \zeta(s)=e_1 e_2$$

The remaining semantic rules can be dealt with in a similar fashion. Note that the size-change graphs produced alongside these evaluations are the *1-limited graphs* — in general this is the best that can be achieved given an arbitrary tree automata-based abstract domain. The specific abstract domain that we shall describe does not admit k-limited graph bases for any $k > 1$, so that there is no loss of generality here.

It is trivial to derive the control safety and dataflow safety properties, given the safety properties of the abstract operations on states. We shall thus again omit proofs and state the result:

Proposition 4.32. *Whenever Lookup^α, Bind^α, Build^α and $\mathit{Replace}^\alpha$ are safe, the resulting analysis is safe.*

Safety of these basic operations is therefore all that we need to achieve in order to produce an appropriate, sound analysis. In the next section we shall focus on how this may be obtained for a specific choice of abstract domain.

4.6.4 Static Analysis

The framework that we have defined is extremely general: we use tree automata to represent sets of states, subject to the single-root property (each automata represents a set of states which share the same root node). Provided the abstract operations on states are defined suitably, this yields a safe analysis.

However, this does not in itself tell us how to derive a computable approximation. In fact a trivial instantiation is to represent *singleton* sets of states as tree automata (any single tree is trivially an automaton), leading to the exact semantics again — achieving nothing.

In this section we shall be concerned with picking a *finite* domain of tree automata for a computable analysis. That is, we must define a finite set *AbsState* of automata, and the

corresponding operations on such automata within this set. The key is to impose restrictions on the form of such automata and the number of states in each automaton.

It should be pointed out that deriving an appropriate abstract domain given the operations that we have specified above can be seen as an instance of the more general problem of abstract interpretation for term graph rewriting systems [GH93], or indeed for term rewriting systems. In particular, techniques used in *abstract reduction* [Nöc90, vEGHN93] are echoed in our setting. Abstract reduction was introduced as a basis for strictness analysis, and proceeds by defining an abstract analogue of a concrete reduction system that operates on *sets* of term graphs. As in our setting, the key is determining when it is appropriate to approximate operations, and thus ensure termination of the algorithm. Again similarly to our formulation, the abstract domain is infinite, consisting of sets of term graphs. However, a difference between the two approaches is that in abstract reduction termination is obtained by losing precision when approximating certain concrete operations, via such operations as *cycle-introduction* (introducing a cyclic term rather than expanding infinitely) and *top-introduction* (replacing a term by $\top$, the worst possible approximation, to ensure termination). In contrast, we choose an *a priori* finite subset of the set of tree automata, and thus convergence of the algorithm is automatic, at some cost to precision.

Choosing an Abstract Domain

Our abstract domain is inspired by the following procedure. Given a tree representing a state, we may *collapse* this tree to a (possibly cyclic) directed graph by identifying nodes with the same label, and redirecting edges accordingly. If this graph is expanded into a set of trees, starting at root label of the original tree, then we must surely recover the original tree.

The graph obtained from collapsing a tree in this fashion has at most one node with each possible label. Given a finite set of node labels, there are thus finitely many such graphs. We may therefore collapse a *set* of trees to a graph by collapsing each of the individual trees and forming the union of the graphs. Again, we may be assured that the original set of trees is approximated by expanding the graph. A simple illustration of this process is shown in Figure 4.12, using our Church numerals example.

Given this intuition, we can define the condition on our automata precisely. As we are guided by the idea that there should only be one occurrence of each node label, we shall require that the states of the automata correspond to node labels: there is at most one state for each node label. Furthermore, the root of a tree produced at state s_l, corresponding to label l, must itself carry the label l. Finally, the single-root condition is satisfied provided we ensure that the automaton only has one initial state.

Definition 4.33. A tree automaton $\mathcal{A} = (Q, \mathcal{F}, I, \Delta)$ has the *single-label property* if:

1. $Q \subseteq \{s_m \mid m \in \mathcal{F}\}$

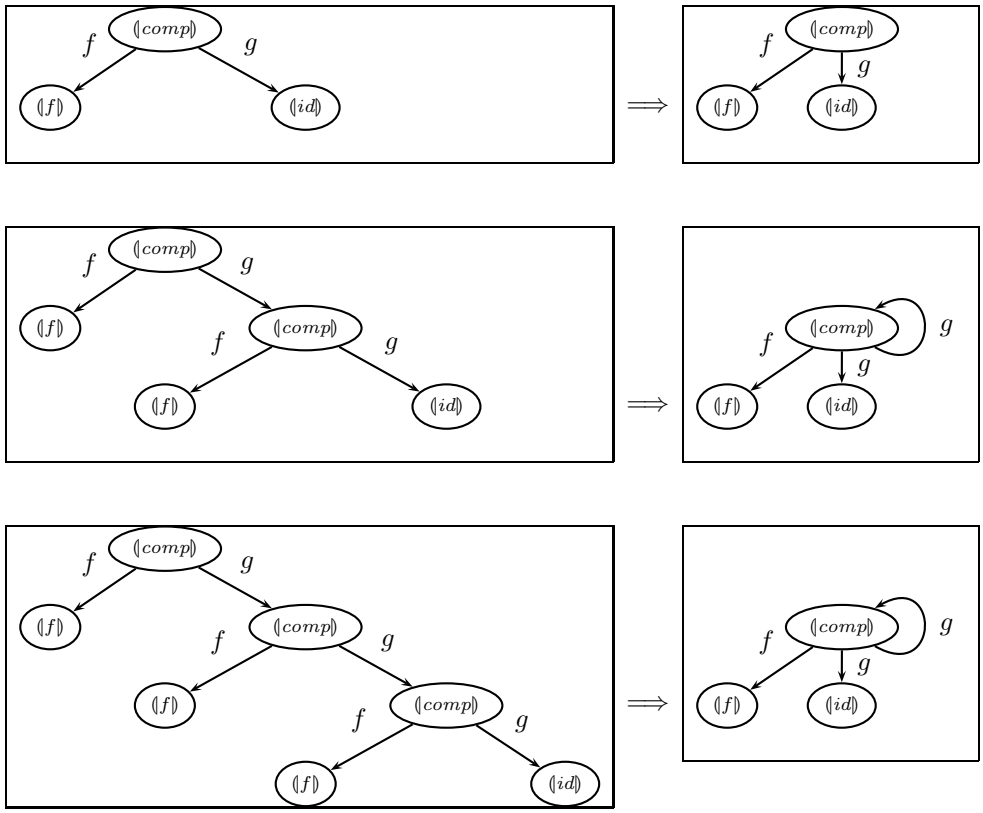


Figure 4.12: Collapsing a set of trees to a directed graph

2. $I = \{s_m\}$ for some node label $m \in \mathcal{F}$
3. Each rule in Δ is of the form:

$$s_m/m(x_1, \ldots, x_n) \to m(s_{m_1}/x_1, \ldots, s_{m_n}/x_n)$$

That is, the root of a tree produced at state s_m is necessarily m.

We shall define our domain of abstract states to be the set of automata with the single-label property. The required properties are easily verified:

Lemma 4.34. *For any fixed (finite) alphabet $\mathcal{F}$, the set of automata with the single-label property is finite. Furthermore each such automaton has the single-root property.*

The connection between automata with the single-label property and the graphs obtained by collapsing is intuitively clear. However it should be noted that the automata formulation is in fact strictly more expressive. This is due to the fact that automata can record *dependencies* between subtrees. This is illustrated below:

Example 4.35. Consider the following automaton, over alphabet $\{a, b, c\}$:

$$\begin{aligned} s_a/a(x,y) &\to a(s_b/x, s_b/y) \\ s_a/a(x,y) &\to a(s_c/x, s_c/y) \\ s_b/b &\to b \\ s_c/c &\to c \end{aligned}$$

Then the set of trees produced by this automaton is just $\{a(b,b), a(c,c)\}$. Furthermore, this automaton has the single-label property.

However, the graph obtained by collapsing the two trees $a(b,b)$ and $a(c,c)$ does not record the dependency between the children of a (*i.e.* that either both children are b or both are c). As a result, this graph expands to the *four* trees $a(b,b)$, $a(b,c)$, $a(c,b)$ and $a(c,c)$.

In practice this property is difficult to exploit fully in the analysis, though it has the potential to increase precision. This is due to the fact that the $Bind^\alpha$ operation only binds one parameter at a time — while this is crucial to deal with partially applied functions, we lose the opportunity to record dependencies between variables.

Operations on Automata

The purpose of this section is to introduce the operations on abstract states represented as tree automata with the single-label property, and to show that these give rise to safe approximations.

Union A fundamental operation on automata is taking the *union* of productions. It should be stressed that this does not give rise to the union of the sets of trees produced — indeed such an operation would not give rise to an automaton with the single-root property, let alone the single-label property.

Definition 4.36. Let $\mathcal{A} = (Q_A, \mathcal{F}, \{s^A\}, \Delta_A)$ and $\mathcal{B} = (Q_B, \mathcal{F}, \{s^B\}, \Delta_B)$ be automata with the single-label property. Then their *union* is the automaton $\mathcal{A} \oplus \mathcal{B} = (Q_A \cup Q_b, \mathcal{F}, \{s^A\}, \Delta_A \cup \Delta_B)$.

This operation is not symmetrical: the start state of the resulting automaton is that of the first automaton, ignoring the second. To remind ourselves of this we avoid using the $\cup$ operator to denote this operation.

Union preserves the single-label property, trivially:

Lemma 4.37. *Let $\mathcal{A}$ and $\mathcal{B}$ have the single-label property. Then $\mathcal{A} \oplus \mathcal{B}$ has the single-label property.*

The crucial property of union that we shall rely upon is the following:

Lemma 4.38. *Let $\mathcal{A}$ and $\mathcal{B}$ be automata with the single-label property, and let $\mathcal{C} = \mathcal{A} \oplus \mathcal{B}$. Then for each state q, the set $\Phi(\mathcal{C}, q)$ of trees produced by $\mathcal{C}$ at state q is a superset of $\Phi(\mathcal{A}, q) \cup \Phi(\mathcal{B}, q)$.*

Proof. We show that for any tree $t \in \Phi(\mathcal{A}, q) \cup \Phi(\mathcal{B}, q)$, it is the case that $t \in \Phi(\mathcal{C}, q)$ by induction on the depth of t.

As $t \in \Phi(\mathcal{A}, q) \cup \Phi(\mathcal{B}, q)$, $t = a(t_1, \ldots, t_n)$, where there is a rule $q/a(x_1, \ldots, x_n) \to a(q_1/x_1, \ldots, q_n/x_n)$ in one of $\mathcal{A}$ or $\mathcal{B}$ (call this $\mathcal{D}$), with $t_i \in \Phi(\mathcal{D}, q_i)$ for each i. As the depth of t_i is less than the depth of t for each i, by the inductive hypothesis $t_i \in \Phi(\mathcal{C}, q_i)$ for each i. As the rule $q/a(x_1, \ldots, x_n) \to a(q_1/x_1, \ldots, q_n/x_n)$ lies in $\mathcal{D}$ it is likewise in $\mathcal{C}$, yielding that $t = a(t_1, \ldots, t_n) \in \Phi(\mathcal{C}, q)$ as required. □

It does not follow that the set of trees produced by $\mathcal{A} \oplus \mathcal{B}$ is (a superset of) the union of the sets of trees produced by $\mathcal{A}$ and $\mathcal{B}$, due to the asymmetry in choosing the start state.

Successor Automata The next operation on tree automata that we shall be concerned with is to produce automata to describe the possible *children* of the root node of trees produced by the automaton.

Let us first introduce some notation for concisely describing state transitions in automata representing states.

Definition 4.39. Let $\mathcal{A} = (Q, \mathcal{F}, \{s_m\}, \Delta)$ be a single-label tree automaton, where $\mathcal{F}$ is the alphabet of state node labels. Then we write $\mathcal{A} : q \xrightarrow{x^i} q^i$ if one of the following transitions

lies in Δ:

$$\begin{array}{rcl} q/e : \{x^1 \mapsto \varphi^1, \ldots, x^n \mapsto \varphi^n\} & \to & e : \{x^1 \mapsto q^1/\varphi^1, \ldots, x^n \mapsto q^n/\varphi^n\} \\ q/(\!|f : \{x^1 \mapsto \varphi^1, \ldots, x^n \mapsto \varphi^n\}|\!) & \to & (\!|f : \{x^1 \mapsto q^1/\varphi^1, \ldots, x^n \mapsto q^n/\varphi^n\}|\!) \\ q/\mathsf{C}(\varphi^1, \ldots, \varphi^n) & \to & \mathsf{C}(q^1/\varphi^1, \ldots, q^n/\varphi^n) \end{array}$$

This definition encodes the idea that from state q, retrieving the value of x can lead to state q'. This is formalised below:

Definition 4.40. Let $\mathcal{A}$ be as above, and suppose that $\mathcal{A} : s_m \xrightarrow{x} q$. We define the automaton $\mathcal{A}[q] = (Q, \mathcal{F}, \{q\}, \Delta)$, trivially a single-label automaton.

Lemma 4.41. *Let $\mathcal{A}$ be as above, and suppose that $t \in \Phi(\mathcal{A})$, with x bound in the environment of t. Then if t' is the x child of t, $t' \in \Phi(\mathcal{A}[q])$ for some q with $\mathcal{A} : s_m \xrightarrow{x} q$.*

Proof. We assume that $t = e : \rho$, as other cases are similar. Then there is a rule in $\mathcal{A}$ of the form $s_m/e : \{x^i \mapsto \varphi^i\}_i \to e : \{x^i \mapsto q^i/\varphi^i\}_i$, where say $x = x^j$. Put $q = q^j$. Then necessarily $\rho(x) \in \Phi(\mathcal{A}, q) = \Phi(\mathcal{A}[q])$, as required. □

State Operations We are now able to define the state operations on single-label tree automata.

Let us first define *Lookup*$^\alpha$. To retrieve a value, x say in the environment, it suffices to follow x productions from the root of the automaton.

Definition 4.42. Let $\mathcal{A} = (Q, \mathcal{F}, \{s_m\}, \Delta)$ be a single-label tree automaton. Then define

$$\mathit{Lookup}^\alpha\ \mathcal{A}\ x = \{\mathcal{A}[q] \mid \mathcal{A} : s_m \xrightarrow{x} q\}$$

Lemma 4.43. *The Lookup$^\alpha$ operation is safe.*

Proof. This is immediate from Lemma 4.41. □

Defining the *Replace*$^\alpha$ operation is likewise straightforward.

Definition 4.44. Let $\mathcal{A} = (Q, \mathcal{F}, \{s_m\}, \Delta)$ be a single-label tree automaton, and suppose that $e^l \in \mathit{Expr}$ is such that $\mathrm{Scope}(e^l)$ is equal to the set of children of m. Then define $\mathit{Replace}^\alpha\ \mathcal{A}\ e := \mathcal{A}^e = (Q, \mathcal{F}, \{s_e\}, \Delta \cup X)$, where X is the set:

$$\left\{ s_e/e : \{x \mapsto \varphi_x\}_x \to e : \{x \mapsto s^x/\varphi_x\}_x \mid s_m/m(\{x \mapsto \varphi_x\}_x) \to m(\{x \mapsto s^x/\varphi_x\}_x) \in \Delta \right\}$$

In the above definition we take the notation $m(\{x \mapsto \varphi_x\}_x)$ to include both expression nodes $m : \{x \mapsto \varphi_x\}_x$ and closure nodes $(\!|m : \{x \mapsto \varphi_x\}_x|\!)$.

Lemma 4.45. *The Replace$^\alpha$ operation is safe.*

Proof. Let $t \in \Xi(\mathcal{A})$. We assume wlog that $t = e : \rho$. Then let $t' = \mathit{Replace}\ t\ e' = e' : \rho$. Then as $t \in \Xi(\mathcal{A})$, there is a production $s_e/e : \{x \mapsto \varphi^x\}_x \to e : \{x \mapsto s^x/\varphi^x\}_x$, such that for each $x \in \operatorname{dom}\rho$, if $t_x = \rho(x)$ then $t_x \in \Xi(\mathcal{A}, s^x)$. Let $\mathcal{A}' = \mathcal{A}^{e'}$. Then as the set of productions of $\mathcal{A}'$ is a superset of that of $\mathcal{A}$, for each s^x, $\Xi(\mathcal{A}', s^x) \supseteq \Xi(\mathcal{A}, s^x) \ni t_x$. As there is further a rule $s_{e'}/e' : \{x \mapsto \varphi^x\}_x \to e' : \{x \mapsto s^x/\varphi^x\}_x$ in $\mathcal{A}'$, with $s_{e'}$ the initial state of $\mathcal{A}'$ we conclude that $t' \in \Xi(\mathcal{A}')$, as required. □

Let us now turn to the definition of the Bind^α operation. This is conceptually a two-stage operation: to construct $\mathit{Bind}^\alpha\ s\ x\ v$, we should:

1. Construction the union of s and v, and
2. Add a new root node to the resulting automaton, binding x to the root of v.

We may define this formally as follows:

Definition 4.46. Let $\mathcal{A} = (Q_A, \mathcal{F}, \{s_m\}, \Delta_A)$ and $\mathcal{B} = (Q_B, \mathcal{F}, \{s_n\}, \Delta_B)$ be single-label automata, where $m = (\!|f : S|\!)$ and $S \subsetneq \mathit{Params}(f)$. Suppose further that x is the first parameter of f not in S. We define

$$\mathit{Bind}^\alpha\ \mathcal{A}\ x\ \mathcal{B} = \mathcal{C} \oplus (\mathcal{A} \oplus \mathcal{B})$$

where $\mathcal{C}$ is the automaton with starting state $s_{m'}$, where $m' = (\!|f : S \cup \{x\}|\!)$, and productions of the form:

$$s_{m'}/(\!|f : \{y \mapsto \varphi^y\}_{y \in S} \cup \{x \mapsto \varphi^x\}|\!) \to (\!|f : \{y \mapsto s^y/\varphi^y\}_{y \in S} \cup \{x \mapsto s_n/\varphi^x\}|\!)$$

for each production in $\mathcal{A}$ of the form:

$$s_m/(\!|f : \{y \mapsto \varphi^y\}_{y \in S}|\!) \to (\!|f : \{y \mapsto s^y/\varphi^y\}_{y \in S}|\!)$$

It is straightforward that the resulting automaton possesses the single-label property (as the $\oplus$ operation preserves the single-label property). Furthermore, this is a safe approximation:

Lemma 4.47. *The Bind^α construction is safe.*

Proof. Let $s \in \Xi(\mathcal{A})$ and $v \in \Xi(\mathcal{B})$. Then $s = (\!|f : \rho|\!)$, where $\operatorname{dom}\rho = S$ and $v_x := \rho(x) \in \Xi(\mathcal{A}, s^x)$, where there is a production $s_m/(\!|f : \{y \mapsto \varphi^y\}_{y \in S}|\!) \to (\!|f : \{y \mapsto s^y/\varphi^y\}_{y \in S}|\!)$ in $\mathcal{A}$. Furthermore, $v \in \Xi(\mathcal{B}, s_n)$, where s_n is the start state of $\mathcal{B}$.

Let $\mathcal{C} = \mathit{Bind}^\alpha\ \mathcal{A}\ x\ \mathcal{B}$. Then by Lemma 4.38, for each x, $v_x \in \Xi(\mathcal{C}, s^x)$, while $v \in \Xi(\mathcal{C}, s_n)$. Hence as there is a production

$$s_{m'}/(\!|f : \{y \mapsto \varphi^y\}_{y \in S} \cup \{x \mapsto \varphi^x\}|\!) \to (\!|f : \{y \mapsto s^y/\varphi^y\}_{y \in S} \cup \{x \mapsto s_n/\varphi^x\}|\!)$$

in $\mathcal{C}$, the state $s' = (\!|f : \rho \oplus \{x \mapsto v\}|\!)$ lies in $\Xi(\mathcal{C})$, as required as $s' = \mathit{Bind}\ s\ x\ v$. □

To complete our description of abstract operations, it remains to define the constructor application operation $Build^\alpha$. As expected, this is essentially equivalent to $Bind^\alpha$, with the exception that several variables are bound simultaneously.

Definition 4.48. Let $\mathcal{A}_i = (Q_i, \mathcal{F}, \{s^i\}, \Delta_i)$ for $i = 1, \ldots, n$ be single-label automata. We define $\mathcal{B} = Build^\alpha\ \mathsf{C}\ \langle \mathcal{A}_1, \ldots, \mathcal{A}_n \rangle$ to be the automaton $\mathcal{C} \oplus (\mathcal{A}_1 \oplus \cdots \oplus \mathcal{A}_n)$, where $\mathcal{C}$ is the automaton with start state label C and the single production:

$$s_{\mathsf{C}}/\mathsf{C}(\varphi^1, \ldots, \varphi^n) \to \mathsf{C}(s^1/\varphi^1, \ldots, s^n/\varphi^n)$$

Lemma 4.49. *The $Build^\alpha$ operation is safe.*

Proof. This is similar to the $Bind^\alpha$ operation. Given values $v_i \in \Xi(\mathcal{A}_i)$, $v_i \in \Xi(\mathcal{B}, s^i)$ for each i. Hence $Build\ \mathsf{C}\ \langle v_1, \ldots, v_n \rangle = \mathsf{C}(v_1, \ldots, v_n) \in \Xi(\mathcal{B})$, as required. □

Equivalence, Order and Reachability

We have completed our description of the use of single-label tree automata as abstract states, and the operations on these values. To obtain a precise description of the abstract domain it remains to define the abstraction function $\alpha : State \to AbsState$, and to define the order $\sqsubseteq$ on the set $AbsState$ of abstract states.

We have in fact already defined the *concretisation* function $\gamma : AbsState \to \mathbb{P}(State)$. Indeed, this is just the Ξ function defined previously. Furthermore, there is a natural notion of order arising from this definition:

$$\mathcal{A} \sqsubseteq \mathcal{B} \qquad \Longleftrightarrow \qquad \gamma(\mathcal{A}) \subseteq \gamma(\mathcal{B})$$

Furthermore, we may define $\alpha : State \to \mathbb{P}(AbsState)$ by using the process of collapsing nodes described previously (and illustrated in Figure 4.12). More formally, we may define this as follows:

1. *Primitive Constants.* The automaton corresponding to the primitive constant c is just the automaton representing the single tree $\alpha^C(c)$.

2. *Constructor Values.* The automaton for $\mathsf{C}(v_1, \ldots, v_n)$ may be defined recursively as $Build^\alpha\ \mathsf{C}\ \langle \alpha(v_1), \ldots, \alpha(v_n) \rangle$.

3. *Closure States.* Consider a closure state of the form $(\!|f : \rho|\!)$. Let $w_x = \alpha(\rho(x))$ for each $x \in S := \operatorname{dom} \rho$. Then $\alpha(x) = \mathcal{C} \oplus (\oplus_{x \in S}\, w_x)$, where $\mathcal{C}$ is the automaton with root $m = (\!|f : S|\!)$ and single production $s_m/(\!|f : \{x \mapsto \varphi^x\}_x|\!) \to (\!|f : \{x \mapsto s^x/\varphi^x\}_x|\!)$, where s^x is the start state of w_x for each x.

4. *Expression States.* The case of expression states is formally identical to that of closure states.

It is now straightforward to verify the following property, by analogy with the safety proofs for operations on states:

Lemma 4.50. *Whenever s is a state, $s \in \gamma(\alpha(s))$. Furthemore, $\gamma(s) = \{t \mid \alpha(t) \sqsubseteq s\}$*

That is, the γ operation is in accordance with the definition of Section 4.3, so that $\alpha(t)$ is indeed the most defined state representing t.

There is however a difficulty that we must face: the $\sqsubseteq$ order, as defined, is in fact only a *preorder* in general. That is, there are automata $\mathcal{A} \neq \mathcal{B}$ such that $\mathcal{A} \sqsubseteq \mathcal{B}$ and $\mathcal{B} \sqsubseteq \mathcal{A}$. In such a case $\gamma(\mathcal{A}) = \gamma(\mathcal{B})$ — these automata represent the same set of trees, but can be differentiated.

The cause of this is apparent by considering the $Lookup^{\alpha}$ operation. We have defined this to keep the set of productions of the automaton intact, while changing the start state. This may make certain productions *unreachable*. As unreachable productions do not affect the set of trees produced, this creates distinct states producing the same trees.

This problem may be avoided by requiring that automata have *no* unreachable productions, and pruning unreachable productions at every stage. This is enough to guarantee that γ is a proper partial order:

Lemma 4.51. *Suppose that $\mathcal{A}$ and $\mathcal{B}$ are automata with no unreachable productions, over the same alphabet, and that $\gamma(\mathcal{A}) = \gamma(\mathcal{B})$. Then $\mathcal{A} = \mathcal{B}$.*

Proof. The proof is by contradiction. Suppose that $\mathcal{A} \neq \mathcal{B}$. Then there is a production (wlog) in $\mathcal{A}$ but not in $\mathcal{B}$, say $s_m/m(x_1, \ldots, x_n) \to m(s^{m_1}/x_1, \ldots, s^{m_n}/x_n)$. This production is reachable, so that there is a run of the automaton producing an intermediate tree t with a variable y at state m. This gives rise (assuming that there are no states with no productions, which our construction guarantees) to a tree $s \in \Phi(\mathcal{A})$ with a subtree (in the position of y in t) of the form $m(m_1(s_1), \ldots, m_n(s_n))$. Such a subtree can only arise from the production shown above. Thus no tree produced by $\mathcal{B}$ can contain such a subtree, a contradiction as required. □

We shall leave the details of adding unreachable production elimination to the reader's imagination, concluding this section.

4.6.5 Discussion

Representing Tree Automata

Having described the domain of single-label tree automata to represent abstract states, and defined the necessary operations on these values, we need to find a good representation that allows these operations to be carried out efficiently. The basic components of operations on automata are:

1. Union (used to implement the $Bind^{\alpha}$ operation and others)

2. Eliminating unreachable states (in particular, this is necessary after a $Lookup^{\alpha}$ operation)

3. Comparing the languages of two automata (both to compute the $\sqsubseteq$ ordering and to check automata for equivalence)

The implementation of these operations is greatly simplified if we make the simplifying assumption, already evoked earlier, that dependencies between function parameters can be ignored. There are two main reasons for this choice:

1. It is unclear whether this can be used effectively in the analysis to actually improve precision, and

2. It does not appear that the precision gains are significant for "natural" programs.

Consequently, we shall ignore such dependencies in the analysis. Given this, the representation of states is straightforward: a state simply corresponds to a (possibly cyclic) graph as shown for example in Figure 4.12. More precisely, we may represent an abstract state as a pair of relations representing the graph, and the root node of the state:

$$AbsState = \mathbb{P}(Node \times VarName \times Node) \times Node$$

where *Node* is the set of node labels (expressions, constants, constructors and closure labels). The edges of the graph are labelled with variable names.

Such graphs represent abstract states, subject to two conditions:

1. All nodes are reachable from the start node (we use unreachable state elimination after each graph operation to ensure this), and

2. Any node n with parameter set S (recall that each node uniquely identifies the set of parameters it expects) has at least one outgoing edge labelled x for each $x \in S$.

These conditions are preserved by operations on states.

The operations on such states are rather simple. The automata union operation is just union of the corresponding relations. Furthermore, eliminating unreachable states reduces to finding the set of reachable nodes of the graph (*i.e.* computing the transitive closure of the relation). Finally, comparing the languages of graphs (that is, the sets of trees that these graphs expand to) can be achieved by comparing the corresponding relations (in the subset order). This last fact is non-obvious, and we shall prove this below:

Lemma 4.52. *Let $\mathcal{A} = (R, n)$ and $\mathcal{B} = (S, n)$ be automata with the same root node, and suppose further that R and S contain no nodes that are not reachable from n. Then if X and Y are the sets of trees obtained by expanding $\mathcal{A}$ and $\mathcal{B}$ (resp.), then $X \subseteq Y$ iff $R \subseteq S$.*

Proof. It is trivial that if $R \subseteq S$, then $X \subseteq Y$. The other direction is of more interest. Suppose that $R \not\subseteq S$. Then there is an edge $(m, x, p) \in R \setminus S$. We note that the condition that each node should have at least one edge for each variable it expects in its environment guarantees that the set of trees obtained by expanding the graph at each node p (say, $\Phi(R, p)$) is nonempty.

Hence there is a tree t in $\Phi(R, p)$, with root node p necessarily. We can thus construct a tree $q \in \Phi(R, m)$ such that the x successor of m is p. As m is reachable, there is a tree $q' \in \Phi(R, n)$ with q as a subtree.

However, no tree in $\Phi(S, n)$ can have an x edge from a node labelled m to a node labelled p, as there is no such edge in S. Hence $q' \notin \Phi(S, n)$, and $X \not\subseteq Y$, as required. □

The simplicity of this representation, and the doubtful use of keeping track of dependencies between function parameters, make it natural to pick this as our representation. Many efficient representations of such labelled relations are known, and in particular *binary decision diagrams* emerge as an appropriate representation as we may expect that many reachable states are closely related (so that the sharing offered by binary decision diagrams is effective in reducing complexity).

Arbitrary Values

We recall that in 0CFA and k-bounded CFA we had introduced a special value $*$ to deal with the problem of *arbitrary* values: unknown values assigned to program template variables. In these analyses this special case was crucial: unknown values cannot be approximated by closure analysis, and thus soundness demands a specific representation.

In contrast, the $*$ value is not *a priori* necessary in the tree automata analysis. For, a tree automaton directly represents a set of states, without constraints of validity with respect to closure analysis. We are therefore free to assign any abstract state t to a template variable X, with the guarantee that the resulting analysis is sound whenever the concrete value s assigned to X satisfies $s \in \gamma(t)$.

Arbitrary Values and Precision However, a practical matter makes it useful to in fact reintroduce this value. Suppose that we would like to prove that *map f xs* terminates for *any* list *xs* (where f is a fixed function say). To achieve this we must find a (set of) abstract states S such that $v \in \gamma(S)$ whenever v is a list. This can be defined as a set of two states, one representing nil, and the other (t say) representing any cons-list.

In particular, this state t must represent all possible lists of functions. As a result this must approximate *all* possible function values (including those that cannot be constructed in the program). The result of this is that the only safe approximation is to have an edge $(\!|h : S|\!) \xrightarrow{x} n$ for each function h, $x \in S$ and node n — encoding the fact that we have no information about the parameters of such functions.

What can we say of the state corresponding to evaluation of *map* with *xs* bound to t? Such a state is a union of several states, one of which is t, and is thus a superset of t. In particular, this state contains edges $(\!|map : \{f, xs\}|\!) \xrightarrow{f} n$ for *all* possible nodes n.

Let us take a step back. In analysing *map f xs*, with *xs* bound to t and f bound to a fixed function, we reach a state in which f can be bound to *any* value — obliterating precision of the call graph.

The cause of this is the interplay between elements of the list (which may be arbitrary values) and the value bound to f, under the $\oplus$ union operation.

A Solution A solution to this problem is to reintroduce the $*$ value, as a possible node label, to encode arbitrary values. As $*$ is distinct from all other node labels, it can never be merged with other nodes in the $\oplus$ operation. This guarantees that the precision problem outlined above cannot arise.

It is a straightforward extension to the analysis to include this special node, and is similar to the similar treatment of $*$ in previous analyses. We shall therefore not elaborate further on this matter.

Of course, other solutions to this problem are possible. In particular, we could refine the abstract domain to ensure that function nodes occurring in different contexts cannot be merged (for instance, a function occurring in a list and one bound to a function parameter need not be merged). However, these would increase the complexity of the analysis, as well as making its description much more involved. The $*$ solution has the advantage of simplicity, and does not substantially increase the size of the abstract domain.

Cost

To conclude this section, let us briefly evaluate the cost of the single-label tree automata analysis by estimating the size of the abstract domain. As we have previously remarked in the context of k-bounded CFA, this is not necessarily a good indication of the analysis' behaviour in practice. For, the set of reachable states is a gross overapproximation to the set of states that are in fact reached when analysing a given natural program.

It is easy to give an upper bound on the size of the abstract domain: the set *Node* of node labels is just the set of 0CFA states. The size of this set, N say, has been argued to be linear in the size of the program (for realistic programs, at any rate). The size of the set $\mathbb{P}(\mathit{Node} \times \mathit{VarName} \times \mathit{Node}) \times \mathit{Node}$ representing abstract states is thus $2^{BN^2}N$, where B is the maximum number of parameters of any function. In general we expect the maximal complexity of this analysis to be exponential in the (square of the) size of the program. However, whether or not this is tractable in practice depends on several factors:

1. How many states are in fact produced in the analysis: as argued before there are many constraints on these states, in particular reachability of nodes in the graph and constructibility within the program (closure analysis validity)

2. The order (subset) relation on states: we only need to record *maximal* states in the call graph

3. The efficiency of binary decision diagrams in representing large numbers of such relations.

4.7 References

The analysis of Jones and Bohr [JB04] corresponds to 0CFA; we introduce k-bounded CFA and tree automata based variants of the size-change termination analysis. The size-change element is shared between such analyses by the use of the dataflow graphs introduced in Chapter 3. We have introduced k-limited size-change graphs, which serve to strictly increase precision. This has not been introduced even for SCT analysis of term rewriting systems (where an equivalent definition could be made).

The 0CFA analysis of higher-order languages is due to Shivers [Shi91, Shi88], and has been widely used as the standard control-flow analysis for functional languages. The fundamental tool in 0CFA is closure analysis, approximating the set of values that a function parameter might take. However, the style of semantics used as a starting point affects the presentation of the analysis and resulting information. Shivers uses a (non-standard, abstract) denotational semantics to define 0CFA. Malacaria and Hankin have shown that 0CFA analysis of PCF can be derived as an abstraction of a game semantics for PCF [MH98]. Jagannathan and Weeks [JW95] have showed that 0CFA analysis is equivalent to set-based analysis [Hei94].

Our k-bounded CFA analysis is inspired from the k-CFA analysis presented by Shivers [Shi91] as a tool to improve precision of the call graph construction. However, Shivers' definition of k-CFA differs from our own — where we approximate the environment of states up to a fixed bound, Shivers approximates the call stack. Our approach appears to increase precision at the cost of a larger state space. In addition, in our setting the environment approach is more natural and allows the use of $(k+1)$-limited size-change graphs, improving precision.

Other variants of k-CFA have been proposed, and a modern treatment is presented by Nielson, Nielson and Hankin [NNH99]. It should be observed that a similarly-named, but qualitatively different, analysis has been denoted as *k-limited CFA*. For instance, Malacaria and Hankin derive this analysis for PCF from a game semantics point of view [MH98]. In k-limited CFA, the aim is to find call sites at which it is known that the set of functions that may be called is small. Thus k-limited CFA is a weaker analysis than 0CFA (as this information may be obtained from the results of 0CFA).

Cousot and Cousot [CC94, CC91] define an abstract interpretation framework for higher-order functions, based on the approximation of function values (by binary relations). This focuses on the *extensional* approximation of functions, however (as input-output pairs), and so is not suited to our purposes. The same applies to the method of Burn, Hankin and

Abramsky [BHA86] for strictness analysis of higher-order functions. Furthermore in our untyped context we cannot rely on a finite depth to types, as is required in these methods.

The tree automata based control-flow analysis that we have presented is new, and seeks to address the poor precision of k-bounded CFA analyses for programs with closures of unbounded depth. A related analysis is the flow analysis of Jones [Jon87], in which regular tree grammars are used to describe reachable program states in lazy higher-order functional programs. Cousot and Cousot describe the analysis of Jones as an abstract interpretation with a (program dependent) finite abstract domain of grammars [CC95]. Gallagher and Puebla [GP02] describe an analysis of logic programs to approximate the *values* of parameters as tree automata.

Chapter 5

Analysing Lazy Programs

5.1 Termination and Lazy Programs

5.1.1 Introduction

We have thus far described the size-change termination analysis for *strict* functional languages, using the language defined in Chapter 2 as an intermediate representation. We have presented several static analyses based on different abstractions of the set of states appearing in the execution of a program.

In this chapter, we shall show how the size-change termination analysis can be made to apply to the analysis of *lazy* functional programs. Full details will not be given, as many of the steps toward the SCT analysis of lazy programs are similar to corresponding work in previous chapters on strict programs.

We will first describe the specificities of the termination analysis problem for lazy programs in more detail in the remainder of this section. We shall then present the SCT analysis for lazy languages. This may be derived directly from the call-by-name semantics we define in this section, mirroring the developments in Chapters 2, 3 and 4. However, it proves crucial to introduce a more refined order on values to deal with laziness. Finally, we shall discuss alternative approaches briefly and illustrate the analysis of lazy programs by an example. Throughout this chapter, example programs will be shown in a syntax similar to the Haskell [PJ03] language.

5.1.2 Laziness and Termination

Call-by-Name and Lazy Evaluation

Cyclic Structures The aim of this chapter is to extend our termination analysis to *lazy* languages such as Haskell. However, it is inconvenient for our purposes to deal with laziness explicitly. The main difficulty is that lazy programs may create *cyclic* structures, which (pure)

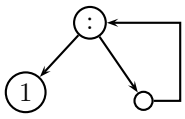


Figure 5.1: Lazy Evaluation: Cyclic Structures

strict functional programs cannot. For instance, consider the following program fragment:

```
ones = 1 : ones
```

This program defines *ones* to be the infinite list of ones. Under lazy evaluation, a use of *ones* causes the structure in Figure 5.1 to be created: a *cyclic* list of ones (so that evaluation of this list is possible in constant space).

The ability to create cyclic structures derives from the fact that in lazy evaluation, multiple instances of the same variable are *shared* to avoid costly recomputations. In particular, in the evaluation of *ones*, the instance of *ones* on the right-hand side of the definition points to the same node as the instance of *ones* being defined.

While the sharing of nodes is crucial to an efficient implementation of non-strict programming languages, it is inconvenient in our setting. For, the SCT method relies on a *well-founded* order on values. We have thus far been able to use the subtree order on representations of values, since values are indeed represented as trees (in particular, *acyclic* graphs) in call-by-value.

In the presence of cyclic structures, the direct equivalent of the subtree order is no longer well-founded. This comes as no surprise: for instance, we were able to show that in a list $x :: xs$, the tail xs of the list was smaller than the whole list. It is clear that this no longer holds in the presence of *infinite* lists such as *ones*. However, defining a well-founded order on possibly cyclic graphs that mimics the subtree order does not seem feasible.

Call-by-Name Fortunately, we shall be able to sidestep this issue entirely. We shall henceforth abandon *laziness* and instead focus on *call-by-name* evaluation. This is entirely valid: lazy evaluation is really no more than an efficient implementation technique for call-by-name evaluation. The difference lies in the sharing of instances: in call-by-name, every occurrence of a variable will be evaluated separately as needed. For instance, consider the program

```
f x = x + x

eight = f (2×2)
```

Evaluation of *eight* will cause the body of f to be evaluated with variable x bound to the (unevaluated) expression $2 * 2$. In lazy evaluation, this will be evaluated once and the result shared for each occurrence of x. In call-by-name, however, the value of each occurrence of x will be computed independently.

Call-by-name is substantially simpler to deal with than lazy evaluation. In particular, it is impossible to create cyclic structures in call-by-name, just as it is in call-by-value. The infinite list *ones* unfolds to an infinite tree, but at any point only finitely many levels of this tree are expanded. Any state in the execution of the program is therefore a *bona fide* finite tree, and the existing orders may be reused.

It is safe and precise to ignore lazy evaluation in favour of call-by-name: a program terminates under lazy evaluation iff it terminates under call-by-name. In the remainder of this chapter we will not make the distinction between laziness and call-by-name, and will use the more seductive terminology of laziness in lieu of call-by-name.

Termination of Lazy Programs

Let us now turn to some of the specific issues with termination analysis of lazy programs. The purpose of this discussion of this section is to answer the question: is a specific termination analysis for lazy programs necessary? It is widely known that lazy evaluation is an optimal evaluation strategy, in the sense that if a program terminates under any evaluation strategy, then it terminates under lazy evaluation. It is therefore the case that *if* we find a program terminating using the existing SCT analysis for call-by-value languages, *then* it is also guaranteed to terminate under lazy evaluation.

In this section we will give examples of features of lazy programs that prevent us from using this strategy — of course, it is sound, but much too imprecise to be useful in practice.

Infinite Structures The first issue is the use of *infinite structures* in lazy programs. Naturally, a lazy program that actually causes evaluation of an infinite structure is nonterminating, but many terminating programs use only a finite part of an otherwise infinite structure. A simple example is shown below:

```
zip xs [] = []
zip [] (y:ys) = []
zip (x:xs) (y:ys) = (x,y) : zip xs ys

infinite i = i : infinite (i+1)

number xs = zip (infinite 0) xs
```

The *zip* function is a standard Haskell function, while *infinite* computes the infinite list $[i, i+1, i+2, \ldots]$. The *number* function takes a list *xs* and returns a list with the same elements, such that each element is paired with its position in the list.

The *number* function is terminating on any finite input list under lazy evaluation (equivalently, call-by-name), but not call-by-value. In call-by-value, evaluation of *number xs* forces full evaluation of *infinite* 0, which is nonterminating. Establishing termination of this program under lazy evaluation therefore requires a new approach.

Recursive Value Definitions The next difficulty is a feature of lazy languages that is entirely absent from strict languages. While the previous example program was nonterminating as a strict program, it could nonetheless be interpreted as one. However, the simple program for creating an infinite list of ones has *no* equivalent as a strict program:

```
ones = 1 : ones
```

In the above, the value *ones* is defined in terms of itself. While lazy languages allow arbitrary values to be defined recursively in this way, strict languages typically only allow *functions* to be defined recursively[1]. For, a definition such as the above cannot be assigned a sensible meaning in a strict language: such a definition would require evaluation of the right-hand side, which conversely requires that the value of *ones* be defined.

It is necessary to deal with such definitions explicitly in an analysis of lazy programs. Values such as *ones* can certainly be used in terminating programs — in particular, *head ones* is terminating.

Circular Programs Finally, let us consider the case of the so-called *circular* functional programs. While these do not strictly speaking use any features that we have not described above, it is interesting to study these and the termination problems that arise.

We shall introduce circular programs by way of illustration, with the famous *repmin* program by Richard Bird [Bir84]. Let us define a type of leaf-labelled binary trees:

```
data Tree α = Tip α | Fork (Tree α) (Tree α)
```

We wish to define a function

```
repmin : Ord α ⇒ Tree α → Tree α
```

which finds the *minimum* value that appears in the tree, and returns a tree of the same shape as the original tree, but with all leaf values replaced by the minimum value. It is trivial to write a program that achieves this in two phases: first find the minimum, and then generate a new tree. The *repmin* program achieves this in one pass by using laziness. We shall define a function

```
rpm : Ord α ⇒ α → Tree α → (Tree α, α)
```

that takes a value x say and a tree, and returns both the tree obtained by replacing all leaves of the tree with x and the minimum of the leaf values of the original tree. It is straightforward to define *rpm*:

```
rpm x (Tip y) = (Tip x, y)
rpm x (Fork t1 t2) = (Fork t1' t2', min m1 m2)
                     where (t1', m1) = rpm x t1
                           (t2', m2) = rpm x t2
```

[1] OCaml is not entirely limited to this, and modest extensions are made in the implementation, though these are not part of the language standard

The *repmin* program can now be completed by *tying the knot*: the *rpm* function achieves what is required provided the value x that is passed to *rpm* is the minimum of the original tree, which is returned as a result by *rpm*:

```
repmin t = t'
           where (t', minval) = rpm minval t
```

This program is *circular* because one of the outputs of *rpm* is used as an input to that function. Why is this program correct? Is it easy to convince oneself that by the fixpoint semantics of recursive definitions, if the program terminates it computes the right tree. Termination of this program is highly nontrivial, however. It relies on the fact that the value of the first parameter x of *rpm* is *not* required to compute the second element of the result — so that when the value of x is required, this forces evaluation of *minval* in *repmin*, which does not depend on itself.

Termination of circular programs is therefore difficult to establish even by hand. We aim to be able to show termination of programs such as *repmin* using the techniques of this chapter. It is plain that this requires an analysis that is aware of laziness (not least because *repmin* uses a recursive value definition, which is not allowed in call-by-value).

5.1.3 Termination Properties of Lazy Programs

As the previous examples have shown, achieving a precise enough termination analysis for lazy programs requires us to treat laziness (equivalently, call-by-name evaluation) specially, rather than rely on our existing analysis for strict programs.

However, we must also address the question of what is meant by termination of a lazy program. The evaluation mechanism of lazy evaluation allows us to have a more fine-grained view of termination, and thus we briefly describe the salient features of lazy evaluation in order to highlight this.

Evaluation to Weak Head Normal Form The essence of call-by-name, or indeed lazy evaluation, is that states are not fully evaluated as they are in call-by-value. In our call-by-value semantics, the judgement $s \Downarrow v$ states that v is the *normal form* of s. That is, v has one of the following forms:

1. A primitive constant, or
2. A partially applied function f, where all applied arguments are fully evaluated, or
3. A constructed value $\mathsf{C}(v_1, \ldots, v_n)$, where the v_i are fully evaluated.

Our values therefore represent fully evaluated states — no further reduction is possible without inspecting function bodies for possible reductions.

In contrast, in lazy evaluation we only require evaluation to *weak head normal form* (WHNF). A value is in weak head normal form if it is not itself a redex (in our setting,

redexes correspond to expressions, as our expressions do not include λ-abstractions). A values is in weak head normal form if it is one of the following:

1. A primitive constant, or
2. A partially applied function f, where applied arguments need not be fully evaluated, or
3. A constructed value $\mathsf{C}(s_1, \ldots, s_n)$, where the s_i need not be fully evaluated.

For example, a list value is in WHNF if its top-level constructor is evaluated: it is either [] or $s_1 :: s_2$, but we need know nothing about s_1 or s_2, and indeed evaluation of either of these could be nonterminating.

Driving Evaluation of Lazy Programs Consider the lazy functional program:

```
infinity = 1 :: infinity

infinity
```

Evaluation of this program in an interpreter will produce, as expected, an infinite list of ones. However, evaluation of the expression *infinity* to WHNF merely produces the following evaluation steps:

$$\mathit{infinity} \quad \rightarrow \quad 1 :: \mathit{infinity} \quad \Downarrow \quad ::(1, \mathit{infinity})$$

The expression $::(1, \mathit{infinity})$ is in WHNF: it is a constructor applied to two states. This suggests that we should only observe these two reductions, and indeed that the interpreter would terminate, if our claim that lazy evaluation means evaluation to WHNF is to hold.

The apparent paradox is explained by the driving mechanism of interpreters for lazy languages: the top-level evaluator does not stop at WHNF, but rather keeps reducing the state further, printing the value as it is produced. More precisely, consider a hypothetical interpreter for a lazy language in which the result of a computation is always a list of characters[2]. The evaluator then proceeds as described by the following pseudocode:

```
eval s =
  reduce s to WHNF v
  case v of
      [] → exit
    | s :: s' →
      reduce s to WHNF c
      print c
      eval s'
```

[2]For instance, in Haskell the *show* function is invoked to transform a value to a string of characters, which can be printed.

The list is therefore evaluated to WHNF; its first character evaluated and printed; and the tail of the list is then evaluated again. This can be seen as a hybrid between reduction to WHNF and normal form: the list is not evaluated to normal form immediately (this would be nonterminating if the list was infinite), but rather repeatedly evaluated to WHNF and printed lazily.

Termination and Laziness The evaluation mechanism for lazy programs therefore yields two immediate possibilities for defining termination of a lazy program:

1. A program terminates if its evaluation to WHNF terminates, or
2. A program terminates if the repeated evaluation process terminates.

Both of these carry intuitive appeal: the first identifies termination with finite evaluation, as in the call-by-value case; while the second more closely mirrors the operational aspect of lazy interpreters.

The two definitions of termination are indeed different: the *infinity* program is terminating in the first sense but not the second. Of course, any program which terminates in the second sense likewise terminates in the first sense.

In the remainder of this chapter we shall ensure that we can analyse programs for termination in either of these meanings. In practice we may be more interested in the second form of termination, however, as the first is too permissive: we do not usually want to proclaim *infinity* terminating!

Termination properties of lazy functional program have been extensively studied, leading in particular to the notion of *evaluators*, due to Burn [Bur87]. Burn defines four evaluators for lazy programs, to account for the varying degrees to which an expression should be evaluated, based on context. The four evaluators defined by Burn are:

1. ξ_{NO} is the trivial evaluator that performs no evaluation,
2. ξ_1 evaluates an expression to WHNF,
3. ξ_2 evaluates the *structure* of an algebraic datatype value, but not its elements, and
4. ξ_3 evaluates an expression to normal form.

Identifying these evaluators allows the mode of evaluation to be varied according to context. For instance, the argument of a non-strict function should be evaluated using ξ_{NO}, while the argument of a strict function may be evaluated using ξ_1 (or indeed in some cases ξ_2 or ξ_3). This leads to the notion of *evaluation transformers* [Bur87]: for each function, given the evaluator used at an application site (defining the context), an evaluator for each function argument may be deduced through static analysis. This generalises strictness analysis to structured datatypes. Burn's evaluators are derived from Wadler's four-point abstract domain [Wad87]. Specifically, evaluators are defined by the abstract domain elements for

which they are nonterminating. The ξ_1 evaluator is thus nonterminating on $\perp$ only, ξ_2 is nonterminating for all partial or infinite datatype values (represented by the ∞ element in Wadler's presentation), while ξ_3 is nonterminating for all finite or infinite datatype values containing an undefined element (represented by $\perp\in$). The final element $\top\in$ of Wadler's four-point domain yields the trivial evaluator that never terminates.

Burn's ξ_1 and ξ_3 evaluators correspond closely to the two modes of termination that we have already identified — evaluation to WHNF and evaluation to normal form (giving termination of the evaluation loop). The ξ_2 evaluator defines a mode of evaluation in which the spine of a list is evaluated, but not any of its arguments, and similarly for other datatypes. The archetypal example of a function requiring evaluation only up to ξ_2 for its argument is the *length* function, which ignores the elements of its input list. We will not aim to analyse programs for ξ_2-termination in this chapter.

For completeness, let us describe a refinement of these evaluation modes. Consider the following program:

```
bottom = bottom

badinfinity = bottom :: badinfinity

badinfinity
```

There is an intuitive sense in which *infinity* is "more terminating" than *badinfinity*: executing *infinity* will cause the evaluator to print ones forever, while executing *badinfinity* is nonterminating but produces no output.

We say a program is *productive* if each evaluation to WHNF in the interpreter loop is terminating. Productiveness corresponds to the idea that the interpreter loop may be nonterminating due to the fact that the result of the program is an infinite structure, rather than a misbehaviour of the program.

Productiveness does not fit in Burn's evaluator model directly, as it is necessary to evaluate the spine of a list in order to evaluate its arguments, and so this requires the ξ_3 evaluator. Productiveness may only be distinguished because the top-level evaluator prints evaluation results lazily. The notion of productiveness thus does not appear to be applicable to intermediate results in a computation, unlike Burn's evaluators and evaluation transformers.

Determining productiveness of the evaluation of a program does not require us to introduce a separate analysis. If, as in Haskell, we only consider the case of programs that return lists of primitive constants (in particular, characters), then productiveness can be reduced to termination of evaluation to WHNF. The following function retrieves the n^{th} element of a list:

```
at (x:xs) 0 = x
at (x:xs) (n+1) = at xs n
```

A program with start expression e is then productive iff the same program with start expression

```
at  e <X>
```

is terminating for all values of X.

5.2 Semantics for Laziness and Termination Analysis

Having made our aims more precise we shall now develop the SCT analysis for lazy (call-by-name) programs. Our treatment mirrors that of Chapters 2 to 4 for the analysis of call-by-value programs. As we have stated previously, few ideas in the development of this analysis are new, and thus details are omitted. The exception is the order on values, which must be refined to handle nontrivial lazy programs.

5.2.1 Call-by-Name Semantics

Let us first define the semantics for our call-by-name language. The syntax of the language is almost identical to that of our call-by-value core language. The only difference is that in addition to function definitions $f\ x_1\ \cdots\ x_n = e$ $(n > 0)$ we shall allow possibly recursive value definitions $x = e$ (corresponding to the case $n = 0$).

Values and States

An important difference between the call-by-name semantics and their call-by-value equivalents is the notion of *value*. In Chapter 2 we defined a value to be a *normal form*: a constant; or a constructor or closure together with an environment binding variables to *values*.

As we have introduced in the previous section, however, in call-by-name values need not be in normal form. This is due to the fact that bindings of variables (either function parameters or constructor parameters) need not be fully evaluated. We have introduced the concept of *weak head normal form* to capture this.

Let us make this more precise. We can define the sets of values, states and environments as follows (using notation from Section 2.2):

$$
\begin{aligned}
\mathit{ConstValue} &= \mathit{PrimConst} \cup \{\mathsf{C}(s_1, \ldots, s_n) \mid \mathsf{C} \in \mathit{Constr} \wedge n = \sharp\mathsf{C} \wedge (\forall i) s_i \in \mathit{State}\} \\
\mathit{Env} &= \mathit{VarName} \mathrel{+\!\!\!\rightarrow}^{\text{fin}} \mathit{State} \\
\mathit{Closure} &= \{(\!|f : \rho|\!) \mid f \in \mathit{FunctionName} \cup \mathit{PrimOp} \wedge \rho \in \mathit{Env} \wedge \operatorname{dom}\rho \subseteq \mathit{Params}(f)\} \\
\mathit{Closure}' &= \{(\!|f : \rho|\!) \in \mathit{Closure} \mid \operatorname{dom}\rho \subsetneq \mathit{Params}(f)\} \\
\mathit{Value} &= \mathit{ConstValue} \cup \mathit{Closure}' \\
\mathit{State} &= \mathit{ConstValue} \cup \mathit{Closure} \cup \{e : \rho \mid e \in \mathit{Expr} \wedge \rho \in \mathit{Env} \wedge \operatorname{dom}\rho \supseteq \operatorname{fv}(e)\}
\end{aligned}
$$

Values, Variables and Constants

$$\frac{}{v \Downarrow v}\ {\scriptstyle v \in Value} \qquad \frac{\rho(x) \Downarrow v}{x : \rho \Downarrow v} \qquad \frac{}{c : \rho \Downarrow c}$$

Function References and Closures

$$\frac{(\!|f : \emptyset|\!) \Downarrow v}{f : \rho \Downarrow v} \qquad \frac{Body(f) : \rho \Downarrow v}{(\!|f : \rho|\!) \Downarrow v}\ {\scriptstyle \mathrm{dom}\,\rho = Params(f)}$$

Primitive Operators

$$\frac{(\forall i) e_i : \rho \Downarrow c_i \quad Apply\ op\ \langle c_1, \ldots, c_n \rangle = c}{op(e_1, \ldots, e_n) : \rho \Downarrow c}\ {\scriptstyle (\forall i) c_i \in PrimConst,\ c \notin Error}$$

Constructors and Pattern Matching

$$\frac{}{\mathsf{C}(e_1, \ldots, e_n) : \rho \Downarrow \mathsf{C}(s_1, \ldots, s_n)}\ {\scriptstyle (\forall i) s_i = e_i : \rho}$$

$$\frac{e : \rho \Downarrow \mathsf{C}_l(s_1, \ldots, s_{n_l}) \quad e_l : \rho \oplus \{x_j^l \mapsto s_j\}_{j=1}^{n_l} \Downarrow v}{\textbf{match}\ e\ \textbf{with}\ \left\langle \mathsf{C}_i(x_1^i, \ldots, x_{n_i}^i) \to e_i \right\rangle_{i=1}^{k} : \rho \Downarrow v}$$

Conditionals

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

Function Application

$$\frac{e_1 : \rho \Downarrow (\!|f : \mu|\!) \quad Bind\ (\!|f : \mu|\!)\ (e_2 : \rho) \Downarrow v}{e_1 e_2 : \rho \Downarrow v}\ {\scriptstyle (\!|f:\mu|\!) \in Closure'}$$

Figure 5.2: Call-by-Name Semantics for the Core Language

These definitions are essentially identical to the respective definitions for call-by-value, with the only difference lying in the fact that the elements bound in environments, as well as those bound to parameters of constructors, may now be arbitrary (unevaluated) states instead of being restricted to values. This corresponds to the difference between normal form and weak head normal form.

Semantics

We may now define the semantics of evaluation. This is given by the judgement $s \Downarrow v$, where $s \in \mathit{State}$ and $v \in \mathit{Value}$ as before. The meaning of this judgement is different, however — $s \Downarrow v$ now means that the weak head normal form of s is v. The $\Downarrow$ relation is defined in Figure 5.2.

It is worth highlighting the semantic rules that differ between the call-by-name and call-by-value semantics:

- A reference $f : \rho$ to a global definition triggers evaluation of $(\!|f : \emptyset|\!)$ — this need no longer be a value, as f might have no parameters (a recursive value definition).

$$\frac{e : \rho \Downarrow w \quad w^{\blacktriangledown} \Downarrow v}{\textbf{force}\ e : \rho \Downarrow v}$$

Driving Evaluation

Eager Evaluation

$$\frac{}{c^{\blacktriangledown} \Downarrow c}\ {\scriptstyle c \in PrimConst} \qquad \frac{}{v^{\blacktriangledown} \Downarrow v}\ {\scriptstyle v \in Closure}$$

$$\frac{(\forall i) \quad s_i \Downarrow w_i \quad {w_i}^{\blacktriangledown} \Downarrow v_i}{\mathsf{C}(s_1, \ldots, s_n)^{\blacktriangledown} \Downarrow \mathsf{C}(v_1, \ldots, v_n)}$$

Figure 5.3: Semantics: Driving evaluation

- Evaluation of a variable state $x : \rho$ now triggers evaluation of the state bound to x in ρ.
- Function and constructors are non-strict: evaluating a function or constructor application does not trigger evaluation of operands.
- However, primitive operators are strict and thus cause immediate evaluation of their operands.

As before, it is necessary to add error rules for dealing with ill-typed or partial operations. We elide this entirely trivial step.

Driving Evaluation

The semantics of Figure 5.2 define evaluation to WHNF. Correspondingly, the size-change termination analysis derived from this will detect programs for which evaluation to WHNF terminates. As we have seen before, while this can be an interesting notion of termination, a more natural notion is termination of the lazy evaluation driving loop.

To address this problem, we introduce a new syntactic construct **force** e to the language. The intention is that evaluating **force** e forces evaluation of e to normal form, rather than just WHNF. Given a program with main expression e, proving termination of the same program with main expression **force** e then proves termination of the lazy evaluation loop.

In choosing reduction to normal form, we are losing some information about the operational behaviour of the evaluation loop, namely the fact that the resulting value is printed *lazily*, without waiting for complete evaluation to normal form, but this is irrelevant for our purposes.

Evaluating an expression of the form **force** e leads to the evaluation of a *forced* value, denoted $v^{\blacktriangledown}$. This denotes the fact that the value v (in WHNF) should be evaluated further to normal form. The semantics of evaluating forced values are shown in Figure 5.3.

Sequentialisation

The next step is the *sequentialising* transformation, which takes the semantics defined in Figure 5.2, and defines a call relation $s \rightarrow s'$, with intended meaning: evaluation of s' is required to evaluate s.

We have defined this transformation in Appendix A for an arbitrary inference system, and so this may be applied automatically here. It suffices to note that the inference system of Figure 5.2 (together with error rules not shown here) satisfies the conditions given in Appendix A (determinism and totality, in particular). We can therefore conclude the essential property:

Property 5.1. *For any state s, $s \not\Downarrow$ iff there is an infinite sequence of calls starting at s.*

For completeness we give the call relation in Figure 5.4. The differences with the call relation in call-by-value naturally reflects the differences in the definition of the $\Downarrow$ relation that we have previously described.

Furthermore, the calls introduced by the **force** construct are given in Figure 5.5.

5.2.2 Ordering States

Let us now turn to the question of comparing values in the lazy semantics. This is superficially similar to the problem of comparing values in the strict semantics, and indeed we shall reuse the order defined in Section 3.1. A state is a finite tree (possibly with primitive constants at the leaves), and thus states may be compared with the subtree order $<$ as before.

However, this in itself is not sufficient to prove termination of nontrivial lazy programs, and in this section we shall remedy this problem (essentially introducing a refinement of the order defined previously).

Observing Decreases in Call-by-Name

Consider the following simple program:

```
f x =
  match x with
    Zero → 0
  | Succ y → 1 + f y
```

Certainly evaluation of f terminates, whenever x is bound to a finite numeral. However, consider the evaluation of the state $(\!|f : \{x \mapsto s\}|\!)$ (for some state s). This gives rise to the following call sequence:

$$(\!|f : \{x \mapsto s\}|\!) \rightarrow \mathit{Body}(f) : \{x \mapsto s\} \rightarrow f\ y : \{x \mapsto s, y \mapsto s'\}$$

for some state s'. Evaluation of the state $f\ y : \{x \mapsto s, y \mapsto s'\}$ (a function application) first triggers evaluation of the operator, which here evaluates to $(\!|f : \emptyset|\!)$. The function f is then

Variables and Function References

$$\frac{}{x : \rho \rightarrow \rho(x)} \qquad \frac{}{f : \rho \rightarrow (\!|f : \emptyset|\!)}$$

Closures

$$\frac{}{(\!|f : \rho|\!) \rightarrow \mathit{Body}(f) : \rho}\ {\scriptstyle \mathrm{dom}\,\rho = \mathit{Params}(f)}$$

Primitive Operators

$$\frac{e_1 : \rho \Downarrow c_1 \quad \cdots \quad e_i : \rho \Downarrow c_i}{\mathit{op}(e_1, \ldots, e_n) : \rho \rightarrow e_{i+1} : \rho}\ {\scriptstyle i<n \ (\forall j \leq i) c_j \in \mathit{PrimConst}}$$

Pattern Matching

$$\frac{}{\textbf{match}\ e\ \textbf{with}\ \left\langle \mathsf{C}_i(x_1^i, \ldots, x_{n_i}^i) \rightarrow e_i \right\rangle_{i=1}^{k} : \rho \rightarrow e : \rho}$$

$$\frac{e : \rho \Downarrow \mathsf{C}_l(s_1, \ldots, s_{n_l})}{\textbf{match}\ e\ \textbf{with}\ \left\langle \mathsf{C}_i(x_1^i, \ldots, x_{n_i}^i) \rightarrow e_i \right\rangle_{i=1}^{k} : \rho \rightarrow e_l : \rho \oplus \{x_j^l \mapsto s_j\}_{j=1}^{n_l}}$$

Conditionals

$$\frac{}{\textbf{if}\ e_g\ \textbf{then}\ e_t\ \textbf{else}\ e_f : \rho \rightarrow e_g : \rho}$$

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

Function Application

$$\frac{}{e_1 e_2 : \rho \rightarrow e_1 : \rho}$$

$$\frac{e_1 : \rho \Downarrow (\!|f : \mu|\!)}{e_1 e_2 : \rho \rightarrow \mathit{Bind}\ (\!|f : \mu|\!)\ (e_2 : \rho)}\ {\scriptstyle (\!|f:\mu|\!) \in \mathit{Closure}'}$$

Driving Evaluation

$$\frac{}{\textbf{force}\ e : \rho \rightarrow e : \rho}$$

Figure 5.4: The Call Relation

Driving Evaluation

$$\frac{}{\textbf{force}\ e : \rho \rightarrow e : \rho} \qquad \frac{e : \rho \Downarrow w}{\textbf{force}\ e : \rho \rightarrow w^{\blacktriangledown}}$$

Eager Evaluation

$$\frac{(\forall j < i) \quad s_j \Downarrow w_j \quad {w_j}^{\blacktriangledown} \Downarrow v_j}{\mathsf{C}(s_1, \ldots, s_n)^{\blacktriangledown} \rightarrow s_i}\ {\scriptstyle i \leq n} \qquad \frac{(\forall j < i) \quad s_j \Downarrow w_j \quad {w_j}^{\blacktriangledown} \Downarrow v_j \quad s_i \Downarrow w_i}{\mathsf{C}(s_1, \ldots, s_n)^{\blacktriangledown} \rightarrow {w_i}^{\blacktriangledown}}\ {\scriptstyle i \leq n}$$

Figure 5.5: Driving Evaluation: The Call Relation

called, passing the *unevaluated* operand. That is, the state bound to x in the callee is

$$t = y : \{x \mapsto s, y \mapsto s'\}$$

The recursive call in this program is thus of the form: $(\!|f : \{x \mapsto s\}|\!) \to^+ (\!|f : \{x \mapsto t\}|\!)$. However, as t is formed from the unevaluated expression y, this does not allow any decrease in the parameter x to be observed. Indeed, $t > s$ in the subtree order and so termination of this recursive loop cannot be proved (the size of x increases at each call).

The problem here is that the $<$ order is *syntactic*: values are compared by comparing their representations as closures. However, the decrease (from $\mathsf{Succ}\ y$ to y) in this program is *semantic* — the *value* of x in the callee is smaller than the value of x in the caller. This cannot be handled using only our existing framework.

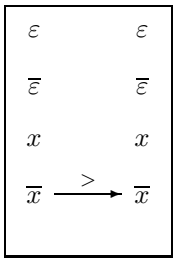
Semantic Decreases

It should be plain from the above example that some notion of semantic decrease (the decrease in the value of a state, rather than the state itself) must be incorporated into our analyses if termination of lazy programs is to be established. To introduce this, define a function Eval such that:

$$\text{Eval } s = \begin{cases} v & \text{if } s \Downarrow v \\ \bot & \text{if } s \not\Downarrow \end{cases}$$

Then Eval is well-defined by determinism of the semantics (though of course Eval is not computable).

We can now refine the analysis as follows. For each node p in the graph basis of a state, we add a node $\overline{p}$, representing the *value* of the substate at path p (or $\bot$ if evaluation of the substate at path p is nonterminating). For instance, a (1-limited) safe size-change graph for the call $f \to f$ shown above is:



The node $\overline{x}$ represents the value of parameter x, and thus it is certainly the case that $\overline{x} \xrightarrow{\geq} \overline{x}$ is safe. Furthermore this graph is now decreasing, and thus termination may be proved.

Let us now make this idea more precise. We first define graph bases and environment path lookup

Definition 5.2. Let s be a state. The *extended* graph basis for s is defined by $\text{gb}^e(s) = \text{gb}(s) \cup \{\overline{p} \mid p \in \text{gb}(s)\}$.

Definition 5.3. Let s be a state and $p \in \mathrm{gb}(s)$. Then $s\nabla\overline{p} = \mathrm{Eval}\ (s\nabla p)$.

We will adopt the convention that identifiers p, q, ... denote environment paths in $\mathrm{gb}(s)$ (so p is never of the form $\overline{r}$), while a, b, ... denote more general paths in $\mathrm{gb}^e(s)$.

As extended graph bases form supersets of existing graph bases all of our definitions (size-change graphs, safety and operations on graphs) carry over to this new setting as expected. It should be noted, however, that while $s\nabla(p +\!\!+ q) = (s\nabla p)\nabla q$, in general $s\nabla\overline{p +\!\!+ q} \neq (s\nabla\overline{p})\nabla\overline{q}$. Indeed, the right-hand side may not be well-defined. As a consequence, $s\nabla\overline{p +\!\!+ q}$ is *not* in general a subtree of $s\nabla\overline{p}$. This affects the *completion* operation defined in Section 4.2, which must not be applied to environment paths of the form $\overline{p}$.

Ordering Values Extended graph bases precisely capture the need to consider the value of substates, in addition to substates themselves. Given this we do not need to redefine the order used to compare values — the $<$ order defined previously is appropriate, but we are now able to compare both unevaluated and evaluated states.

A minor refinement must be made, as we have introduced the new value $\bot$ (representing the result of a nonterminating computation). We take $\bot < v$ for all $v \neq \bot$; this is safe as the $<$ order is still well-founded. This makes a nonterminating state smaller (semantically) than any other, which is useful in order to observe decreases statically, when it cannot be known whether any given state is terminating.

5.2.3 Constructing Size-Change Graphs

We have shown in the above how graph bases may be extended to deal with semantic decreases in lazy programs. To complete the picture we must now show that such decreases may be generated mechanically. We first observe that as extended graph bases are supersets of graph bases, any safe size-change graph defined previously is still safe in the extended sense. We thus only need to give *additional* size-change information, used to prove termination in call-by-name.

Operations on Graphs We have defined the basic operations on graphs in Section 3.3. As noted above these carry over to extended size-change graphs, with trivial modifications. For completeness we list the extended definitions below (omitting graph bases):

$$\begin{aligned}
1_s &= \{a \xrightarrow{\geq} a \mid a = p \vee a = \overline{p} \text{ for some } p \in \mathrm{gb}(s)\} \\
0 &= \emptyset \\
\gamma^* &= \{a \xrightarrow{l} b \in \gamma \mid a \neq \varepsilon \wedge a \neq \overline{\varepsilon}\} \\
\gamma^{*\dagger} &= \{a \xrightarrow{l} b \in \gamma \mid a, b \notin \{\varepsilon, \overline{\varepsilon}\}\} \\
\gamma[q/p] &= \{a \xrightarrow{l} q +\!\!+ r \mid a \xrightarrow{l} p +\!\!+ r \in \gamma\} \cup \{a \xrightarrow{l} \overline{q} \mid a \xrightarrow{l} \overline{p} \in \gamma\}
\end{aligned}$$

Values and Evaluation Steps Suppose that $s \Downarrow v$, and that γ is safe for (s, v). Then as v is a value, $v\nabla\overline{\varepsilon} = \text{Eval } v = v = v\nabla\varepsilon$. Furthermore, $s\nabla\overline{\varepsilon} = \text{Eval } s = v = v\nabla\varepsilon$. Hence the graph

$$\begin{aligned}\gamma^{\text{val}} &= \gamma \cup \text{ValueGraph} \\ &\quad \textbf{where } \text{ValueGraph} = \{\overline{\varepsilon} \to \overline{\varepsilon}, \overline{\varepsilon} \to \varepsilon\}\end{aligned}$$

is likewise safe for (s, v). We note that the edge $\overline{\varepsilon} \to \varepsilon$ in ValueGraph implies all edges of the form $\overline{\varepsilon} \xrightarrow{>} p$ (for $p \neq \varepsilon$), and that these are obtained by completion of the graph as before.

Similarly, suppose that $s \to s' \Downarrow v$ with γ safe for (s, s'), and that the value of s is the same as that of s'. Such a call represents one step in the evaluation of s, and instances of this pattern may be found in the rules for evaluating conditionals, pattern matching and function application. Then Eval $s =$ Eval s', so $s\nabla\overline{\varepsilon} = s'\nabla\overline{\varepsilon}$. Thus the graph

$$\begin{aligned}\gamma^{\text{step}} &= \gamma \cup \text{StepGraph} \\ &\quad \textbf{where } \text{StepGraph} = \{\overline{\varepsilon} \to \overline{\varepsilon}\}\end{aligned}$$

is safe for (s, s').

Finally, whenever c is a constant then Eval $c = c$. Hence the graphs $\mathit{SizeChange}_{\text{op}}\ \mathit{op}$ encoding size-change properties of primitive operators can be extended: for each edge $p \xrightarrow{l} q$ we add edges $\overline{p} \xrightarrow{l} \overline{q}$, $\overline{p} \xrightarrow{l} q$ and $p \xrightarrow{l} \overline{q}$ (as all environment paths represent constants).

Pattern Matching The graph for evaluating pattern matching expressions is unchanged from the call-by-value case.

Function Application Let us now consider the case of a function application $s = e_1 e_2 : \rho$. To evaluate s, the operand $s_1 = e_1 : \rho$ is evaluated to a closure w say. Then $s \to s'$, where s' is the result of binding $s_2 = e_2 : \rho$ to x say in w. Let γ be safe for (s_1, w). Then as in the call-by-value case the graph $\gamma^{*\dagger}$ is safe for (s, s'). The problem of evaluating dataflow for the *operand* is of more interest — this is where the naive application of our existing size-change graphs is insufficient.

We have two cases. Suppose first that $s_2 \Downarrow v$, with δ safe for (s, v). Then $v = \text{Eval } v = \text{Eval } s_2 = \text{Eval } (s'\nabla x)$. Thus for any edge $a \xrightarrow{l} \varepsilon$ or $a \xrightarrow{l} \overline{\varepsilon}$ in δ, the edge $a \xrightarrow{l} \overline{x}$ is safe for (s_2, s'). Thus the following graph is safe for (s, s'):

$$\gamma^{*\dagger} \cup \{a \xrightarrow{l} \overline{x} \mid a \notin \{\varepsilon, \overline{\varepsilon}\} \wedge (a \xrightarrow{l} \varepsilon \in \delta \vee a \xrightarrow{l} \overline{\varepsilon} \in \delta)\}$$

This graph records dataflow information from the caller state to the *evaluated* argument.

If $s_2 \not\Downarrow$, we may take $\gamma^{*\dagger}$ as size-change graph for the call, as no other information can be given.

Static Analysis The graph defined for function application in the above may not directly be used in static analysis. For, to construct the graph it is necessary to know whether or not $s_2 \Downarrow$. This may however be approximated, yielding a graph construction that is appropriate for static analysis.

The crucial observation is that as $\bot \leq v$ for all values v, *any* arrow of the form $a \xrightarrow{\geq} \overline{x}$ is safe if evaluation of x is nonterminating. Furthermore, whenever p is an unevaluated environment path, then p cannot represent $\bot$, whence $p \xrightarrow{\geq} \overline{x}$ is safe if the value of x is $\bot$.

Define:

$$\begin{array}{lcl} \mathrm{app}^{\mathrm{lazy}}(\gamma,\delta) & = & \gamma^{*\dagger} \cup \delta^{*\,\mathrm{eval}}_{x} \\ \textbf{where } \beta^{\mathrm{eval}}_{x} & = & \{p \xrightarrow{l} \overline{x} \mid p \xrightarrow{l} \varepsilon \in \beta \vee p \xrightarrow{l} \overline{\varepsilon} \in \beta\} \cup \{\overline{p} \xrightarrow{\geq} \overline{x} \mid \overline{p} \to \varepsilon \in \beta \vee \overline{p} \to \overline{\varepsilon} \in \beta\} \end{array}$$

In the definition of $\beta^{\mathrm{eval}}_{x}$ some information is lost — edges of the form $\overline{p} \xrightarrow{>} \overline{\varepsilon}$ are converted to *nonincreasing* edges $\overline{p} \xrightarrow{\geq} \overline{x}$. This is a source of imprecision, but is necessary to deal with the case of potentially nonterminating operands.

In return for this loss of precision we obtain the following property (as justified above):

Lemma 5.4. *Let $s = e_1 e_2 : \rho$, and suppose that $s \to s'$, where $s_1 = e_1 : \rho \Downarrow w$ and $s' = \mathit{Bind}\ w\ (e_2 : \rho)$. Suppose further that γ is safe for (s_1, w). Then:*

1. *If $s_2 \Downarrow v$ and δ is safe for (s_2, v), then $\mathrm{app}^{\mathrm{lazy}}(\gamma,\delta)$ is safe for (s, s'), and*
2. *If $s_2 \not\Downarrow$, then $\mathrm{app}^{\mathrm{lazy}}(\gamma,\delta)$ is safe for (s, s') for any graph δ.*

This may be used to produce size-change graphs in the static call graph. Let t be an abstract state, say $t = e_1 e_2$ (for definiteness we take 0CFA here, but this applies to any control-flow analysis), and put $t_1 = e_1$ and $t_2 = e_2$. We define:

$$\mathrm{OperandGraphs}(t_2) = \begin{cases} \{\delta \mid t_2 \Downarrow^{\alpha} v, \delta\} & \text{if this is nonempty} \\ \{0\} & \text{otherwise} \end{cases}$$

The second case ensures that *some* graph can be produced even if the operand is not found to have a value. Then the function application call is labelled with all graphs $\mathrm{app}^{\mathrm{lazy}}(\gamma,\beta)$ with $\beta \in \mathrm{OperandGraphs}(t_2)$.

It should be noted that this is well-defined. The difficulty is that $\mathrm{OperandGraphs}(s)$ is not a monotonic function of the annotated static call graph, but is used in the fixpoint definition of this call graph. However this can give rise to no circularity. For, whether or not the set $\{\delta \mid t_2 \Downarrow^{\alpha} v, \delta\}$ is empty depends only on the *control flow* (static call graph): this is empty iff $t_2 \not\Downarrow^{\alpha}$. As $\mathrm{OperandGraphs}(t_2)$ is only used to produce size-change annotations, not control flow, this is safe.

Constructor Application Consider a constructor application $s = \mathsf{C}(e_1, \ldots, e_n) : \rho \Downarrow \mathsf{C}(s_1, \ldots, s_n) = v$, where $s_i = e_i : \rho$. As $s \Downarrow v$, the graph ValueGraph is safe for (s, v).

We can do better, however, as in the case of function application, and indeed constructor application is entirely analogous to function application. Let t be the abstract state $\mathsf{C}(e_1, \ldots, e_n)$, and $t_i = e_i$ for each i. For graphs γ_i, define:

$$\text{constr}^{\text{lazy}}(\gamma_1, \ldots, \gamma_n) = \text{ValueGraph} \cup \gamma_1{}^{\text{eval}}_{x_1} \cup \cdots \cup \gamma_n{}^{\text{eval}}_{x_n}$$

Then the edge $t \Downarrow \mathsf{C}$ may safely be annotated with all graphs of the form $\text{constr}^{\text{lazy}}(\gamma_1, \ldots, \gamma_n)$, where $\gamma_i \in \text{OperandGraphs}(t_i)$.

Driving Evaluation We finally define size-change graphs to annotate evaluation of *forced* states to normal form. The interesting case is that of a constructor value $s = \mathsf{C}(s_1, \ldots, s_n)^{\blacktriangledown}$ say. Then $s \to s_i$ for each i. Furthermore, whenever $s_i \Downarrow w_i$, then $s \to w_i{}^{\blacktriangledown}$. As s_i is the substate bound to x_i in s, the graph $1[x_i/\varepsilon]$ is safe for (s, s_i) for each i.

Furthermore, suppose that γ_i is safe for (s_i, w_i) for each i. Then $1[x_i/\varepsilon]; \gamma_i$ is safe for $(s, w_i{}^{\blacktriangledown})$.

5.2.4 Static Analysis

The analysis of lazy languages may now be completed by adapting the static analyses to this new setting. This is comparatively straightforward: the abstract domains are essentially unchanged, with the usual exception that the set of trees to be considered now includes trees with states, not just values, as subtrees.

We define 0CFA analysis of lazy languages in Figures 5.6 and 5.7, for reference. The process of deriving this analysis from the semantics is essentially identical to the developments of Chapter 4, and thus we shall not describe the analysis further. Naturally, the size-change annotations use the more refined size-change graphs constructions defined above. As before the k-bounded CFA and tree automata analyses may be defined for lazy languages, but these are omitted as their derivation is somewhat mechanical.

Soundness of the analysis is proved exactly as in the call-by-value case. It must be shown that closure analysis safely approximates reachable states, and from this fact control-safety of the call graph can be deduced. We will therefore not repeat the proof, but merely state the result:

Proposition 5.5. *The 0CFA analysis defined in Figures 5.6 and 5.7 is a safe approximation of the dynamic call graph.*

Values, Function Names and Constants

$$\frac{}{v \Downarrow v, 1}\ {\scriptstyle v \in AbsVal} \qquad \frac{(\!|f : \emptyset|\!) \Downarrow v, \gamma}{f \Downarrow v, \mathrm{StepGraph}; \gamma} \qquad \frac{}{c \Downarrow \alpha(c), \mathrm{ValueGraph}}$$

Primitive Operators

$$\frac{(\forall i) e_i \Downarrow c_i, \gamma_i \quad \mathit{Apply}^{\alpha}\ \mathit{op}\ \langle c_1, \ldots, c_n \rangle = c}{\mathit{op}(e_1, \ldots, e_n) \Downarrow c, \mathit{OpGraph}\ \mathit{op}\ \langle (c_1, \gamma_1), \ldots, (c_n, \gamma_n) \rangle}$$

Constructors and Pattern Matching

$$\frac{}{\mathsf{C}(e_1, \ldots, e_n) \Downarrow \mathsf{C}, [\mathrm{constr}(\gamma_1, \ldots, \gamma_n)]_1}\ {\scriptstyle (\forall i)\gamma_i \in \mathrm{OperandGraphs}(e_i)}$$

$$\frac{e \Downarrow c, \gamma \quad e_l \Downarrow v, \delta}{s = \mathbf{match}\ e\ \mathbf{with}\ \left\langle \mathsf{C}_i(x_1^i, \ldots, x_{n_i}^i) \rightarrow e_i \right\rangle_{i=1}^{k} \Downarrow v, \mathrm{patmatch}_s^l(\gamma)^{\mathrm{step}}; \delta}\ {\scriptstyle c \sqsupseteq \mathsf{C}_l}$$

Conditionals

$$\frac{e_g \Downarrow c \sqsupseteq \alpha(\mathbf{true}), \gamma_g \quad e_t \Downarrow v, \gamma}{\mathbf{if}\ e_g\ \mathbf{then}\ e_t\ \mathbf{else}\ e_f \Downarrow v, \mathrm{StepGraph}; \gamma} \qquad \frac{e_g \Downarrow c \sqsupseteq \alpha(\mathbf{false}), \gamma_g \quad e_f \Downarrow v, \gamma}{\mathbf{if}\ e_g\ \mathbf{then}\ e_t\ \mathbf{else}\ e_f \Downarrow v, \mathrm{StepGraph}; \gamma}$$

Function Application

$$\frac{e_1 \Downarrow (\!|f : S|\!), \gamma_1 \quad \mathit{Bind}\ (\!|f : S|\!)\ e_2 \Downarrow v, \delta}{e_1 e_2 \Downarrow v, \left[\mathrm{app}^{\mathrm{lazy}}(\gamma_1, \gamma_2)\right]_1; \delta}\ {\scriptstyle S \subsetneq Params(f) \wedge x = \mathrm{hd}(Params(f) \setminus S)\ \gamma_2 \in \mathrm{OperandGraphs}(e_2)}$$

Closure Analysis

$$\frac{(\exists e_1 e_2 \in P) \quad e_1 \Downarrow (\!|f : S|\!), \gamma \quad e_2 \Downarrow v, \delta}{x \Downarrow v, \overline{1[\varepsilon/x]}; \delta}\ {\scriptstyle S \subsetneq Params(f) \wedge x = \mathrm{hd}(Params(f) \setminus S)}$$

$$\frac{(\exists \mathsf{C}_j(e_1, \ldots, e_n) \in P) \quad e \Downarrow \mathsf{C}_j, \gamma \quad e_j \Downarrow v, \delta}{x_i^j \Downarrow v, \overline{1[\varepsilon/x_i^j]}; \delta}\ {\scriptstyle \mathrm{Binder}(x_i^j) = \mathbf{match}\ e\ \mathbf{with}\ \left\langle \mathsf{C}_i(x_1^i, \ldots, x_{n_i}^i) \rightarrow e_i \right\rangle_{i=1}^{k}}$$

$$\frac{(\exists e_1 e_2 \in P) \quad e_1 \Downarrow *, \gamma}{x \Downarrow *, \overline{1[\varepsilon/x]}}\ {\scriptstyle x \text{ is a function parameter}}$$

$$\frac{e \Downarrow *, \gamma}{x \Downarrow *, \overline{1[\varepsilon/x]}}\ {\scriptstyle \mathrm{Binder}(x) = \mathbf{match}\ e\ \mathbf{with}\ \left\langle \mathsf{C}_i(x_1^i, \ldots, x_{n_i}^i) \rightarrow e_i \right\rangle_{i=1}^{k}}$$

Driving Evaluation

$$\frac{e \Downarrow w, \gamma \quad w^{\blacktriangledown} \Downarrow v, \delta}{\mathbf{force}\ e \Downarrow v, \gamma^{*\mathrm{step}}; \delta} \qquad \frac{}{c^{\blacktriangledown} \Downarrow c, 1}\ {\scriptstyle c \in AbstConst}$$

$$\frac{}{v^{\blacktriangledown} \Downarrow v, 1}\ {\scriptstyle v \in Closure} \qquad \frac{}{*^{\blacktriangledown} \Downarrow *, 1}$$

$$\frac{(\exists \mathsf{C}(e_1, \ldots, e_n) \in P) \quad (\forall i) e_i \Downarrow w_i, \gamma_i \quad w_i{}^{\blacktriangledown} \Downarrow v_i, \delta_i}{\mathsf{C}^{\blacktriangledown} \Downarrow \mathsf{C}, \left[\mathrm{constr}(\overline{1[\varepsilon/x_1]}; \gamma_1; \delta_1, \ldots, \overline{1[\varepsilon/x_n]}; \gamma_n; \delta_n)\right]_1}$$

Program Template Variables

$$\frac{}{X \Downarrow \mathfrak{W}(X), \mathrm{ValueGraph}}\ {\scriptstyle \text{with valuation } \mathfrak{W}}$$

Figure 5.6: 0CFA Analysis in the Lazy Semantics (Evaluation)

Primitive Operators

$$\frac{(\forall j < i) \quad e_j \Downarrow v_j}{op(e_1, \ldots, e_n) \to e_i, 1^*}\ {}^{i \leq n}$$

Evaluating Closures and Function References

$$\frac{}{(\!|f : S|\!) \to Body(f), 1^{*\text{step}}}\ {}^{S = Params(f)} \qquad \frac{}{f \to (\!|f : \emptyset|\!), \text{StepGraph}}$$

Pattern Matching

$$\frac{\dfrac{}{\textbf{match}\ e\ \textbf{with}\ \left\langle \mathsf{C}_i(x_1^i, \ldots, x_{n_i}^i) \to e_i \right\rangle_{i=1}^{k} \to e, 1^*} \quad e \Downarrow c, \gamma}{s = \textbf{match}\ e\ \textbf{with}\ \left\langle \mathsf{C}_i(x_1^i, \ldots, x_{n_i}^i) \to e_i \right\rangle_{i=1}^{k} \to e_l, \text{patmatch}_s^l(\gamma)^{\text{step}}}\ {}^{c \sqsupseteq \mathsf{C}_l}$$

Conditionals

$$\frac{\dfrac{}{\textbf{if}\ e_g\ \textbf{then}\ e_t\ \textbf{else}\ e_f \to e_g, 1^*} \quad e_g \Downarrow c \sqsupseteq \alpha(\textbf{true})}{\textbf{if}\ e_g\ \textbf{then}\ e_t\ \textbf{else}\ e_f \to e_t, 1^{*\text{step}}} \qquad \frac{e_g \Downarrow c \sqsupseteq \alpha(\textbf{false})}{\textbf{if}\ e_g\ \textbf{then}\ e_t\ \textbf{else}\ e_f \to e_f, 1^{*\text{step}}}$$

Function Application

$$\frac{}{e_1 e_2 \to e_1, 1^*} \qquad \frac{e_1 \Downarrow (\!|f : S|\!), \gamma_1}{e_1 e_2 \to \mathit{Bind}\ (\!|f : S|\!)\ e_2, \left[\text{app}^{\text{lazy}}(\gamma_1, \gamma_2)\right]_1}\ {}^{\substack{S \subsetneq Params(f) \wedge \\ x = \text{hd}(Params(f) \setminus S) \\ \gamma_2 \in \text{OperandGraphs}(e_2)}}$$

Closure Analysis

$$\frac{(\exists e_1 e_2 \in P) \quad e_1 \Downarrow (\!|f : S|\!), \gamma}{x \to e_2, \overline{1[\varepsilon/x]}}\ {}^{S \subsetneq Params(f) \wedge x = \text{hd}(Params(f) \setminus S)}$$

$$\frac{(\exists \mathsf{C}_j(e_1, \ldots, e_n) \in P) \quad e \Downarrow \mathsf{C}_j, \gamma}{x_i^j \to e_j, \overline{1[\varepsilon/x_i^j]}}\ {}^{\text{Binder}(x_i^j) = \textbf{match}\ e\ \textbf{with}\ \left\langle \mathsf{C}_i(x_1^i, \ldots, x_{n_i}^i) \to e_i \right\rangle_{i=1}^{k}}$$

Driving Evaluation

$$\frac{}{\textbf{force}\ e \to e, 1^*} \qquad \frac{e \Downarrow w, \gamma}{\textbf{force}\ e \to w^{\blacktriangledown}, \gamma^{*\text{step}}}$$

$$\frac{(\exists \mathsf{C}(e_1, \ldots, e_n) \in P)}{\mathsf{C}^{\blacktriangledown} \to e_i, \overline{1[\varepsilon/x_i]}}$$

$$\frac{(\exists \mathsf{C}(e_1, \ldots, e_n) \in P) \quad e_i \Downarrow w_i, \gamma}{\mathsf{C}^{\blacktriangledown} \to w_i^{\blacktriangledown}, \overline{1[\varepsilon/x_i]}; \gamma}$$

Figure 5.7: 0CFA Analysis in the Lazy Semantics (Calls)

5.3 Discussion

5.3.1 Static Analysis

Primitive Operators and Observable Decreases

The introduction of nodes corresponding to evaluated parameters in the graph basis of a state was necessary to observe *any* decreases in argument sizes along recursive calls, due to the fact that arguments are passed unevaluated in call-by-name. We have shown that this allows decreases to be recorded along calls such as $f\ (\mathsf{Succ}\ x) \to f\ x$. That is, decreases in *algebraic constants* may be handled with the techniques introduced in this chapter.

However, dealing with decreases in *primitive constants* is unfortunately nontrivial in our analysis. Consider the trivial function definition:

```
f x = if x = 0 then 0 else f (x-1)
```

Evaluation of the expression $x-1$ may be annotated with the size-change information $\overline{x} \xrightarrow{>} \overline{\varepsilon}$ (the value of n is greater than the value of the result). However, this only gives rise to a size-change arrow $\overline{x} \xrightarrow{\geq} \overline{x}$ in the call $f \to f$, using the graph constructions that we have defined above.

This loss of information prevents us from proving that f terminates. It is, however, required in general for soundness. For, if evaluation of $x - 1$ was nonterminating, then the graph obtained in the abstract interpretation need not be valid (that the arrow $\overline{x} \xrightarrow{\geq} \overline{x}$ can be retained is due to our choice of $\bot$ as the least element in the $<$ order).

The techniques of this section therefore require us to abandon the use of natural numbers as primitive constants, and to use an encoding of Peano numbers as an algebraic datatype instead. There does not appear to be a natural extension of this analysis that can handle recursion over primitive constant values.

Finite Input

Let us briefly observe an important consequence of the 0CFA analysis defined in Figures 5.6 and 5.7. Note that the special value $*$ is given the evaluation judgement:

$$\frac{}{*^{\blacktriangledown} \Downarrow *, 1}$$

This implies that forcing evaluation of $*$ to *normal form*, rather than WHNF, is always terminating. As a consequence, only *finite* values are described by $*$.

This is intentional, as allowing infinite inputs precludes termination proofs for even simple functions (say, *reverse*). It is thus useful to be able to represent values that are guaranteed to be finite (but are otherwise unspecified). Should we wish to establish termination of a function on (for instance) any finite or infinite list input, this may be achieved by constructing

an arbitrary list value within the program. For instance, an arbitrary infinite list can be constructed as:

```
anylist = <X> : anylist
```

where X is a program template variable.

Unbounded Environments

We have noted in previous chapters that programs may create environments of unbounded depths. It may be argued that this is somewhat rare, however — in fact, in our presentation of k-bounded CFA it was stated that for many programs this is the case, justifying k-bounded CFA (for small values of k) as a useful abstraction.

Certainly our example of the higher-order fold-left function (p. 80) cannot be said to be typical of most functional programs. However, the behaviour exhibited by this program is typical of many *lazy* functional programs.

An example of this behaviour can be found in the definition of the (space-leaking) *sum* function using *foldl* in Haskell [Bir98, ch. 7.5]:

```
foldl f e [] = e
foldl f e (x:xs) = foldl f (f e x) xs

sum xs = foldl (+) 0 xs
```

This program traverses the list, accumulating the sum of its entries in the e parameter of *foldl*. However, as this traverses the list evaluation of the sum is not forced (the term $f\ e\ x$, which as f is bound to + adds a summand, appears in operand position and so is not evaluated).

When the recursion reaches the bases case of *foldl*, the state of the evaluation is therefore a suspended computation represented by a tree of depth the length of the list. The evaluation of this state is then forced, resulting in the sum being computed.

As a result, the states that appear in the execution of lazy programs are much more likely to be represented by trees of arbitrary depth. The tree automata analysis thus emerges as the most natural candidate for static analysis of lazy programs when the context-insensitive 0CFA analysis is not sufficient.

This case illustrates that even for very simple, obviously terminating programs (though leaking in this case), the operational behaviour under call-by-name can be quite involved, making control flow analysis more challenging.

5.3.2 Another Approach: Program Tranformation

Throughout this chapter, we have outlined the development of the size-change termination analysis for lazy programs. The steps towards the analysis are essentially similar to the call-by-value case, with the exception of the need to introduce a more general notion of order on values to deal with lazy programs effectively.

We shall now briefly discuss an alternative approach, based on *transforming* a lazy program into a call-by-value program, while preserving the termination properties of the original program.

Laziness in OCaml

The OCaml language is of course a strict programming language, and thus is given a call-by-value semantics. However, the language provides support for lazy evaluation using *explicit* annotations in the source code.

A built-in type

```
type α Lazy.t
```

is provided to represent lazy values of type α. Such a value is initially suspended, and its evaluation may be forced when required. Furthermore, if evaluation is forced, the result is cached so that further uses of the lazy value do not trigger recomputation. A lazy value can be constructed by expressions of the form **lazy** e: if e has type α then **lazy** e has type α *lazy*. The value of e is *not* evaluated when **lazy** e is evaluated, which requires **lazy** to be a keyword rather than a constructor or defined identifier. Finally,

```
force : α Lazy.t → α
```

forces evaluation of a lazy value (if it is not already evaluated) and returns its value.

As an example of the use of the OCaml laziness constructs let us give the *repmin* program in OCaml using explicit laziness:

```
type α tree = Tip of α | Fork of α tree × α tree

rpm : α Lazy.t → β tree → α tree Lazy.t × β

let rec rpm x =
  function
    Tip y → (lazy (Tip (force x)), y)
  | Fork (t1, t2) →
     let (tr1, mt1) = rpm x t1
     and (tr2, mt2) = rpm x t2 in
       (lazy (Fork (force tr1, force tr2)), min mt1 mt2)

let repmin t =
  let rec p = lazy (rpm (lazy (snd (force p))) t) in
  let (tres, minval) = force p in
    force tres
```

In this program, the x parameter of *rpm* is a *lazy* value, as is the tree that *rpm* returns. The minimum value returned by *rpm* does not need to be lazy, and the pair returned by *rpm* can

likewise be eagerly evaluated. The recursive definition in the body of *repmin* is made more complicated than necessary by the syntactic restrictions of (the implemented extension of) OCaml on recursive value definitions.

Embedding Call-by-Name in Call-by-Value

The support for explicit laziness in OCaml highlights the second approach that we may take from a termination analysis point of view. By using an embedding of lazy languages in call-by-value languages based on program transformation, we can sidestep the issue entirely and reuse our existing analysis for strict languages.

Let us make the translation from call-by-name to call-by-value more precise. As before we shall not be concerned with *laziness* (in particular the use of caching), but rather with the implementation of call-by-name. These are of course equivalent from a termination standpoint.

The translation of call-by-name into call-by-value is well-known [Plo75]: every application $e_1 e_2$, where the value of e_2 should be passed by name, is replaced by $e_1\ (\lambda().e_2)$[3]. The expression $\lambda().e_2$ is a *suspension* (or thunk) — evaluation of this expression terminates immediately, and the value of e_2 can be recovered from it.

Within the body of a function $f\ x_1\ \cdots\ x_n = e$, for each variable x_i which should be passed by name, we replace every occurrence of x_i within e by $x_i\ ()$.

Termination Analysis

The encoding shown in the above suffices to reduce the termination analysis problem for lazy languages to call-by-value languages. For this to be useful in practice, however, we must check that the translated programs are amenable to size-change termination analysis. The control flow of a translated program (under call-by-value) is essentially equivalent to the control flow of the original program under call-by-name. The only issue is therefore whether size changes may be effectively observed in translated lazy programs.

Unfortunately, the same difficulty arises as in the development of the analysis for lazy programs: decreases along function calls are not observed, as the operand is not evaluated. In the translated program this corresponds to the operand being a function $\lambda().e$, and the size of such a function value need not be related to the size of the value of e. It is therefore essential to deal with laziness directly, rather than relying purely on program transformation. Let us however note that this problem may be alleviated somewhat by the use of *strictness analysis* [Myc80, BHA86, Wad87] to identify function parameters that do not need to be passed by name. It is then safe to evaluate such function parameters in applications, allowing decreases to be observed exactly as in the call-by-value case.

[3] In this section we assume that our language contains λ-abstractions; these can be eliminated by λ-lifting as usual.

Binding Construct	Variable	Values	
infinite	i	14	3
at	xs	13	11
	n	15	12
	y	1	
	ys	2	
	m	$*$	
(::)	x_1	1	
	x_2	2	

Figure 5.8: Closure Analysis

5.4 An Example

Let us now illustrate the termination analysis of lazy programs with a simple example. We shall not attempt in this section to give an experimental evaluation of the effectiveness of this analysis, but rather to illuminate the definition by example.

Our example program is artificial, for concision:

```
infinite i = 1i :: 2(infinite 3(4i+5 1))

at xs n =
  match 6xs with
    y :: ys →
    7match 8n with
        Zero → 9y
      | Succ m → 10(at 11ys 12m)

at 13(infinite 14 0) 15<*>
```

where superscripts denote expression labels. While we evaluate the result to WHNF, as the result is a natural number in this case, this is equivalent to normal form.

The results of closure analysis are given in Figure 5.8. For each function parameter, the values found by closure analysis simply correspond to the *labels* of expressions in operand position at binding sites (represented by integers), as parameters are passed unevaluated. The exception is the value $*$, as it is assumed that this represents a value in normal form. This is crucial, as this program would not terminate if n was bound to an infinite numeral.

The 0CFA call graph is shown in Figure 5.9. Rectangular nodes indicate expression states (denoted by the label of the corresponding expression), while oval nodes indicate closure states.

There are two cycles in the call graph. The first cycle is the expected cycle around *at*, while the second cycle occurs when evaluation of i is triggered. This second cycle is justified by the fact that the value of i is represented by an accumulated suspended computation — the k^{th} value of *infinite* 0 is obtained by adding 1 k times to 0, and as this is delayed it is

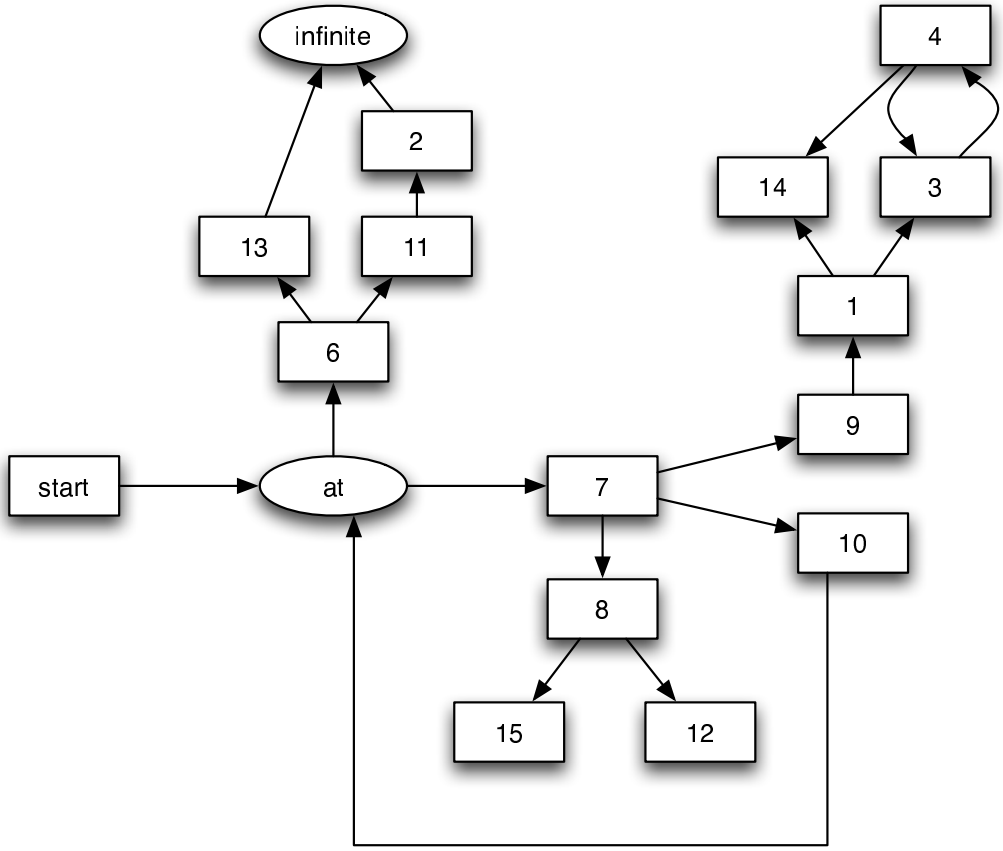
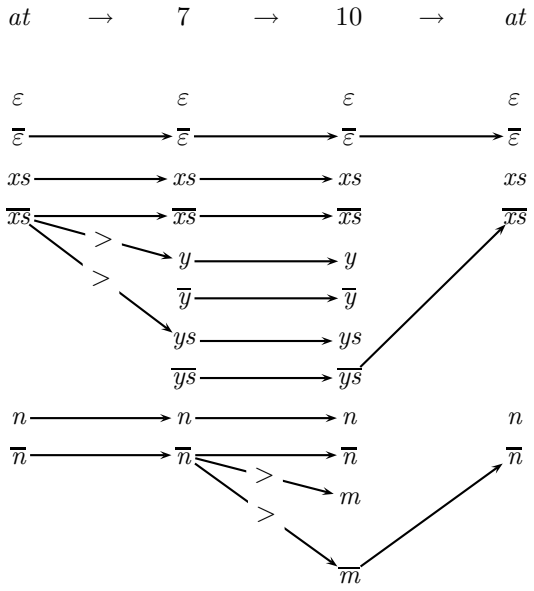


Figure 5.9: Call Graph

represented by a suspension of depth k. Forcing this suspension causes evaluation of all $k-1$ previous values of i.

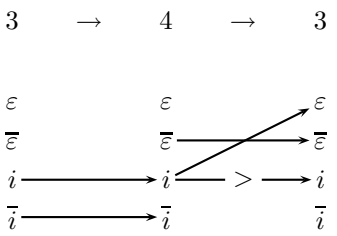
Both loops in the call graph are size-change terminating. Let us first examine the $at \to at$ loop. The size-change graphs for intermediate calls are shown below:



The composition of these graphs yields the decrease $\overline{m} \xrightarrow{>} \overline{m}$, which allows termination to be proved. This is a decrease in the *value* of m, rather than its syntactic representation.

It is important to note that this decrease is only observable because of the edge $\overline{n} \xrightarrow{>} \overline{m}$ in the graph for $7 \to 10$. This edge is safe (and can be deduced in the static analysis) precisely because the value of n is always $*$, which is assumed evaluated to normal form. In particular, $\overline{m}$ always represents the same state as m (unlike, for instance, $\overline{ys}$ which may be different from ys). This is crucial for soundness: the termination proof is only valid when the n parameter of at has a normal form (and thus represents a finite numeral).

To complete the picture, let us consider the second cycle, namely $3 \to 4 \to 3$ in the evaluation of i. The size-change graphs are shown below:



The composition of these graphs contains the edge $i \xrightarrow{>} i$, guaranteeing termination. In this cycle, unlike the previous case, the decrease is syntactic. The origin of the decrease is the variable evaluation call $4 \to 3$: the callee state is just the value of i in the caller, whence the arrow $i \to \varepsilon$ is safe. The edge $i \xrightarrow{>} i$ is deduced by completion of the graph with respect to the subtree order.

This completes our example of the size-change termination analysis for lazy programs. This example, though simple, illustrates many important features:

1. Using *semantic* decreases is crucial to allow nontrivial termination proofs
2. *Syntactic* decreases are invaluable to prove termination of loops caused by forcing nested suspensions
3. The use of $*$ to represent *finite* input only allows a more refined termination criterion.

Chapter 6

Expressive Power

6.1 Introduction

We have introduced, in chapters 3 and 4, the size-change criterion for termination, and three classes of static analyses approximating this criterion (the context-insensitive 0CFA analysis, the bounded context domain k-bounded CFA analysis and single-label tree automata). We have further shown that these analyses are *sound*: whenever one of the static analyses described previously detects that a program is terminating, this program is indeed guaranteed to terminate.

However, undecidability of the halting problem guarantees that no computable analysis can be *complete*. For each of our analyses, there is thefore a set of programs that do terminate, but are not detected by the analysis. Furthermore, soundness in itself is not a very informative property — after all, the trivial termination analysis that returns a "don't know" answer on every input program is sound.

The purpose of this chapter is to shed more light on what we can hope to achieve by the methods previously introduced. We start by making the problem more precise by outlining classical results about precision and decidability of program analysis. We shall then focus on the causes of *false negatives*, to illustrate the inherent limitations of static analysis. We shall then use this to relate the expressiveness of different static analyses (0CFA, k-bounded CFA and tree automata). Finally, we shall give an indication of how size-change termination compares to other existing termination criteria of programs (focusing on the simply-typed λ-calculus and Knuth-acyclic attribute grammars).

As a roadmap for this chapter, a summary of the results that we aim to show is presented in Figure 6.1. This illustrates the relationship between the different static analyses that we have presented (0CFA, the k-bounded CFA analyses denoted simply by kCFA, and the tree automata analysis denoted by TA in the diagram). Furthermore, the set $\lambda^{\rightarrow}$ of simply-typed λ-expressions is shown in relation to our static analyses. The aim of this chapter is to prove Figure 6.1 correct.

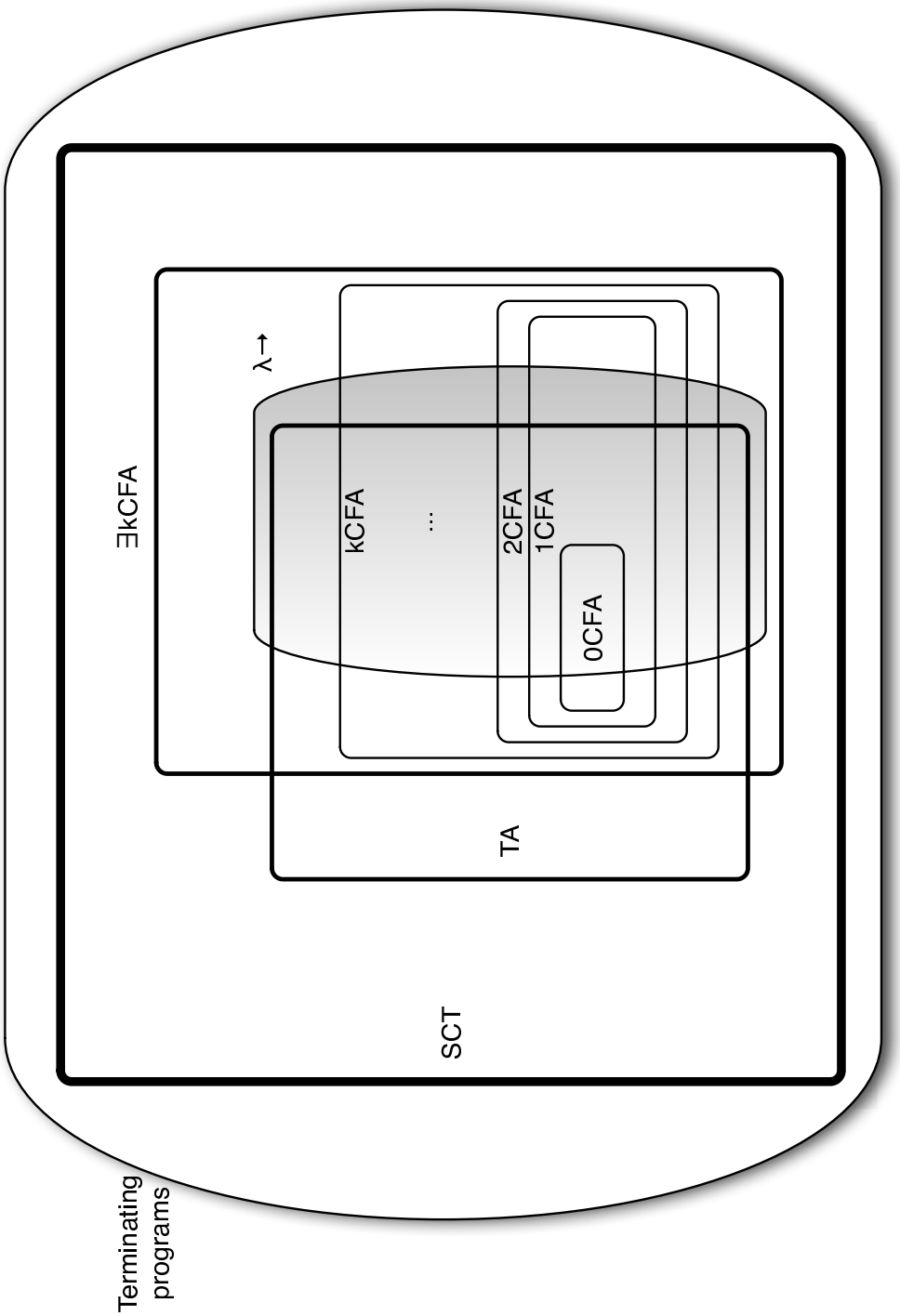


Figure 6.1: Relationships between Classes of Terminating Functional Programs

6.1.1 Program Properties and Undecidability

Throughout this section we denote programs by P, Q, etc. Given any program P, the *extension* of P is the function that P computes. In our framework, unknown parameters are modelled by free template variables (X, Y, ...) to which values are assigned by a *valuation* $\mathfrak{V}$. The extension of a program can therefore be expressed as:

$$[\![P]\!]\ \mathfrak{V} = v \quad \iff \quad P \Downarrow v \text{ with valuation } \mathfrak{V}$$

The determinism property guarantees that this is well-defined. If $P \not\Downarrow$ with valuation $\mathfrak{V}$ (by totality, this happens exactly when execution of P is nonterminating) we write $[\![P]\!]\ \mathfrak{V} = \bot$. This notation will be familiar to readers versed in denotational semantics. However, we should note that $\bot$ just represents nontermination in our framework, and excludes run-time errors.

An (intensional) *property* of programs is a predicate $\varphi(P)$ on programs (equivalently, a set of programs). Examples of properties include: P terminates on every input, P never returns an error value, P is size-change terminating on 0CFA. Of these three properties, only the last is decidable. A program analysis corresponds to a decidable intensional criterion on programs.

A property φ of programs is *extensional* if: whenever $[\![P]\!] = [\![P']\!]$, then $\varphi(P) \iff \varphi(P')$. That is, φ is really a property of the *function* computed by P, rather than P itself. Termination is an extensional property: P is terminating on every input iff $[\![P]\!]\ \mathfrak{V} \neq \bot$ for all $\mathfrak{V}$. On the other hand many properties are not extensional, ranging from trivial properties such as "P is 42 lines long" to more interesting examples such as 0CFA size-change termination.

As noted above, termination is an extensional property, and indeed many important properties of programs can be stated as extensional properties. Examples include: lack of runtime-errors, all exceptions are caught, even reachability of a particular program point. While not every useful program property can be represented in this way, it is nonetheless unfortunate that one of the earliest results of computability show that extensional properties cannot be verified automatically:

Theorem 6.1 (Rice's Theorem)**.** *The only decidable extensional properties of programs are the trivial properties* ***true*** *and* ***false****.*

An important consequence of Rice's theorem is that any static analysis will give different results on semantically equivalent programs. In our setting, for any analysis there will be two equivalent programs P and P' such that P is found terminating but P' is not (in fact, it is easy to see that there must be infinitely many such pairs). The remainder of this chapter will be concerned with trying to evaluate the impact of this undesirable behaviour.

6.1.2 Decidable Termination Criteria

In this chapter, we shall be concerned with evaluating the expressiveness of our size-change termination analyses. It is therefore useful to briefly describe well-known termination criteria. We shall focus on *typing disciplines* that guarantee termination, as the set of well-typed programs (with respect to any given type system) is particularly easy to describe.

Primitive Recursion One of the earliest classes of terminating programs to be studied is the class of *primitive recursive* functions. This intends to encode simple recursive definitions of the form:

$$\begin{aligned} f(0) &= C \\ f(n+1) &= H(n, f(n)) \end{aligned}$$

This is usually defined for first-order programs. In our setting, we might define a program to be primitive recursive if for each function f with parameters $x_1, \ldots, x_n$, there is a parameter x_i such that each recursive call to f (*i.e.* call to f from the body of f) is of the form

$$f\ x_1\ \cdots\ x_i\ \cdots\ x_n \quad \rightarrow \quad f\ x_1\ \cdots\ (x_i - 1)\ \cdots\ x_n$$

That is, recursion is always controlled by the x_i parameter, and other parameters are fixed. For simplicity we ban mutual recursion. The idea of primitive recursion over natural numbers can be extended to other datatypes (for instance, *foldr* encodes primitive recursion over lists).

It is trivial that first-order primitive recursive programs are necessarily terminating. This does not hold in the higher-order case, unless it is further assumed that programs are well-typed (in some appropriate type system, such as the Hindley-Milner type system of ML [Mil78]). Furthermore, a first-order primitive recursive program is size-change terminating.

Simple Types While primitive recursion controls recursion in first-order programs, techniques for controlling termination of higher-order programs are of more immediate interest to evaluate our analyses. We shall therefore focus on types for the λ-calculus, starting with the simplest such system, simple types.

The set of simply-typed λ-expressions (assuming no constants for simplicity) can be defined as follows. The set of *types* consists of O (the *ground* type), together with types $T \rightarrow U$ whenever T and U are types. The type $T \rightarrow U$ of course denotes the type of functions from T to U.

The idea of simple types is straightforward: a type is assigned to each variable. A term $\lambda x.e$ has type $U \rightarrow V$, where x has type U and e has type V, while an application $e_1 e_2$ requires the type of e_1 to be of the form $U \rightarrow V$, where e_2 has type U. Membership of a given λ-expression in the class of simply-typed λ-expressions is decidable (that is, type inference for simple types is decidable).

Our interest for the simply-typed λ-calculus lies in the following result [Tai67, Gan80]:

Theorem 6.2. *The simply-typed λ-calculus is* strongly normalising. *That is, if e is a simply-typed λ-expression, then any reduction sequence starting at e is finite.*

In particular, a simply-typed λ-expression e is terminating both under call-by-name and call-by-value.

System T System T, due to Gödel, extends the simply-typed λ-calculus with types and values for natural numbers and booleans. Furthermore, System T introduces a *recursion* operator R. We may view System T as introducing *primitive recursion*, over natural numbers, to the simply-typed λ-calculus.

System T introduces the following constructs. First, we add constants O (zero), $\mathsf{S}\ t$ where t is a term (successor), and T and F (booleans). Furthermore, an **if**-like construct $\mathsf{D}\ u\ v\ t$ (decide between u and v, based on the value of t) is added. Finally, recursion is encoded by the term $\mathsf{R}\ u\ v\ t$. Its intended meaning is: $\mathsf{R}\ u\ v\ \mathsf{O} = u$, while $\mathsf{R}\ u\ v\ (\mathsf{S}\ t) = v\ (\mathsf{R}\ u\ v\ t)\ t$. That is, the first parameter provides the base case, the second provides the step function, while the final parameter is the recursion variable.

In addition, System T introduces the type Nat and Bool to describe natural numbers and booleans. The type rules are as expected (for instance, $\mathsf{S}\ t$ is only defined if $t : \mathsf{Nat}$, and itself has type Nat).

As system T extends the simply-typed λ-calculus with primitive recursion, the following does not come as a surprise:

Theorem 6.3. *System T is strongly normalising.*

As before, type inference for System T is decidable.

System F The final type system that we shall consider is System F, independently due to Girard [Gir71] and Reynolds [Rey74]. This is a very expressive type system, which is substantially stronger than simple types. Indeed, the Hindley-Milner type system is weaker than System F.

The intention of System F is to encode *polymorphism*, as found in functional languages. As an example, consider the identity function *id*. In simple types, this can be given any type of the form $T \to T$ — there is a distinct identity function for each such type. Using polymorphic types, the identity function can be given a proper type of the form $\forall\alpha.\alpha \to \alpha$.

Polymorphism in System F goes beyond what languages such as ML offer, however. In ML a polymorphic type is of the form $\forall\overrightarrow{\alpha}.T$, where T is a type scheme (essentially a simple type with type variables $\overrightarrow{\alpha}$). That is, quantification over type variables happens at the outer level only. In contrast, in System F one can write a type of the form:

$$(\forall\alpha.\alpha \to \alpha) \to (\forall\alpha.\alpha \to \alpha)$$

This type represents a function that expects a *polymorphic* function, and returns a polymorphic function of the same type. This cannot be expressed in the Hindley-Milner type system. A consequence of this fact is that in System F, unlike the Hindley-Milner type system, a term t need not have a *principal type* (type from which all other possible types for t can be derived).

More precisely, System F adds two constructs to simple types. First, on the type level, types of the form $\forall\alpha.T$ are introduced. On the term level, the construct $\Lambda X.t$ forms a term of polymorphic type $\forall\alpha.T$, where t has type T and α is not free in the type of a free variable of t. Dually to this type abstraction construct, a type application of the form $t\ U$ (where t is a term of polymorphic type) instantiates a polymorphic type $\forall\alpha.T$ to a particular value U of α.

System F is of interest to us once more through its termination properties, due to the following highly nontrivial result:

Theorem 6.4. *System F is strongly normalising.*

System F represents a kind of golden standard for type systems guaranteeing termination. It is more powerful than common type systems for functional languages (Hindley-Milner, and extensions of it as found in Haskell), and therefore models extremely expressive functional languages. In a sense, System F is *too* powerful, however: type inference for System F is not decidable [Wel99].

6.1.3 Intensional and Extensional Characterisations

Given a program analysis φ, we may consider the following characterisations of the effectiveness of φ:

1. The *intensional* characterisation: the set of programs accepted by φ, and
2. The *extensional* characterisation: the set of *functions* accepted by φ. This can be defined as $\{[\![P]\!] \mid \varphi(P)\}$.

We shall be more interested in evaluating the *intensional* expressiveness of static analyses. To understand why, let us briefly review existing results on extensional expressiveness of program safety criteria.

For simplicity, we shall only be concerned with programs whose denotation is a *first-order* function, say of type $\mathbb{N} \to \mathbb{N}$. Intermediate results in the computation may be higher-order, but we restrict externally observable behaviour to this first-order type. We therefore only consider the question of determining the set of functions $f : \mathbb{N} \to \mathbb{N}$ that can be expressed as a program P with the relevant property.

Primitive Recursion The set of functions expressible by (first-order) primitive recursive programs is known as the set of *primitive recursive functions*. This is a fairly large class of functions, including many common arithmetic functions such as polynomials or exponentials.

However, primitive recursive functions do not include all terminating computable functions (indeed, as the property of being primitive recursive is a decidable intensional property, this would be impossible). Concretely, it is possible to prove the following [Ack77]: if $f : \mathbb{N} \to \mathbb{N}$ is primitive recursive, then there exists $m \in \mathbb{N}$ such that for all x, $f(x) < ack\, m\, x$. In particular, Ackermann's function itself is not primitive recursive.

First-Order Size-Change Termination We have previously remarked that the set of first-order SCT programs includes the set of primitive recursive programs. It naturally follows that the extensional expressive power of (first-order) SCT is at least as high as that of primitive recursion.

In fact, the set of functions expressible in first-order SCT is strictly larger. It corresponds to Péter's *multiple recursive functions* [Pét69], as shown by Ben-Amram [BA02]. This class of functions generalises primitive recursive functions by allowing recursion on several variables, and indeed nested recursive calls, as long as recursive calls exhibit *lexicographic descent*. The archetypal example of a multiple-recursive (but not primitive-recursive) function is Ackermann's function.

Simple Types Let us now consider the expressive power of typed λ-calculi, restricted to integer functions $\mathbb{N} \to \mathbb{N}$ (or, more generally, $\mathbb{N}^k \to \mathbb{N}$). Of course, the simply-typed λ-calculus as we have introduced it does not contain constructs for numbers, and therefore we must pick an *encoding* assigning a term $\overline{n}$ to each natural number n. A standard choice is to use *Church numerals*:

$$\overline{n} = \lambda f.\lambda x.f^n x$$

With this encoding[1], we may characterise the expressive power of the simply-typed λ-calculus for arithmetic functions.

As may perhaps be expected from the fact that the simply-typed λ-calculus does not provide any recursion operator (fixpoint combinators such as Y are of course not expressible, due to normalisation properties), this is rather low. In fact, the only functions expressible in this way are the so-called *extended polynomials* [Sch76]: essentially polynomials of several variables with natural number coefficients, together with the *sign function sg* defined by $sg(0) = 0$ and $sg(n + 1) = 1$. The simply-typed λ-calculus is therefore quite inadequate for expressing useful functions.

System T The aim of System T is specifically to express nontrivial integer functions, and therefore it should come as no surprise that its expressive power is rather higher than simple types. However, as the only recursion operator introduced corresponds to primitive recursion, it might appear that only primitive recursive functions are expressible in System T.

[1] A technicality: we assume that all Church numerals are given type $\mathbf{int} = (O \to O) \to O \to O$

In fact, the use of primitive recursion, *with* higher-order values, allows us to express many more functions (including Ackermann's function). As a result, the expressive power of System T is extremely high: *any* total computable function f, which is *provably* total in Peano Arithmetic (PA), is expressible in System T [GLT89, 7.4]. Of course, not every total function is provably total in any given system (by Gödel's incompleteness theorem), but the set of PA-provable total functions is very large indeed.

System F Finally, let us conclude our discussion of extensional characterisations by considering the "golden standard", System F. As this is strictly more powerful than System T, it is gratifying to note that its extensional expressive power is higher. In fact, System F can express all functions that are provably total in *second-order* Peano arithmetic (second-order refers to the possibility of quantifying over predicates in this context). The class of functions provably total in second-order Peano arithmetic arguably includes any function which we may need to represent.

Problems with Extensional Characterisations

As we have shown in the above, very precise characterisations of the extensive expressiveness of type disciplines are known. However, we argue that these are not practically useful as a benchmark for the expressiveness of such programming disciplines.

The fundamental problem with extensional characterisations is the following. Suppose that φ is a program property, and that f is representable by a program P with property φ. As we have seen before, it is not usually the case that *all* programs expressing function f have property φ. In particular, the fact that f is representable does not mean that the *most natural* program for f satisfies φ. Furthermore, in some cases, although a function is representable, there is no *efficient* program expressing the program in a given form.

As a consequence, we can note the following unsatisfactory aspects of the above characterisation:

- The set of primitive recursive functions is reasonably substantial, but primitive recursion is overly restrictive as a programming discipline.
- To an even greater extent, the characterisation of System T representable functions is misleading: System T is not an adequate system for writing programs, though the set of functions that *can* be written in System T is large.

In contrast, in evaluating the expressiveness of static analyses, we are concerned with the set of *programs* that can be accepted — as static analyses behave differently on semantically equivalent programs, this is a much more fine-grained metric. We shall therefore focus on (approximate) *intensional* characterisations in the remainder of this chapter.

6.2 Characterising Precision of Static Analysis

Having set the problem of evaluating the expressiveness of static analyses, let us now turn to the special case of size-change termination. In this section, we shall be concerned with the set of *false negatives* of SCT-based analyses: the (undesirable, but unavoidable) cases of programs P which are terminating, but are not recognised as such by the analysis.

In particular, we shall be interested in answering the following questions:

1. Given a terminating program P, does there exist a static analysis for SCT that succeeds in detecting termination of P?
2. Given a terminating program P, does there exist a static analysis for SCT, with fixed abstract domain, that detects P as terminating?
3. Is a given analysis (say, 0CFA) as precise as it could be, given its abstract domain?

To give meaningful answers to these questions, we must clarify the causes of failure of the analysis on terminating programs.

6.2.1 The Perfect Static Call Graph

Fix a program P, possibly with free template variables. Then for each valuation $\mathfrak{V}$, there is a dynamic call graph (say $\mathsf{G}_{\mathfrak{V}}$), defined by the reduction and call relations $\Downarrow$ and $\rightarrow$. For notational simplicity we shall only focus on the call relation $\rightarrow$ in what follows.

Suppose that (X, α) is the abstract domain of an SCT analysis, where $\alpha : \mathit{State} \rightarrow X$ is the abstraction function. For any state s, define $\mathrm{gb}^{\alpha}(s) = \mathrm{gb}(\alpha(s))$. Finally, let $G^{\alpha}(s, s')$ be the most precise size-change graph with in-basis $\mathrm{gb}^{\alpha}(s)$ and out-basis $\mathrm{gb}^{\alpha}(s')$ that is safe for (s, s')

Then by the safety criterion, the graph

$$\mathsf{G}^{\alpha}_{\mathfrak{V}} = \{\alpha(s) \rightarrow^{\alpha} \alpha(s'), G^{\alpha}(s, s') \mid s \rightarrow s' \in \mathsf{G}_{\mathfrak{V}}\}$$

is at least as precise as any graph produced by a sound analysis with abstract domain (X, α).

Consequently, if $\mathfrak{W}$ is an abstract valuation, then the graph:

$$\mathsf{G}^{\alpha}_{\mathfrak{W}} = \bigcup \{\mathsf{G}^{\alpha}_{\mathfrak{V}} \mid \mathfrak{V} \text{ is compatible with } \mathfrak{W}\}$$

is a subset of any safe call graph for P, given the abstract domain and valuation. We call $\mathsf{G}^{\alpha}_{\mathfrak{W}}$ the *perfect* static call graph with respect to (X, α) and $\mathfrak{W}$.

The perfect static call graph represents an ideal standard for the precision of any analysis with given abstract domain. While the perfect static call graph is of course an approximation to the dynamic call graph, the call graph obtained (in a computable fashion) by the analysis may be strictly less precise than the perfect call graph. We shall return to this point later.

6.2.2 Nontermination in the Perfect Call Graph

Given a safe static call graph, the SCT algorithm can be used to find whether the program is size-change terminating (with respect to the given call graph). It is easy to give a useful characterisation of failure of the SCT criterion, based on Theorem 3.43:

Lemma 6.5. *Let γ be a size-change graph. Then for some $n > 0$, γ^n is idempotent. We write $\overline{\gamma}$ for the idempotent power of γ; this is well-defined.*

Proof. As there are only finitely many size-change graphs, the sequence γ, γ^2, γ^3, ... is finite. Hence for some $1 \leq i < j$, $\gamma^i = \gamma^j$. Then $\gamma^{i+a} = \gamma^{i+b}$ whenever $a \equiv b \pmod{j-i}$, as $\gamma^{i+a+(j-i)} = \gamma^{j+a} = \gamma^j\gamma^a = \gamma^i\gamma^a = \gamma^{i+a}$ for all a. Pick $a \equiv -i \pmod{j-i}$, with $a > 0$. Then $(\gamma^{i+a})^2 = \gamma^{2i+2a} = \gamma^{i+(i+2a)} = \gamma^{i+a}$ as $i + 2a \equiv a \pmod{j-i}$, so γ^{i+a} is idempotent.

Furthermore suppose that γ^n and γ^m are idempotent powers of γ, with $n, m > 0$. Then $\gamma^n = (\gamma^n)^m = \gamma^{nm} = (\gamma^m)^n = \gamma^m$, so that $\overline{\gamma}$ is well-defined. □

The size-change termination principle then amounts to the following:

Lemma 6.6. *Consider an annotated static call graph. Then this is* not *size-change terminating iff: there is a self-loop $s \to_\alpha^+ s$, where the size-change graph γ obtained by composing graphs along this sequence of calls is such that $\overline{\gamma}$ has no in-situ decrease.*

We will say a graph γ is *not decreasing* iff $\overline{\gamma}$ has no in-situ decrease.

Lemma 6.7. *Suppose that γ is not decreasing, and suppose that $\delta \subseteq \gamma$. Then δ is not decreasing.*

Proof. Let $\gamma^m = \overline{\gamma}$ and $\delta^n = \overline{\delta}$. Then $\overline{\delta} = (\overline{\delta})^m = \delta^{nm} \subseteq \gamma^{nm} = (\gamma^m)^n = (\overline{\gamma})^n = \overline{\gamma}$, as composition is monotonic. Hence if $\overline{\delta}$ had an in-situ decrease, so would $\overline{\gamma}$, a contradiction. □

We have claimed in the previous section that the perfect static call graph is the most precise static call graph, given the abstraction function. While this is intuitively clear, we must prove that nontermination in the perfect static call graph implies nontermination in any static call graph (with same abstraction function).

It is not immediate that this is indeed the case. For, a cycle in the perfect static call graph has the form $\alpha(s_0) \to \alpha(s_1) \to \cdots \to \alpha(s_n) = \alpha(s_0)$. By soundness, there must be a corresponding call sequence $t_0 \to t_1 \to \cdots \to t_n$ in any safe call graph, but we cannot deduce that $t_n = t_0$. In fact this is sufficient, as we shall show below.

The proof relies on the following property of the static call graph:

Property 6.8 (Monotonicity)**.** *A static call graph is* monotonic *if:*

1. *Whenever $s \to s'$ and $t \sqsupseteq s$, there exists $t' \sqsupseteq s'$ such that $t \to t'$; and*
2. *In the above, for any $\gamma \in G(s \to s')$ there exists $\delta \in G(t \to t')$ such that $\gamma \supseteq \delta$.*

The monotonicity property is easily seen to hold of the analyses described in Chapter 4, and can be considered a requirement of any SCT static analysis. The upshot of the monotonicity property is that we can deduce nontermination in a situation slightly more general that that of Lemma 6.6, as required to argue that any safe call graph is nonterminating if the perfect call graph is.

Lemma 6.9. *Consider a monotonic static call graph. A* subcycle *is a sequence of calls* $s_0 \to s_1 \to \cdots \to s_n$*, where* $s_n \sqsupseteq s_0$*. A subcycle is* nondecreasing *if for some* $\gamma_i \in G(s_i \to s_{i+1})$*, the graph* $\gamma_0; \cdots ; \gamma_{n-1}$ *is nondecreasing. Then if there exists a nondecreasing subcycle in the call graph, the call graph is not size-change terminating.*

Proof. By Lemma 6.6, it suffices to show the existence of a nondecreasing cycle in the call graph.

Suppose that $s_0^i \to s_1^i \to \cdots \to s_n^i$ is a nondecreasing subcycle. Then we can construct a subcycle $s_0^{i+1} \to \cdots \to s_n^{i+1}$ as follows. Set $s_0^{i+1} = s_n^i \sqsupseteq s_0^i$. Then for each i, given $s_j^{i+1} \sqsupseteq s_j^i$, by monotonicity there exists $s_{j+1}^{i+1} \sqsupseteq s_{j+1}^i$ such that $s_j^{i+1} \to s_{j+1}^{i+1}$. This is a subcycle as $s_n^{i+1} \sqsupseteq s_n^i = s_0^{i+1}$.

Furthermore, pick $\delta_j^i \in G(s_j^i \to s_{j+1}^i)$ for each i, and suppose that $\delta_0^i; \cdots \delta_{n-1}^i$ is nondecreasing (this is given for $i = 0$). Then by monotonicity we may pick $\delta_j^{i+1} \in G(s_j^{i+1} \to s_{j+1}^{i+1})$ such that $\delta_j^{i+1} \subseteq \delta_j^i$. Hence $\delta_0^{i+1}; \cdots ; \delta_{n-1}^{i+1}$ is nondecreasing.

Now if for some i, $s_n^i = s_n^{i+1}$, then as $s_n^i = s_0^{i+1}$, the $(i+1)^{th}$ subcycle is a proper cycle, and as this is nondecreasing the call graph is size-change terminating.

Otherwise, for each i, $s_n^i \sqsubset s_n^{i+1}$. This gives an infinitely increasing chain of states, which is impossible as the set of abstract states is finite. This concludes the proof. □

The general tool for comparing termination in static call graphs is to show that a (more precise) call graph can *simulate* a less precise one. This is defined precisely below:

Definition 6.10 (Simulation)**.** Fix a program P, and consider two static call graphs for P, say $\mathcal{A}$ over abstract domain A defining judgements $\Downarrow^A$ and $\to^A$; and $\mathcal{B}$ over B, defining $\Downarrow^B$ and $\to^B$.

A map $f : A \to B$ is a *simulation* if:

1. Whenever $s \Downarrow^A v$ (resp. $s \to^A s'$) and $t \sqsupseteq f(s)$, then there exists $w \sqsupseteq f(v)$ such that $t \Downarrow^B w$ (resp. $t \to^B t'$, where $t' \sqsupseteq f(s')$).

2. In the above, for any $\gamma \in G(s \Downarrow^A v)$ there exists $\delta \in G(t \Downarrow^B w)$ such that $\gamma \supseteq \delta$ (and likewise for call edges).

Proposition 6.11. *If* f *is a simulation from* $\mathcal{A}$ *to* $\mathcal{B}$*, and if* $\mathcal{B}$ *is size-change terminating, then so is* $\mathcal{A}$*.*

Proof. Suppose that $\mathcal{A}$ is not size-change terminating; we show that neither is $\mathcal{B}$. As $\mathcal{A}$ is not SCT, there is a sequence of calls $t_0 \to^A t_1 \to^A \cdots \to^A t_n = t_0$, with $\gamma = \gamma_0; \cdots ; \gamma_{n-1}$ (for some $\gamma_i \in G(t_i \to^A t_{i+1})$) nondecreasing.

We can therefore construct a sequence of states $s_0 = f(t_0)$ and $s_i \sqsupseteq f(t_i)$ (for each i) such that $s_0 \to^B \cdots \to^B s_n \sqsupseteq s_0$.

Furthermore for each i there exists $\delta_i \in G(s_i \to^B s_{i+1})$ such that $\delta_i \subseteq \gamma_i$. Hence $\delta_0; \cdots; \delta_{n-1} \subseteq \gamma_1; \cdots; \gamma_{n-1} = \gamma$ is nondecreasing. We have therefore exhibited a nondecreasing subcycle, concluding the proof by the previous lemma. □

Let us conclude that the perfect static call graph is indeed as precise as any safe call graph:

Proposition 6.12. *If the perfect call graph for P with respect to α is nonterminating, then so is any safe call graph with abstraction function α (with the same valuation $\mathfrak{W}$).*

Proof. Suppose that the static call graph is nonterminating. Then by Lemma 6.6 there is a sequence of calls $\alpha(s_0) \to \alpha(s_1) \to \cdots \to \alpha(s_n) = \alpha(s_0)$, where if $\gamma_i = G^\alpha(s_i, s_{i+1})$, then $\gamma = \gamma_0; \cdots; \gamma_{n-1}$ is not decreasing.

By safety, given the sequence of calls $s_0 \to s_1 \to \cdots \to s_n$ in the dynamic call graph, there is a corresponding sequence of calls $t_0 \to t_1 \to \cdots \to t_n$ in the static call graph, where: $t_0 = \alpha(s_0)$, $t_i \sqsupseteq \alpha(s_i)$ for each i, and some $\delta_i \in G(t_i \to t_{i+1})$ is safe for (s_i, s_{i+1}) for each i.

Now $t_n \sqsupseteq \alpha(s_n) = \alpha(s_0) = t_0$, so that this is indeed a subcycle. Furthemore, some $\delta_i \in G(t_i \to t_{i+1})$ is safe for (s_i, s_{i+1}), so that as $G^\alpha(s_i, s_{i+1})$ is the maximal safe size-change graph for (s_i, s_{i+1}) with graph bases $\mathrm{gb}^\alpha(s_i)$ and $\mathrm{gb}^\alpha(s_{i+1})$, necessarily $\delta_i \subseteq \gamma_i$ for each i. Hence $\delta_0; \cdots; \delta_{n-1} \subseteq \gamma$ is nondecreasing, as required by Lemma 6.9. □

Nontermination Criteria

We can now describe our first criterion for detecting false negatives:

Proposition 6.13. *Suppose that for some dynamic call graph $\mathsf{G}_\mathfrak{V}$ there is a call sequence $s = s_0 \to s_1 \to \cdots \to s_n = s'$, such that $\alpha(s) = \alpha(s')$ and $G^\alpha(s_0, s_1); \cdots; G^\alpha(s_{n-1}, s_n)$ is not decreasing. Then the perfect static call graph $\mathsf{G}^\alpha_\mathfrak{W}$ (for any valuation $\mathfrak{W}$ describing $\mathfrak{V}$) is not size-change terminating.*

When this holds, *no* safe static call graph with abstraction function α can be size-change terminating. We call this case a (local) *abstract domain SCT failure* — the failure to detect termination is a consequence of the choice of abstract domain, rather than the actual analysis.

Proof. There is a corresponding sequence $t = t_0 \to^\alpha t_1 \to^\alpha \cdots \to^\alpha t_n = t'$, where $t_i = \alpha(s_i)$ for each i, in $\mathsf{G}^\alpha_\mathfrak{V} \subseteq \mathsf{G}^\alpha_\mathfrak{W}$. Furthermore, each edge $t_i \to^\alpha t_{i+1}$ is annotated with the single graph $G^\alpha(s_i, s_{i+1})$. By assumption, the composition of these graphs is not decreasing. Hence $\mathsf{G}^\alpha_\mathfrak{W}$ is not size-change terminating, as required. □

This criterion for SCT failure based on the abstract domain is *local*: it relies on examining size changes along a *sequence* of calls. This purely local observation can be a source of imprecision. For instance, consider the following artificial program:

```
f n = if n = 0 then 0 else g (n+1)

g n = f (n−2)
```

Then f calls itself indirectly, resulting in the sequence of calls $f(n) \to g(n+1) \to f(n-1)$ whenever $n > 0$. This does imply a decrease in n along each recursive call. Likewise, the recursive call from g to itself features a decrease in the value of n. However, this global decrease is a result of an *increase* in n (in the call $f \to g$), followed by a larger decrease. As a result, it is *not* detected by any size-change termination analysis, but as a global analysis of loops could detect the decrease in n, we wish to differentiate this type of failure of SCT.

This is the purpose of the following criterion:

Proposition 6.14. *Suppose that for some dynamic call graph $\mathsf{G}_{\mathfrak{V}}$ there is a call sequence $s \to^+ s'$, such that $\alpha(s) = \alpha(s')$ and $G^\alpha(s, s')$ is not decreasing. Then the perfect static call graph $\mathsf{G}^\alpha_{\mathfrak{W}}$ (for any valuation $\mathfrak{W}$ describing $\mathfrak{V}$) is not size-change terminating.*

When the above holds, we say there is a *global* abstract domain SCT failure. Clearly, if a given program exhibits global abstract domain failure, it also exhibits local abstract domain failure, but the converse does not hold, as the above example shows.

Proof. It suffices to show that global abstract domain failure implies local abstract domain failure. Let $s = s_0 \to s_1 \to \cdots \to s_n = s'$ be the call sequence, and $\delta_i = G^\alpha(s_i, s_{i+1})$. Then $\delta = \delta_0; \cdots; \delta_{n-1}$ is safe for (s, s'). As $G^\alpha(s, s')$ is the maximal graph safe for (s, s') (with given graph bases), $\delta \subseteq G^\alpha(s, s')$. Hence as $G^\alpha(s, s')$ is not decreasing by assumption, neither is δ, as required. □

6.2.3 Loss of Precision

The perfect static call graph encodes the *best* result that can be achieved by a static analysis with given abstraction function. It is obtained by taking the dynamic call graph and *a posteriori* applying the abstraction function. However, the dynamic call graph itself cannot be computed, and therefore this does not constitute an appropriate basis for defining a program analysis.

An implementable static analysis will instead compute the static call graph by working *within* the abstract domain. This leads to a loss of precision, however.

Example 6.15. Let us illustrate this with a simple example — arithmetic operations and the *odd / even* abstract domain with three values, O representing the set of odd numbers, E that of even numbers, and the top value $\top$ denoting any number.

It is straightforward to define safe approximations to arithmetic operations such as $+$ and $/$, satisfying the following condition (for instance for $+$): if $a \sqsupseteq \alpha(n)$ and $b \sqsupseteq \alpha(m)$, then $a +^\alpha b \sqsupseteq \alpha(n + m)$.

In fact, we can state a stronger condition that holds of $+$: $\alpha(n) +^{\alpha} \alpha(m) = \alpha(n+m)$ for all n and m. This encodes the odd/even rule for addition: if the parity of the summands are known, so is the parity of the sum.

Consider in contrast the case of the / operation. Then $\alpha(4/2) = \alpha(2) = E$, but $\alpha(4)/^{\alpha}\alpha(2) = E/^{\alpha}E = \top$. For, the result of dividing two even numbers may be odd or even.

These operations therefore exhibit different behaviours — in the case of $+$, working within the abstract domain does not cause any loss of precision, but precision is lost for /.

Similarly, a static analysis with given abstract domain α may not compute a call graph that is as precise as the perfect static call graph, and this may cause more failures.

We describe such failures as *abstract loss of precision*. More precisely, this is the case of any program P for which the perfect static call graph (for a fixed abstraction) is size-change terminating, but the call graph produced in the analysis is not.

Loss of Precision in the SCT Analysis

The problem of precision loss presented above also extends to the control-flow analyses that we have presented, and thus to our SCT analyses. As an example, consider the following program:

```
app f x = f x
ignore f x = ()

id x = x

omega x = omega x

app app id (app ignore omega ())
```

There are two occurrences of the *app* function in operator position, giving the following values for closure analysis:

Variable	Values	
f	$(\!\|app : \emptyset\|\!)$	$(\!\|ignore : \emptyset\|\!)$
x	$(\!\|id : \emptyset\|\!)$	$(\!\|omega : \emptyset\|\!)$

The information that is lost in the above is the dependency between f and x — in particular, x can only evaluate to the (dangerous) *omega* function if f is *ignore* (and so x is never called). As a result, this program terminates in the exact semantics. The perfect call graph for the 0CFA abstraction is shown in Figure 6.2 (with intermediate states elided).

The perfect 0CFA call graph is size-change terminating (as it is acyclic). However, the program is not size-change terminating with the computed 0CFA call graph, due to the lost

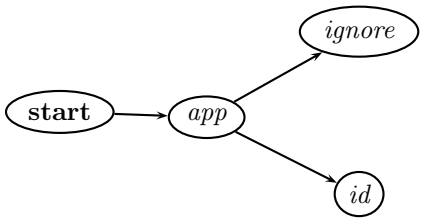


Figure 6.2: Perfect 0CFA Call Graph

dependency between f and x. In fact, *omega* is reachable in the computed 0CFA call graph, but not in the perfect 0CFA call graph.

6.2.4 Arbitrary SCT Analyses

Finally, let us complete this classification of SCT failures by introducing the problem of whether a program (or a set of programs) can be found terminating by *any* SCT analysis.

Let us first note that this is question is only interesting for *program templates*. For, a terminating closed program has a finite call graph, and we can therefore construct an abstraction function that acts as the identity on this finite graph. However, a program with free variables has a possibly unbounded set of call graphs when the assignment of values to variables is varied. As such it is not obvious whether or not there is a finite set of abstract states X and an abstraction function making the perfect static call graph terminating.

We shall make the following simplifying assumptions:

- We fix the abstract graph basis function gb^α. This will typically return the k-limited graph basis, for some fixed k, and
- We assume that free variables can be given any value. To this end we require that each abstract domain contain an element $*$ describing any value.

We then say that a program is *SCT-analysable* if for *some* abstraction function $\alpha : \mathit{State} \to X$, where X is a finite set, the perfect static call $\mathsf{G}^\alpha_{\mathfrak{W}}$ is size-change terminating, where $\mathfrak{W}$ is a valuation mapping each free variable to the arbitrary value $*$.

We could make a similar definition for *sets* of programs — a set of programs is analysable if there is an analysis that succeeds on all programs in this set (the analysis may not vary from program to program).

When can a program be shown to be non-analysable? We shall not attempt to give a precise characterisation of this, but give a sufficient condition, partly as an aid to intuition. The criterion that we shall use is straightforward: if, depending on the values of inputs, arbitrarily long call sequences without decreases are possible, no choice of analysis will prove termination.

Two states s and s' are *compatible* if:

1. s and s' occur at the same program point, and
2. $\mathrm{gb}^\alpha(s) = \mathrm{gb}^\alpha(s')$.

Let us say that a call sequence $s_0 \to s_1 \to \cdots \to s_n$ is *absolutely nondecreasing* if for any $0 \leq i < j \leq n$, provided that s_i and s_j are compatible, $G^\alpha(s_i, s_j)$ is nondecreasing.

Furthermore, we say that such a call sequence is *locally nondecreasing* if whenever i and j are as above, $G^\alpha(s_i, s_{i+1}); \cdots ; G^\alpha(s_{j-1}, s_j)$ is nondecreasing. As before, the absolute (global) nondecreasing property implies the local property.

We can then state the sufficient condition:

Proposition 6.16. *Suppose that for any M there is a valuation $\mathfrak{V}$ such that $\mathsf{G}_\mathfrak{V}$ contains a locally nondecreasing call sequence of length at least M. Then for any abstraction α, $\mathsf{G}^\alpha_\mathfrak{V}$ is not size-change terminating.*

Proof. Let $\alpha : \mathit{State} \to X$ be an abstraction function. Pick a call sequence $s_0 \to^+ s_n$ and valuation $\mathfrak{V}$ such that $n > |X| \times N$, where N is the size of the program. For each state s we let $\alpha'(s) = (\alpha(s), pp)$ where pp is the program point of s.

Then by the pigeonhole principle, for some $0 \leq i < j \leq n$, $\alpha'(s_i) = \alpha'(s_j)$. Then necessarily $\alpha(s_i) = \alpha(s_j)$, and s_i and s_j are compatible (since $\mathrm{gb}^\alpha(s_i) := \mathrm{gb}^\alpha(\alpha(s_i)) = \mathrm{gb}^\alpha(\alpha(s_j))$ $=: \mathrm{gb}^\alpha(s_j)$), so that by the local nondecreasing property, $G^\alpha(s_i, s_{i+1}); \cdots ; G^\alpha(s_{j-1}; s_j)$ is nondecreasing. It follows by Proposition 6.13 that $\mathsf{G}^\alpha_\mathfrak{W}$ is not size-change terminating, where as $\mathfrak{W}$ maps each free variable to $*$, $\mathfrak{W}$ is necessarily consistent with $\mathfrak{V}$. □

Such failure cases give an upper bound on what any SCT analysis can achieve. Programs which fail to be size-change terminating under any static analysis are beyond the scope of the SCT method and the size order used to prove termination.

6.3 Comparing Static Analyses

We have thus far introduced the different ways in which a terminating program can fail to be size-change terminating, both with respect to a particular static analysis and in the case of an arbitrary analysis.

In this section, we shall use this background to answer the question of the relationship between the different static analyses that we have previously introduced (0CFA, k-bounded CFA and tree automata). In particular, we shall be concerned with determining whether one analysis is more powerful than another (recognises a superset of the set of programs recognised by the weaker analysis).

6.3.1 The k-bounded CFA Hierarchy

In sections 4.4 and 4.5, we have described the k-bounded CFA family of analyses, starting with the important special case of 0CFA. We have informally described k-bounded CFA as a

tool to increase precision: increasing the value of k will make the class of accepted programs larger, at the cost of increased complexity. This is certainly plausible, given the construction of k-bounded CFA analyses. The aim of this section is to prove this.

More precisely, we shall let kCFA denote the set of programs accepted by the k-bounded CFA analysis[2]. We aim to prove correct the following k-bounded CFA hierarchy:

$$0\text{CFA} \subsetneq 1\text{CFA} \subsetneq 2\text{CFA} \subsetneq \cdots$$

We shall prove this in two stages. First, we shall show that $k\text{CFA} \subseteq (k+1)\text{CFA}$ for any k. Then, we shall for each k exhibit a program in $(k+1)\text{CFA} \setminus k\text{CFA}$.

Monotonic Increase: kCFA Containments

We first aim to show that for each k, $k\text{CFA} \subseteq (k+1)\text{CFA}$. Fix a program P. Then it suffices to show that: if P is found terminating by k-bounded CFA, it is found terminating by $(k+1)$-bounded CFA. It will be convenient to argue by the contrapositive, so that we shall show that if P is *not* $(k+1)$-bounded CFA terminating, it is not k-bounded CFA terminating.

The proof proceeds by showing that the k-bounded CFA call graph is less precise than the $(k+1)$-bounded CFA call graph. Specifically, the map $s \mapsto [s]_k$ defined in Section 4.5, taking a $(k+1)$-bounded CFA state to the corresponding k-bounded CFA state, is a simulation from the $(k+1)$-bounded CFA call graph to the k-bounded CFA call graph, whence the result follows by Proposition 6.11.

This result is stated below, where we write $\Downarrow^k$ and $\rightarrow^k$ for the evaluation and call judgements (respectively) for k-bounded CFA.

Lemma 6.17. *Suppose that $t \Downarrow^{k+1} w$ (resp. $t \rightarrow^{k+1} t'$), with size-change graph γ. Then whenever $s \sqsupseteq [t]_k$, there exists $v \sqsupseteq [w]_k$ (resp. $s' \sqsupseteq [t']_k$) such that $s \Downarrow^k v$ (resp. $s \rightarrow^k s'$), with size-change graph $\delta \subseteq [\gamma]_k$.*

Corollary 6.18. *kCFA $\subseteq (k+1)$CFA.*

The proof of Lemma 6.17 is surprisingly involved. While we shall not give full details, we outline the steps in the proof.

The first step is to show that any state reachable in k-bounded CFA is *closure analyis valid* (in the sense of Definition 4.21). That is, 0CFA closure analysis is a sound approximation of the successor relation in the tree defining such a state.

Definition 6.19. We extend the predicate $\text{Valid}_{\text{CA}}(s)$ to k-bounded CFA states by the following definition:

- The $k = 0$ case: $\text{Valid}_{\text{CA}}(e)$, $\text{Valid}_{\text{CA}}(\llparenthesis f : S \rrparenthesis)$, $\text{Valid}_{\text{CA}}(\mathsf{C})$, $\text{Valid}_{\text{CA}}(c)$ and $\text{Valid}_{\text{CA}}(*)$ all hold.

[2]We shall not refer to other notions of k-CFA in this chapter, and thus trust that there is no ambiguity.

- $\mathrm{Valid}_{\mathrm{CA}}(\langle\!|f : \rho|\!\rangle)$ and $\mathrm{Valid}_{\mathrm{CA}}(e : \rho)$ hold iff for all x, $\mathrm{Valid}_{\mathrm{CA}}(\rho(x))$ and $[\rho(x)]_0 \sqsubseteq v$ for some $v \in \mathrm{values}(x)$.
- $\mathrm{Valid}_{\mathrm{CA}}(\mathsf{C}(v_1, \ldots, v_n))$ holds iff for all i, $\mathrm{Valid}_{\mathrm{CA}}(v_i)$, and $[v_i]_0 \sqsubseteq v$ for some $v \in \mathrm{values}(\mathsf{C}, i)$.

Lemma 6.20. *Suppose that s is a reachable k-bounded CFA state. Then $Valid_{CA}(s)$ holds.*

Proof. The proof follows the proof of closure analysis validity of reachable (exact) states (Lemma 4.22) exactly, and we do not duplicate it here. It suffices to prove, by induction on the proof of $s \Downarrow^k v$, that: whenever $\mathrm{Valid}_{\mathrm{CA}}(s)$ and $s \Downarrow^k v$, then $\mathrm{Valid}_{\mathrm{CA}}(v)$ and $[s]_0 \Downarrow^0 w$ for some $w \sqsupseteq [v]_0$. □

We may deduce that the $\ll$ operation is a (non-functional) inverse of the restriction operator:

Corollary 6.21. *Suppose that $s \sqsupseteq [t]_k$, where t is a reachable $(k+1)$-bounded CFA state. Then $s \ll t'$ for some $t' \sqsupseteq t$.*

Proof. Let us prove this for the case $t = \langle\!|f : \rho|\!\rangle$, as other cases are similar, by induction on k. If $s = *$ then the result is trivial.

For the base case, take $k = 0$. Then wlog $s = \langle\!|f : S|\!\rangle$ where $S = \mathrm{dom}\,\rho$. Now as t is reachable, by Lemma 6.20 $\mathrm{Valid}_{\mathrm{CA}}(t)$ holds. Hence for each $x \in S$, $\rho(x) = [\rho(x)]_0 \sqsubseteq v_x$ for some $v_x \in \mathrm{values}(x)$. Put $t' = \langle\!|f : \{x \mapsto v_x\}_{x \in S}|\!\rangle$. Then certainly $s \ll t'$. Furthermore, $t' \sqsupseteq t$, as required.

For the inductive step, $s = \langle\!|f : \mu|\!\rangle$, where $\mu(x) \sqsupseteq [\rho(x)]_{k-1}$ for all $x \in S$. By the inductive hypothesis, for each x there exists $v_x \sqsupseteq \rho(x)$ such that $\mu(x) \ll v_x$. Then $s \ll t' := \langle\!|f : \{x \mapsto v_x\}_{x \in S}|\!\rangle$, and $t' \sqsupseteq t$, as required. □

Corollary 6.22. *Suppose that $s \ll t$, and that $s' \sqsupseteq [s]_{k-1}$. Then there exists $t' \sqsupseteq [t]_k$ such that $s' \ll t'$.*

Proof. By Corollary 6.21, $s' \ll t'$ for some $t' \sqsupseteq s$. But $[t]_k = s$, as required. □

The significance of Corollary 6.22 is to show that the $\ll$ operator (used to approximate the values of variables in k-bounded CFA) can be simulated — the results of $\ll$ at depth $k+1$ are reflected at depth k.

It is now possible to complete the proof of Lemma 6.17.

Proof of 6.17. The proof is largely a straightforward induction on proof trees. We shall deal with two nontrivial cases, and spare the reader details of easy cases. The two interesting cases, as ever, are variable lookup and function application, as these affect the environment.

Variable Lookup Suppose that $t = x : \rho \Downarrow^{k+1} w$. Then $u := \rho(x) \ll w$. Now $[t]_k = x : \sigma$, where $\sigma(x) = [\rho(x)]_{k-1}$ for each x. Therefore as $s \sqsupseteq [t]_k$, $s = x : \mu$, where $\mu(x) \sqsupseteq [\rho(x)]_{k-1}$. Now $s \Downarrow^k v$ whenever $\mu(x) \ll v$. By Corollary 6.22, $\mu(x) \ll v$ for some $v \sqsupseteq [w]_k$, as required.

Function Application We now consider the function application case. Suppose that $t = e_1 e_2 : \rho \Downarrow^k w$, where $t_1 = e_1 : \rho \Downarrow^{k+1} (\!|f : \sigma|\!)$, $t_2 = e_2 : \rho \Downarrow^{k+1} u$, and $t_b = (\!|f : \sigma \oplus \{y \mapsto [u]_k\}|\!) \Downarrow^{k+1} w$ (for some y). Let $\sigma' = \sigma \oplus \{y \mapsto [u]_k\}$.

Let $s \sqsupseteq [t]_k$. Then $s = e_1 e_2 : \mu$, where $\mu(x) \sqsupseteq [\rho(x)]_{k-1}$ for each x. Let $t_1 = t_1 : \mu$. Then $t_1 \sqsupseteq [s_1]_k$, so by the inductive hypothesis $t_1 \Downarrow^k a$ for some $a \sqsupseteq [(\!|f : \sigma|\!)]_k$. That is, $a = (\!|f : \tau|\!)$, where for each x, $\tau(x) \sqsupseteq [\sigma(x)]_{k-1}$.

Similarly, $s_2 = t_2 : \mu \Downarrow^k u' \sqsupseteq [u]_k$. Now put $s_b = (\!|f : \tau \oplus \{y \mapsto [u']_{k-1}\}|\!)$. Let $\tau' = \tau \oplus \{y \mapsto [u']_{k-1}\}$.

For each $x \in \operatorname{dom} \tau = \operatorname{dom} \sigma$, $\tau'(x) = \tau(x) \sqsupseteq [\sigma(x)]_{k-1} = [\sigma'(x)]_{k-1}$ as before. Now $\tau'(y) = [u']_{k-1}$, where $u' \sqsupseteq [u]_k$. Pick v' such that $u' \ll v'$ and $v' \sqsupseteq u$ (by Corollary 6.21). Then $u' = [v']_k$. Hence $[u']_{k-1} = [[v']_k]_{k-1} = [v']_{k-1}$. Furthermore, $[\sigma'(y)]_{k-1} = [[u]_k]_{k-1} = [u]_{k-1}$. As $v' \sqsupseteq u$, and restriction is trivially monotonic, we may deduce that $\tau'(y) \sqsupseteq [\sigma'(y)]_{k-1}$.

Hence $s_b \sqsupseteq [t_b]_k$. We may thus apply the inductive hypothesis to deduce that $t_b \Downarrow^k v \sqsupseteq [w]_k$, as required, as then $t \Downarrow^k v$. □

Increased Precision: $(k+1)$CFA and kCFA

A Simply-Typed λ-Expression in 1CFA $\setminus$ 0CFA Let us first consider, as a motivating example, a program which lies in 1CFA but not 0CFA. In fact (as in the next section) we shall use *simply-typed λ-expressions* as examples. Termination is of course guaranteed by strong normalisation, and as an added benefit we can show that k-bounded CFA analyses cannot account for all simply-typed λ-expressions[3].

Consider the expression:

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

This is simply-typed. The principal types of variables in this expression are (with type variables τ and μ):

$$\begin{aligned} a &: ((\tau \to \tau) \to \mu \to \mu) \to \mu \to \mu \\ b &: \tau \to \tau \\ c &: \tau \to \tau \\ d &: \mu \\ e &: (\tau \to \tau) \to (\mu \to \mu) \\ f &: \tau \end{aligned}$$

After λ-lifting, this expression yields the following program:

```
D d = d
```

[3]Settling, in the negative, a conjecture of Jones [JB04].

```
C c = D
B a b = a C
A a = a (B a)

F f = f
E e = e F

A E
```

We shall show that this program is not 0CFA-terminating. This is an *abstract domain* failure (Propositions 6.13 and 6.14), and thus is not due to imprecision in the static call graph.

To prove this, consider the following call sequence in the dynamic call graph of Λ_0 (eliding some intermediate states, for concision):

$$
\begin{array}{lll}
 & \textbf{start} & (1) \\
\rightarrow & (\!|A : \{a \mapsto (\!|E : \emptyset|\!)\}|\!) & (2) \\
\rightarrow & (\!|E : \{e \mapsto (\!|B : \{a \mapsto (\!|E : \emptyset|\!)\}|\!)\}|\!) & (3) \\
\rightarrow & (\!|B : \{a \mapsto (\!|E : \emptyset|\!), b \mapsto (\!|F : \emptyset|\!)\}|\!) & (4) \\
\rightarrow & (\!|E : \{e \mapsto (\!|C : \emptyset|\!)\}|\!) & (5) \\
\rightarrow & (\!|C : \{c \mapsto (\!|F : \emptyset|\!)\}|\!) & (6) \\
\rightarrow & (\!|D : \emptyset|\!) & (7)
\end{array}
$$

where the labels identify each state in the execution — we write s_i for the i^{th} state. In particular, consider the calls $s_3 \rightarrow s_4 \rightarrow s_5$. Then $\alpha_0(s_3) = \alpha_0(s_5) = (\!|E : \{e\}|\!)$, so that this gives rise to a self-loop in any 0CFA analysis.

Furthermore, the (1-limited) graph $G^{\alpha}(s_3, s_5)$ is empty: the values of e in s_3 and s_5 are incomparable[4].

As a result, we may apply Proposition 6.14 to conclude that *no* analysis with the same abstract domain as 0CFA can prove termination of Λ_0.

However, Λ_0 is terminating with 1-bounded CFA. To see that this is the case, let us first give the results of closure analysis (Figure 6.3). It is a property of λ-lifted programs that multiple copies of the same variable have the same values under closure analysis, so we only list each variable once.

The resulting 1-bounded CFA call graph is exactly what is obtained from applying the abstraction function α_1 to the dynamic call graph (that is, the 1-bounded CFA call graph is perfect given the abstraction function). In particular, the call sequence shown above maps

[4] In the analysis of Jones and Bohr [JB04] these are comparable, with the value of e in s_5 less than that in s_3. This does not affect the result, but in that case we must consider local, rather than global, size changes.

Variable	Value
a	$(\!\|E : \emptyset\|\!)$
b	$(\!\|F : \emptyset\|\!)$
c	$(\!\|F : \emptyset\|\!)$
d	
e	$(\!\|B : \{a \mapsto (\!\|E : \emptyset\|\!)\}\|\!) \mid (\!\|C : \emptyset\|\!)$
f	

Figure 6.3: Closure Analysis for Λ_0

to:

$$
\begin{array}{lll}
 & \textbf{start} & (1) \\
\rightarrow & (\!|A : \{a \mapsto (\!|E : \emptyset|\!)\}|\!) & (2) \\
\rightarrow & (\!|E : \{e \mapsto (\!|B : \{a\}|\!)\}|\!) & (3) \\
\rightarrow & (\!|B : \{a \mapsto (\!|E : \emptyset|\!), b \mapsto (\!|F : \emptyset|\!)\}|\!) & (4) \\
\rightarrow & (\!|E : \{e \mapsto (\!|C : \emptyset|\!)\}|\!) & (5) \\
\rightarrow & (\!|C : \{c \mapsto (\!|F : \emptyset|\!)\}|\!) & (6) \\
\rightarrow & (\!|D : \emptyset|\!) & (7)
\end{array}
$$

The only difference may be found on line 3, where the value to which a is bound is lost. However this does not affect precision, as there is only one value for a in closure analysis, and hence this can be recovered exactly. A consequence of this is that the 1-bounded CFA call graph for Λ_0 is acyclic, and thus trivially Λ_0 is terminating with 1-bounded CFA. Note that one level of information about the environment is enough to distinguish the problematic states s_3 and s_5 (even though the depth of s_3 is two).

The General Case: $(k+1)$CFA $\setminus$ kCFA The example of Λ_0 may be extended to a λ-expression Λ_k which fails to be size-change terminating under k-bounded CFA, but is terminating under $(k+1)$-bounded CFA (for any k). The strategy for this is to insert uses of the *apply* function $\lambda xy.xy$ to insert additional levels of indirection in environments.

For instance, we shall define Λ_1 from Λ_0 as follows:

$$\Lambda_1 = \Big(\lambda ga.a\big(g(\lambda b.a(g(\lambda cd.d)))\big)\Big)\Big(\lambda xy.xy\Big)\Big(\lambda e.e(\lambda f.f)\Big)$$

where we have inserted two calls to g (in the bodies of λa and λb respectively).

In general, define $h^k E$ by $h^0 E = E$ and $h^{k+1} E = h(h^k E)$. That is,

$$h^k E = \underbrace{h(\cdots(h}_{k \text{ times}}(E)\cdots)$$

We then define

$$\Lambda_k = \Big(\lambda ga.a\big(g^k(\lambda b.a(g^k(\lambda cd.d)))\big)\Big)\Big(\lambda xy.xy\Big)\Big(\lambda e.e(\lambda f.f)\Big)$$

for each k. Applying this with $k = 0$ yields an expression that is equivalent to, but slightly more complicated than, the previously defined Λ_0. We shall show that Λ_k lies in $(k+1)$CFA$\setminus$ kCFA.

The expression Λ_k is simply-typed, with the same types as before for variables a through f, and:

$$\begin{aligned} g &: ((\tau \to \tau) \to \mu \to \mu) \to (\tau \to \tau) \to \mu \to \mu \\ x &: (\tau \to \tau) \to \mu \to \mu \\ y &: \tau \to \tau \end{aligned}$$

In particular, the type level of Λ_k is the same as that of Λ_0 for each k. After λ-lifting, this becomes[5]:

```
D d = d
C c = D
B g a b = a (g^k C)
A g a = a (g^k (B g a))

Y x y = x y
X x = Y x

F f = f
E e = e F

A X E
```

First of all, let us show that Λ_k is not k-bounded CFA terminating. As before we argue by soundness, and so we must describe call sequences in the dynamic call graph of Λ_k.

For any value v, we let $\varphi_k(v)$ denote the state defined by:

$$\varphi_0(v) = v \text{ and } \varphi_{k+1}(v) = (\!|Y : x \mapsto \varphi_k(v)|\!)$$

That is, $\varphi_k(v)$ is state obtained by adding the apply function k times to the value v. Let us likewise define $\psi_k(v, w)$ to be the result of applying $\varphi_k(v)$ to w:

$$\psi_k(v, w) = (\!|Y : x \mapsto \varphi_{k-1}(v), y \mapsto w|\!) \text{ for all } k > 0$$

It is easy to show:

Lemma 6.23. *1. Whenever $E : \rho \Downarrow v$, and $\rho(g) = (\!|X : \emptyset|\!)$, then $g^k E : \rho \Downarrow \varphi_k(v)$.*

[5]We keep the convention that the function corresponding to an abstraction $\lambda v.e$ is named V (for each variable name v). It is unfortunate that this introduces a function named Y in this example, but we trust that there is no confusion with the paradoxical combinator Y.

2. *If* $e_1 : \rho \Downarrow \varphi_k(v)$ *and* $e_2 : \rho \Downarrow w$, *then* $e_1 e_2 : \rho \to \psi_k(v, w)$

3. $\psi_{k+1}(v, w) \to^+ \psi_k(v, w)$ *for all* $k \geq 1$.

4. $\psi_1(v, w) \to^+$ *Bind* v w

Furthermore, let us note that for any $l > k$, the values $\varphi_l(v)$ are identical up to depth l for any v. More precisely,

$$\alpha_k(\varphi_l(v)) = \varphi_{k-1}(\{\!|Y : \{x\}|\!\}) \text{ for any } v \text{ whenever } l > k$$

The spurious use of the application function Y therefore allows us to "confuse" k-bounded CFA analysis — for large enough l, $\varphi_l(v)$ and $\varphi_l(w)$ cannot be distinguished even if $v \neq w$.

We can deduce that the following call sequence occurs in the dynamic call graph of Λ_k:

	start	(1)
$\to$	$\{\!|A : \{a \mapsto \{\!|E : \emptyset|\!\}, g \mapsto \{\!|X : \emptyset|\!\}\}|\!\}$	(2)
$\to$	$\{\!|E : \{e \mapsto \varphi_k(\{\!|B : \{a \mapsto \{\!|E : \emptyset|\!\}, g \mapsto \{\!|X : \emptyset|\!\}\}|\!\})\}|\!\}$	(3)
$\to$	$\psi_k(\{\!|B : \{a \mapsto \{\!|E : \emptyset|\!\}, g \mapsto \{\!|X : \emptyset|\!\}\}|\!\}, \{\!|F : \emptyset|\!\})$	(4)
$\to^+$	$\{\!|B : \{a \mapsto \{\!|E : \emptyset|\!\}, g \mapsto \{\!|X : \emptyset|\!\}, b \mapsto \{\!|F : \emptyset|\!\}\}|\!\}$	(5)
$\to$	$\{\!|E : \{e \mapsto \varphi_k(\{\!|C : \emptyset|\!\})\}|\!\}$	(6)
$\to$	$\psi_k(\{\!|C : \emptyset|\!\}, \{\!|F : \emptyset|\!\})$	(7)
$\to^+$	$\{\!|C : \{c \mapsto \{\!|F : \emptyset|\!\}\}|\!\}$	(8)
$\to$	$\{\!|D : \emptyset|\!\}$	(9)

The reasoning to show that Λ_k is not k-bounded CFA terminating is now very similar to our reasoning in the case $k = 0$. We observe that $\alpha_k(s_3) = \alpha_k(s_6)$ — this follows by our previous observation that $\alpha_{k-1}(\varphi_k(v)) = \alpha_{k-1}(\varphi_k(w))$ for any v, w. This therefore gives rise to a cycle in the call graph.

Let us now show that this cycle cannot be size-change terminating. In fact, $G^\alpha(s_3, s_6) = \emptyset$ — there are no nontrivial size-change relationships, giving the required result. To prove this, we observe that the $((k+1)$-limited) graph bases of s_3 and s_6 have the following form:

$$\mathrm{gb}_{k+1}(s_3) = \mathrm{gb}_{k+1}(s_6) = \{\varepsilon\} \cup \{ex^l \mid 0 \leq l \leq k\}$$

Now for all i, $j \leq k$, we have:

$$s_3 \nabla ex^i = \varphi_{k-i}(\{\!|B : \{a \mapsto \{\!|E : \emptyset|\!\}\}|\!\}) \text{ and } s_6 \nabla ex^j = \varphi_{k-j}(\{\!|C : \emptyset|\!\})$$

Therefore $s_6 \nabla ex^j$ is not a subtree of $s_3 \nabla (ex^i)$ for any i, j, as the former contains a subtree $\{\!|C : \emptyset|\!\}$, while the latter does not. Likewise, $s_6 \nabla p$ is not a subtree of s_3 for any environment path p of length at most $k + 1$.

As $G^\alpha(s_3, s_6) = \emptyset$, we may conclude from Proposition 6.16 that Λ_k is not k-bounded CFA terminating.

It remains to show that Λ_k is $(k+1)$-bounded CFA terminating. The proof of this proceeds exactly as before: the call graph constructed in $(k+1)$-bounded CFA for Λ_k is acyclic. Full details of the $(k+1)$-bounded CFA call graph for Λ_k are rather tedious to give, so we shall merely note that:

- All states appearing in the call graph, with the exception of s_3 and s_4, have depth at most $k+1$, and thus no precision is lost in the construction of these states in the abstract interpretation
- As there is only one value for a in closure analysis, states s_3 and s_4 are recovered exactly, whence the $(k+1)$-bounded CFA call graph is exact.

We have therefore shown that for each k, Λ_k lies in $(k+1)\text{CFA} \setminus k\text{CFA}$.

The Cumulative k-bounded CFA Criterion

We have thus far shown that for every k, $k\text{CFA} \subseteq (k+1)\text{CFA}$, and that the containment is strict. In this section we shall consider the possibility of a static analysis subsuming every k-bounded CFA analysis, and show that this cannot be achieved.

Define the class

$$\exists k\text{CFA} = \bigcup_{k \geq 0} k\text{CFA}$$

Then our previous results show that

$$0\text{CFA} \subsetneq k\text{CFA} \subsetneq (k+1)\text{CFA} \subsetneq \exists k\text{CFA} \text{ for each } k$$

Of course, we do not have an effective algorithm for deciding membership in the class $\exists k$CFA — we cannot apply k-bounded CFA for all values of k.

In fact, membership in $\exists k$CFA is undecidable. Let us first note the following

Lemma 6.24. *Let P be a program obtained by λ-lifting from a terminating λ-expression. Then $P \in k\text{CFA}$ for some k.*

Proof. The proof in fact applies to any program without constants or free variables, but we shall not require this. We merely note that the dynamic call graph of P is finite, as P is terminating (and P is a closed program with no template variables). The depth of the states appearing in the dynamic call graph of P is therefore bounded, say by k.

Then the k-bounded CFA analysis of P is *exact*: the call graph obtained is the dynamic call graph. This can be checked by inspection on the rules for k-bounded CFA — if all states have depth at most k, then the restriction and $\ll$ operations reduce to the identity, and so k-bounded CFA is equivalent to the exact semantics (in the absence of constants). □

As a result, we may show the following:

Proposition 6.25. *If $\exists k\,CFA \subseteq S \subseteq \mathcal{T}$, where $\mathcal{T}$ is the set of terminating programs, then S is not decidable.*

In particular, $\exists k$CFA is not decidable. As any sound static termination analysis decides a subset of $\mathcal{T}$, we conclude:

Corollary 6.26. *No sound static termination analysis subsumes every k-bounded CFA size-change termination analysis.*

Proof of Proposition 6.25. If P decides $S \subseteq \mathcal{T}$, where $S \supseteq \exists k$CFA, then P decides termination for the λ-calculus: if e is a terminating λ-expression, then $e \in \exists k\text{CFA} \subseteq S$, and thus P accepts e. Furthermore, if e is a nonterminating λ-expression, then $e \notin \mathcal{T} \supseteq S$, so P rejects e.

As termination of (closed) λ-expressions is undecidable, this is a contradiction [Chu36]. □

6.3.2 Tree Automata

In this section we relate the expressiveness of the tree automata analysis to the k-bounded CFA hierarchy. We shall first show that the tree automata analysis is at least as expressive as 0CFA, but that it does not subsume 1-bounded CFA. Finally, we shall show that the tree automata analysis and the $\exists k$-CFA criterion are incomparable. Throughout this section we let TA denote the set of programs accepted by the tree automata analysis.

Tree Automata and k-bounded CFA

It should not come as a surprise that the tree automata analysis generalises the context-insensitive 0CFA analysis. The proof, which we shall not describe in detail, is analogous to the proof of soundness of 0CFA (in the same way as Lemma 6.20).

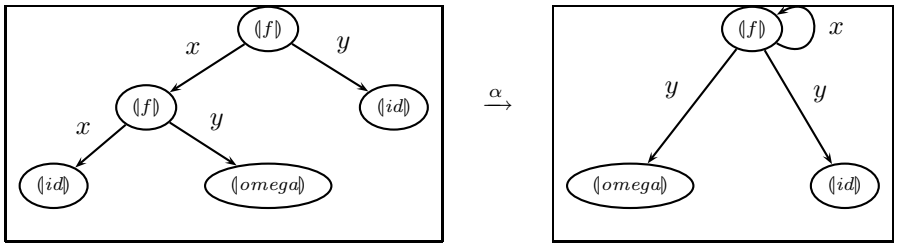
Lemma 6.27. *The map $\mathcal{A} \mapsto \zeta(\mathcal{A})$ is a simulation from the single-label tree automata call graph of a program to its 0CFA call graph.*

Corollary 6.28. *TA $\supseteq$ 0CFA.*

Proof. The proof of Lemma 6.27 proceeds by showing that all reachable states in the tree automata analysis are safely approximated by 0CFA.

We extend the $\text{Valid}_{\text{CA}}(s)$ predicate to automata in the expected fashion: $\text{Valid}_{\text{CA}}(\mathcal{A})$ holds iff whenever there is an edge $n \xrightarrow{x} m$ in $\mathcal{A}$, then $m \sqsubseteq v$ for some $v \in \text{values}(x)$. That is, any node bound to a variable x is described by closure analysis.

Define $\Downarrow^A$ and $\rightarrow^A$ to be the evaluation and call judgements in the automata analysis. The proof may then be completed by proving, by induction on the proof of $\Downarrow^A$ (resp. $\rightarrow^A$), that: whenever $\text{Valid}_{\text{CA}}(\mathcal{A})$ and $\mathcal{A} \Downarrow^A \mathcal{B}$ (resp. $\mathcal{A} \rightarrow^A \mathcal{B}$) then $\text{Valid}_{\text{CA}}(\mathcal{B})$, and $\zeta(\mathcal{A}) \Downarrow^0 b$



(1) (2)

Figure 6.4: Tree Automata and 1CFA Analyses

(resp. $\zeta(\mathcal{A}) \rightarrow^0 b$) for some $b \sqsupseteq \zeta(\mathcal{B})$. The proof of this property proceeds as the proof of soundness of 0CFA. □

While TA supersedes 0CFA, by Proposition 6.25 TA $\not\supseteq \exists k$CFA, whence for some k, TA $\not\supseteq k$CFA. In fact, as we shall show below, TA $\not\supseteq$ 1CFA.

Consider the following program:

```
id x = x
omega x = omega x

f x y z = y z

f (f id omega) id 1
```

Then this program is trivially terminating: the main reduction sequence is:

$$\textbf{start} \rightarrow \langle\!|f : \{x \mapsto \langle\!|f : \{x \mapsto \langle\!|id : \emptyset|\!\rangle, y \mapsto \langle\!|omega : \emptyset|\!\rangle\}|\!\rangle, y \mapsto \langle\!|id : \emptyset|\!\rangle\}|\!\rangle \rightarrow \langle\!|id : \{x \mapsto 1\}|\!\rangle \Downarrow 1$$

Furthermore, this program is terminating under 1-bounded CFA. The main reduction sequence is exactly the 1-limited abstraction of the above:

$$\textbf{start} \rightarrow \langle\!|f : \{x \mapsto \langle\!|f : \{x, y\}|\!\rangle, y \mapsto \langle\!|id : \emptyset|\!\rangle\}|\!\rangle \rightarrow \langle\!|id : \{x \mapsto \top_c\}|\!\rangle \Downarrow \top_c$$

and thus the call graph constructed by 1-bounded CFA is acyclic. This relies crucially on the observation that *omega* is not reachable — although it is passed to f as the y parameter, this occurs inside an expression bound to the x parameter of f, which is ignored. This program does not lie in 0CFA, however.

Furthermore, this program does not lie in TA. Consider the expression f (f *id omega*) *id*, evaluated in the empty environment (call this state s). In the exact semantics, $s \Downarrow v$, where v is the state shown in Figure 6.4 (1). By soundness, $\alpha(s) \Downarrow^A w$, where w is (a superstate of) the state shown in Figure 6.4 (2).

In particular, the value bound to y in w can be one of $(\!|id : \emptyset|\!)$ (the "right" value) or $(\!|omega : \emptyset|\!)$. This is a consequence of the sharing of the f nodes — the two distinct f nodes in v are merged, and thus the bindings for y are merged.

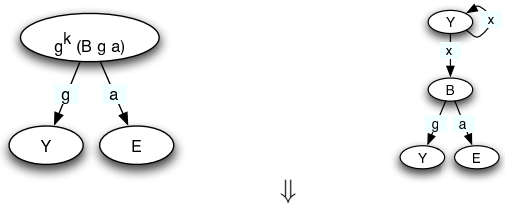
As a result, by soundness the static call graph must contain the call $\mathbf{start} \to Bind^{\alpha}\ w\ 1 = w'$ say. But w' is a (complete) f state with binding $x \mapsto (\!|omega : \emptyset|\!)$. Thus by soundness w' calls the *omega* function, which of course is nonterminating, resulting in a nonterminating verdict[6].

While this example is undoubtedly artificial, it serves as a minimal example of the fact that the merging of nodes in the single-label tree automata analysis can reduce precision, compared to 1-bounded CFA.

Tree Automata and kCFA We have thus far shown that 0CFA $\subseteq$ TA, and 1CFA $\not\subseteq$ TA (whence naturally kCFA $\not\subseteq$ TA for any $k > 1$). However, while the tree automata analysis cannot subsume every k-bounded CFA analysis, it does accept programs that are recognised by k-bounded CFA, but not $(k-1)$-bounded CFA, for each $k > 1$.

In fact, for each k, the expression Λ_k (or rather, the λ-lifted program obtained from it) lies in TA. As $\Lambda_k \in (k+1)\text{CFA} \setminus k\text{CFA}$, this yields the required result. Let us therefore show that Λ_k is found terminating by the tree automata analysis for each k. The expression Λ_k, together with the relevant parts of its dynamic call graph, can be found on page 170.

We assume that $k > 1$ — the base cases $k = 0$ and $k = 1$ can be treated individually. The crucial observation is that for any $k > 1$, the expression $g^k\ (B\ g\ a)$, in the body of A, always evaluates to the same state. This is shown below:



Likewise, the expression $g^k\ C$ in the body of B always evaluates to the same state (for any k). As a result, the call graph for Λ_k does not depend on k (for $k > 1$). Note that there are two occurrences of nodes for the Y function in the second automaton above. This does not contradict our statement that nodes with the same label are shared — the node label includes that number of bound parameters. The root node of this value corresponds to the 0CFA state $(\!|Y : \{x\}|\!)$, while the other occurrence corresponds to $(\!|Y : \emptyset|\!)$.

The call graph for Λ_k (for any $k > 1$) is then given in Figure 6.5 — intermediate states are, as usual, elided for conciseness.

The call graph of Figure 6.5 contains two cycles, one corresponding to the g^k application in the body of A, and the other to the g^k application in the body of B. Both of these cycles are

[6]This example uses the fact that the intermediate value w is constructed, and would fail in an extension of our analysis in which simultaneous applications are possible. In that case, however, we can transform the program into a program lying in 2CFA $\setminus$ TA by inserting spurious calls to the *apply* function.

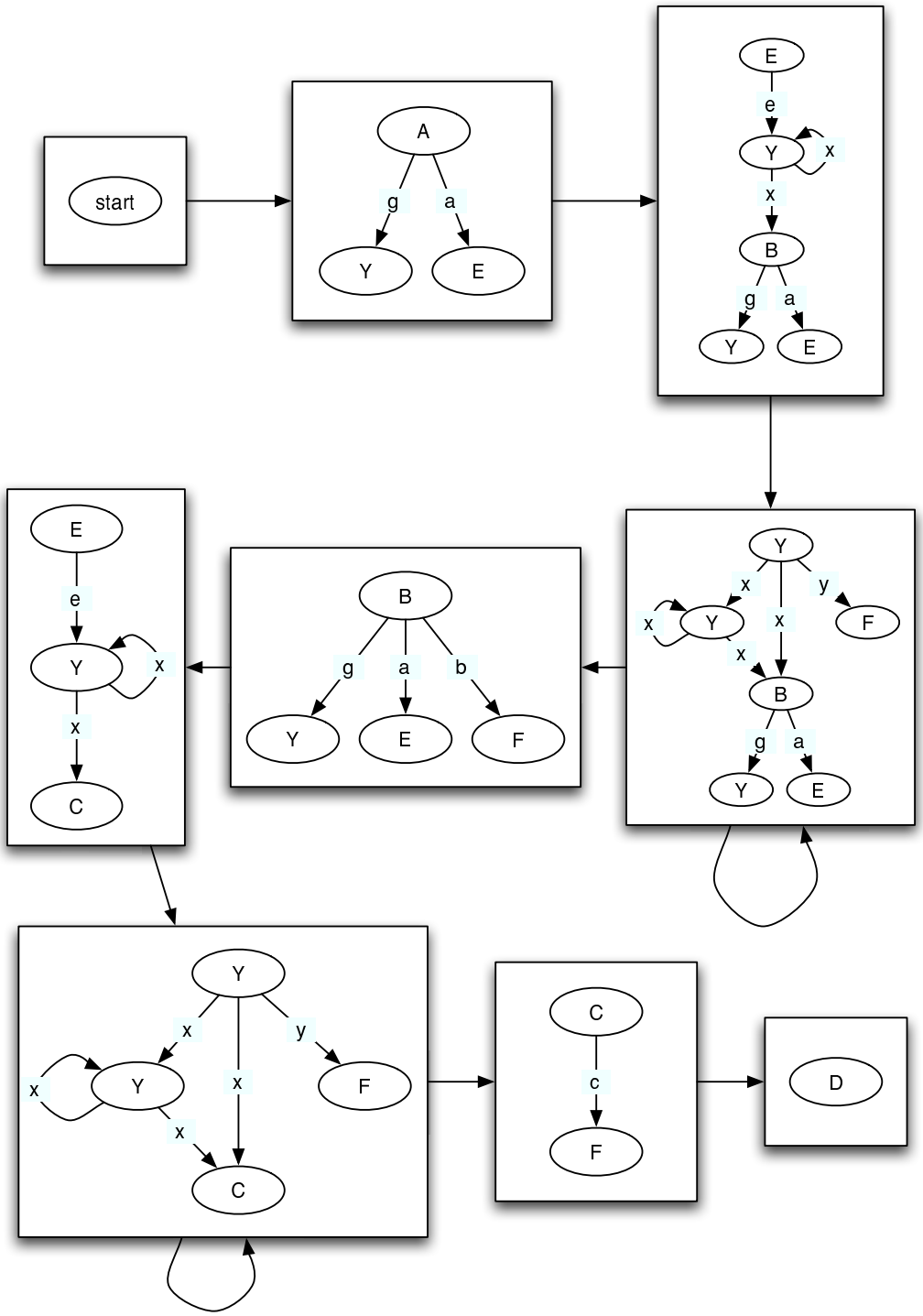
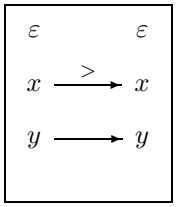


Figure 6.5: Tree Automata Call Graph for Λ_k

size-change terminating, giving size-change termination of Λ_k. To see this, consider (wlog) the cycle on the last line in Figure 6.5. Evaluation of this state (s say) entails evaluation of the body $x\ y$ of Y, that is evaluation of the state $\mathit{Bind}\ v\ (\!|F : \emptyset|\!)$, where $v \in \mathit{Lookup}^{\alpha}\ s\ x$. There are two choices of such values v; the cycle in the call graph occurs if the Y value for v is picked. However, then v is a substate of s — as we have noted before, when a cycle in the call graph occurs because of the application of a function parameter, the resulting cycle is size-change terminating. The size-change graph for this cycle is:



The decrease in x guarantees termination of this program, completing the proof that Λ_k terminates in the tree automata analysis for each k. Thus $\Lambda_k \in \mathrm{TA} \cap ((k+1)\mathrm{CFA} \setminus k\mathrm{CFA})$ for each k.

Comparing Tree Automata and $\exists k$CFA

The results of the previous section show that TA and kCFA are incomparable for each $k > 0$: while the tree automata analysis does not accept all programs that kCFA does, it does accept programs in $(k+1)\mathrm{CFA} \setminus k\mathrm{CFA}$. The purpose of this section is to show further that TA includes programs that are not accepted by *any* k-bounded CFA analysis, that is to say to prove that $\mathrm{TA} \not\subseteq \exists k\mathrm{CFA}$. The intuition behind this result is that the tree automata abstract domain can record information about environment paths of unbounded depth, which no k-bounded CFA analysis can achieve.

The program P that we shall consider in this section is given in Figure 6.6. The reader familiar with attribute grammars [Knu68, Knu71] may find it useful to note that this program was obtained from the grammar shown in Figure 6.7 (after translation to a lazy functional program [Joh87], and transforming this lazy program to a strict one). This grammar excludes terminal symbols, for simplicity. We shall not require any major results from the theory of attribute grammars, and merely include this as an aid to intuition. This program is terminating, a consequence of the fact that the grammar of Figure 6.7 is acyclic in the sense of Knuth [Knu68, Knu71].

The input to this program is a tree of type s, representing a parse tree. The structure of these parse trees is shown in Figure 6.8, where the cycles around nodes labelled C_1 indicate any number of occurrences.

The Dynamic Call Graph of P As the full description of the call graph of P is lengthy and somewhat tedious, we shall keep this presentation informal. The evaluation of P proceeds

```
type s = S of a × b
and  a = A of c
and  b = B of c
and  c = C1 of c | C2 of d
and  d = D

let pS fA fB () =
  let rec aIn ()  = bOut ()
  and     aOut () = fA aIn ()
  and     bOut () = fB ()
    in aOut ();;

let pA fC theta () =
  let rec cIn ()  = theta ()
  and     cOut () = fC cIn ()
    in cOut ();;

let pB fC () =
  let rec cIn ()  = 0
  and     cOut () = fC cIn ()
    in cOut ();;

let pC1 fC phi () =
  let rec cIn ()  = phi () + 1
  and     cOut () = fC cIn ()
    in cOut ();;

let pC2 fD phi () =
  let rec dIn ()  = phi () + 1
  and     dOut () = fD dIn ()
    in dOut ();;

let pD psi () =
  psi ();;

let rec nS (S (a, b)) = pS (nA a) (nB b)
    and nA (A c) = pA (nC c)
    and nB (B c) = pB (nC c)
    and nC = function C1 c → pC1 (nC c) | C2 d → pC2 (nD d)
    and nD D = pD;;

nS <S> ()
```

Figure 6.6: Example Program

Nonterminals	S, A, B, C, D				
Synthesised Attributes	$S \uparrow \mathbf{res}$, $A \uparrow \alpha$, $B \uparrow \beta$, $C \uparrow \gamma$, $D \uparrow \delta$				
Inherited Attributes	$A \downarrow \theta$, $C \downarrow \varphi$, $D \downarrow \psi$				
Grammar	S	$\rightarrow$	AB	$A \downarrow \theta = B \uparrow \beta$	$S \uparrow \mathbf{res} = A \uparrow \alpha$
	A	$\rightarrow$	C	$C \downarrow \varphi = A \downarrow \theta$	$A \uparrow \alpha = C \uparrow \gamma$
	B	$\rightarrow$	C	$C \downarrow \varphi = 0$	$B \uparrow \beta = C \uparrow \gamma$
	C	$\rightarrow$	C'	$C' \downarrow \varphi = C \downarrow \varphi + 1$	$C \uparrow \gamma = C' \uparrow \gamma$
		$\mid$	D	$D \downarrow \psi = C \downarrow \varphi + 1$	$C \uparrow \gamma = D \uparrow \delta$
	D	$\rightarrow$	ε	$D \uparrow \delta = D \downarrow \psi$	

Figure 6.7: Example Program: The Attribute Grammar

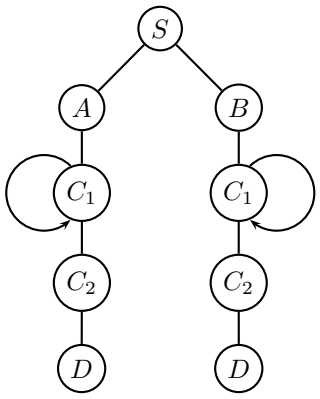


Figure 6.8: Example Program: Structure of the Input

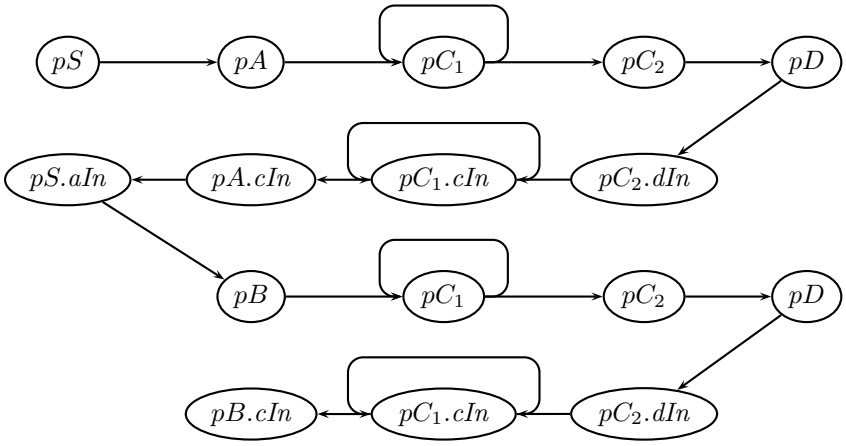


Figure 6.9: Dynamic Call Graph

as follows: first, the input parse tree is traversed by the nS function, creating a closure with same structure as the input (each nonterminal node in the tree is replaced with the corresponding production function). The call $nS\ \langle S\rangle\ ()$ in the body of the program then triggers the evaluation of the pS function to evaluate the value of the synthesised attribute of the start nonterminal S. The evaluation of the pS function is where the interest lies.

Evaluation of the productions follows the dependencies in the attributes. The main call sequence in the program is then shown in Figure 6.9 (excluding environments for clarity). Cycles indicate any number of repetitions (depending on the input).

Nodes in the call graph correspond to attributes of subtrees of the input. As such, they may be identified by the *path* from the root of the input to the corresponding subtree. As the parent of each node is passed as a parameter (to evaluate inherited attributes), there is a corresponding path in the environments of nodes. In particular, the two nodes labelled D can be differentiated by the values of inherited attributes in their environment.

Consider first the node occurring in the A subtree (d_1 say). Then the ψ parameter of pD is bound to an instance of $pC_2.dIn$ (its parent node). In turn the φ parameter of pC_2 is bound to $pC_1.cIn$ — and likewise each inherited attribute is bound to the corresponding function in its parent node. As a result, the chain of inherited attributes is as follows:

$$d_1 \to pC_2.dIn \to \underbrace{pC_1.cIn \to \cdots \to pC_1.cIn}_{m \text{ times}} \to pA.cIn$$

where m is the number of occurrences of C_1 in the A subtree. In contrast, evaluating inherited attributes in the D node occurring in the B subtree (d_2 say) eventually leads to the pB function:

$$d_2 \to pC_2.dIn \to \underbrace{pC_1.cIn \to \cdots \to pC_1.cIn}_{n \text{ times}} \to pB.cIn$$

P is not in kCFA While we shall not offer a formal proof that P is not recognised by any kCFA analysis, the intuition behind the result is thus: any k-bounded CFA analysis must identify nodes d_1 and d_2 in the dynamic call graph (where d_1 and d_2 are the two nodes labelled D, as above). For, these nodes are only differentiated by the value of their inherited attributes. As shown above, the difference is that $pA.cIn$ is reachable in the environment of d_1, while $pB.cIn$ is reachable in the environment of d_2.

However, the environments of d_1 and d_2 are *identical* at depths up to $\min\{m, n\}$ (where as above m and n are the number of occurrences of C_1 in both branches of the tree). As m and n depend on the input, in the dynamic call graph for *any* input there are occurrences of d_1 and d_2 which cannot be differentiated by k-bounded CFA, say $\alpha_k(d_1) = \alpha_k(d_2) = d$. Furthermore, as $d_1 \to^+ d_2$, by soundness $d \to_k^+ d$, and we have identified a self-loop.

Finally, the k-limited size-change graph for the call $d \to^+ d$ is not decreasing. This can be seen from the following (proofs omitted):

Fact 6.29. *The $k+1$-limited graph basis of d (for large enough m, n) is:*

$$\{\varepsilon\} \cup \{\psi\varphi^i fC^j \mid 0 \leq i + j \leq k \wedge 0 \leq j \leq i\} \cup \{\psi\varphi^i fC^i fD \mid 0 \leq i \leq (k-1)/2\}$$

Fact 6.30. *The arrows in the maximal safe $k+1$-limited size-change graph for (d_1, d_2) are:*

1. $\psi\varphi^i \xrightarrow{>} \psi\varphi^{i'} fC^{j'}$ *for all* $j' > 0$, *and* $\psi\varphi^i \xrightarrow{>} \psi\varphi^{i'} fC^{i'} fD$.
2. $\psi\varphi^i fC^j \xrightarrow{>} \psi\varphi^{i'} fC^{j'}$ *iff* $i - j > i' - j'$, *and* $\psi\varphi^i fC^j \xrightarrow{\geq} \psi\varphi^{i'} fC^{j'}$ *iff* $i - j = i' - j'$
3. $\psi\varphi^i fC^j \xrightarrow{>} \psi\varphi^{i'} fC^{i'} fD$
4. $\psi\varphi^i fC^i fD \xrightarrow{\geq} \psi\varphi^{i'} fC^{i'} fD$

for all choices of i, j, i' and j' such that the total length of paths is at most $k+1$.

We can deduce the required result:

Corollary 6.31. *The maximal safe $k+1$-limited size-change graph for (d_1, d_2) is not decreasing and so P does not lie in kCFA.*

Proof. To prove that the graph is not decreasing, we note that environment paths can be ordered so that if $p \xrightarrow{>} q$ is an arrow in the graph, then $p \prec q$, and if $p \xrightarrow{\geq}$ then $p \preccurlyeq q$. Thus no power of this graph can possess an arrow of the form $p \xrightarrow{>} p$. □

P is accepted by TA However, the program P *is* accepted by the tree automata SCT analysis. The call graph of P under the automata analysis is described by Figure 6.9 (the schematic representation of dynamic call graphs). In particular, the two D nodes are differentiated, and so the program terminates. The only cycles in the call graph are the two cycles of the form $pC_1 \to pC_1$, but these lead to a decrease in fC corresponding to the fact that the size of the corresponding subtree decreases in place.

We have thus exhibited a program that lies in TA $\setminus$ kCFA, for any k.

Nonterminals	X (start symbol) and Y
Synthesised Attributes	$X \uparrow \alpha$, $Y \uparrow \beta$
Inherited Attributes	$Y \downarrow \theta$
Grammar	$p : X \to YX' \quad Y \downarrow \theta = X' \uparrow \alpha \quad X \uparrow \alpha = Y \uparrow \beta$ $t : X \to \varepsilon \quad X \uparrow \alpha = 0$ $q : Y \to \varepsilon \quad Y \uparrow \beta = Y \downarrow \theta$

Figure 6.10: Non-SCT Program: The Grammar

```
type x = P of y × x | T
and  y = Q;;

let p fY fX () =
  let rec yOut () = fY yIn ()
  and     xOut () = fX ()
  and     yIn  () = xOut ()
  in yOut ();;

let t () = 0;;

let q inh () = inh ();;

let rec tX = function
    P (y,x) → p (tY y) (tX x)
  | T → t

and tY Q = q;;

tX <x> ()
```

Figure 6.11: Non-SCT Program

6.3.3 Size-Change Terminating Programs

Finally, let us consider the class SCT of programs that are size-change terminating with *some* choice of abstraction (more precisely, programs whose perfect static call graph under some abstraction function is size-change terminating). This class is not decidable (in particular, it is a superset of $\exists k$CFA and therefore cannot be decidable), so it is not *a priori* given that SCT does not equal the class $\mathcal{T}$ of terminating programs. In fact, our aim in this section is to show that this is not the case: there are programs that are *not* size-change terminating for any choice of abstraction.

Once more we present a program based on an attribute grammar. The grammar is shown in Figure 6.10. This grammar is (absolutely) noncircular, and so the program P obtained from it (given in Figure 6.11) is terminating. We shall show that P is not size-change terminating.

We shall show that this program is not size-change terminating for any choice of abstrac-

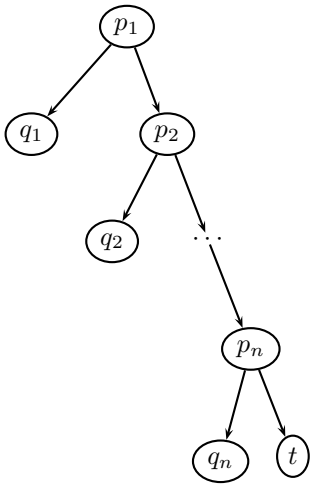


Figure 6.12: Non-SCT Program: Input Tree

tion function, using the results of Section 6.2.4. We must first choose the abstract graph basis function gb^α, mapping each abstract state to a set of environment paths. For simplicity we choose the 1-limited graph basis function (as in 0CFA and the tree automata analysis), but the result holds for k-limited graph bases also.

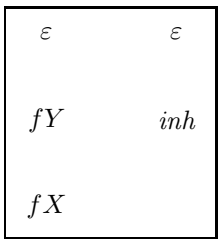
The input to the program is a tree of type x. This is represented in Figure 6.12, with distinct nodes identified with integer labels.

This leads to the following call sequence (where we use the integer labels appearing in the figure as superscripts to indicate which node in the input tree each instance of a function represents):

$$p^1 \to q^1 \to p^1.qIn \to p^2 \to q^2 \to p^2.qIn \to p^3 \to \cdots$$

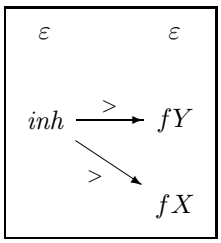
Some intermediate states are not represented in the above call sequence. The length of this call sequence increases with n, and thus the length of call sequences of this form in the dynamic call graph varies unboundedly with the input. Hence (Section 6.2.4) at least two states in this call sequence must be identified by any abstraction, and it suffices to show that the composition of size-change graphs between any two compatible states (states with the same program point and graph basis) is nondecreasing.

The (1-limited) size-change graphs for the three different kinds of calls in the above call sequence are shown below:



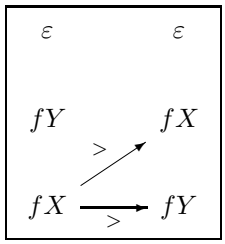
$p \to q$

γ



$q \to p.yIn$

δ



$p.yIn \to p$

ε

We can now show that the unbounded call sequence shown above is *locally nondecreasing* — the composition of graphs along any call sequence $p \to^+ p$, $q \to^+ q$ or $p.qIn \to^+ p.qIn$ is nondecreasing.

This is trivial however — any such composition must contain the graph γ, and so as γ is the empty graph so is the composition.

We have therefore shown that P is *not* size-change terminating for any choice of abstraction with 1-limited graph bases. The result extends to arbitrary k-limited graph bases.

The relationship between the different classes: kCFA, $\exists k$CFA, TA, $\lambda^{\to}$ (the set of simply-typed λ-expressions), SCT and $\mathcal{T}$ (the set of terminating programs) is illustrated in Figure 6.1 (p. 151).

6.4 References

There appears to be relatively little work relating the expressiveness of distinct static analyses, or comparing what may be achieved by static analysis with other methods of determining safety, such as typing. A notable exception is Palsberg and Schwartzbach's work on safety analysis of the λ-calculus [PS95]. In this context, *safety analysis* denotes the use of an equivalent of 0CFA to approximate the set of values that operators are applied to (in a λ-calculus with constants) to verify that unsafe operations such as $0 - \mathbf{true}$ are never performed at runtime. 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-bounded CFA analysis achieves this for termination in the SCT framework.

Heintze [Hei95] shows that several variants of 0CFA may be seen as equivalent to *flow-annotated* type systems. The flow-annotated type systems are variants of well-known type systems (such as simple types) in which each function type is given a set of labels denoting the possible program locations corresponding to these functions. It is shown that 0CFA is equivalent to a type system with subtyping and recursive types. The simply-typed λ-calculus is equivalent to a restriction of 0CFA that prohibits reasoning about recursion.

The author is grateful to Luke Ong and William Blum for many enjoyable discussions on the relationship between size-change termination and simple types. Oege de Moor suggested the study of attribute grammars as functional programs with highly nontrivial termination properties.

Chapter 7

Discussion and Related Work

7.1 Summary

7.1.1 Size-Change Termination of Higher-Order Programs

We have shown that the size-change termination analysis can fruitfully be extended to higher-order functional programming languages, both strict (call-by-value) and lazy. We have extended the ideas of first-order size-change termination [LJBA01] and of the λ-calculus termination analysis of Jones and Bohr [JB04] to a language that is adequate to faithfully model real purely functional programming languages. Programs written in (the purely functional core of) ML and related languages, or lazy languages such as Haskell, can be translated into a form that can be handled by our analysis by straightforward program transformations. The higher-order size-change termination analysis has been implemented for a subset of Objective Caml, and it is straightforward to extend this to further languages.

Our formulation of higher-order size-change termination draws on much existing work on the SCT criterion, from the basic idea behind this method for proving termination to the syntactic order on closures, defined in Section 3.1. However, our formulation is substantially more general than has previously been achieved. In our setting, it is straightforward to derive both the call relation (defining the dynamic call graph) and dataflow annotations together with calls. Furthermore, where previous incarnations of SCT analyses track size changes in immediate function parameters, we have defined more general notions of graph bases and environment paths. This both simplifies the description of the analysis (in particular, size changes in the closure order can be derived from dataflow) and allows for strictly increased precision.

The extension to lazy languages is highly nontrivial. The size order on closures is not sufficient to prove termination of nontrivial lazy programs, and it is necessary to extend the analysis to consider for each parameter both its syntactic representation and that of its value. However, given this change in the order used to prove termination, the extension to

lazy languages demonstrates that the higher-order SCT analysis may be straightforwardly instantiated for a language with very different semantics. Any static analysis developed for termination analysis of strict languages can therefore be used for lazy languages with little modification.

7.1.2 Static Analysis and Precision

The higher-order SCT analysis requires the construction of a static call graph for the program, annotated with dataflow and size-change information. The size-change annotations are essentially fixed by the order relation chosen (and can be derived from the dataflow annotations in the exact semantics), up to the depth limit on environment paths. However, the choice of abstraction of the control flow is crucial in the concrete realisation of the termination analysis.

We have described three static call graph constructions, each of which is essentially defined by the abstract representation of states (in particular the representation of environments). The first analysis, 0CFA, is the best-known existing flow analysis of functional programs. The advantages of 0CFA are numerous: this is a straightforward analysis (both to describe and implement), and the computation of the call graph is inexpensive. However, the call graph constructed by 0CFA is often too imprecise for good termination proofs — even in simple cases such as uses of the *fold* functions, 0CFA may introduce many spurious dependencies in the call graph. This is not an accident of our definition of 0CFA, but instead is inherent to the abstract domain in which each node of the call graph corresponds to a program point.

The k-bounded CFA analysis aims to improve precision of static analysis by extending the abstract domain: we record additional information about the environment in each node in the call graph. Our notion of k-bounded CFA differs from the classical formulation of k-CFA, as rather than approximate contexts (the call stack), we choose to approximate environments. This is arguably more natural in our setting, and enables the use of more general $(k+1)$-limited size-change graphs to improve precision further. Furthermore, common cases such as *map* are easier to handle in the environment formulation. We have shown that the increased precision of k-bounded CFA (for small values of k) allows more effective termination proofs, at the cost of course of an increase in complexity. We have further shown that the set of programs recognised by the k-bounded CFA analysis strictly increases with k.

The use of k-bounded CFA only appears appropriate to disambiguate uses of such "simple" functions as *map* and *fold*, however. It is both unnatural and expensive to record information about the environment up to a bounded depth in programs that create closures of unbounded depth. While many examples of such programs appear artificial (for instance, programs using Church numerals), lazy programs provide numerous examples of this phenomenon in practice. We have introduced a new analysis, based on the representation of environments as a restricted class of tree automata. This representation is highly natural for programs with unbounded environments, and fixes no arbitrary bound as in k-bounded CFA.

This again comes at the cost of complexity, as the state space of the analysis is exponential. Furthermore, the tree automata analysis accepts programs that *no* k-bounded CFA analysis can, but fails to accept programs that are found terminating by 1-bounded CFA.

The study of the *expressive power* of static analysis, in terms of the set of programs accepted by the analysis, appears to have been often neglected. It is certainly the case that experimental evidence is an important tool for evaluating the practical applicability of a program analysis. For, the test of usefulness in practice can only be success on the type of programs that human programmers write. However, a more principled view is both of theoretical interest and of use in understanding the tradeoff between complexity and precision.

We have described the relationships between classes of programs accepted by our various analyses precisely, in terms of the containments that hold between these sets of programs. Furthermore, we have highlighted the different causes for imprecision in the analysis (in particular, the difference between a failure due to the choice of abstract domain and one that stems from a loss of precision in the analysis). Finally, we have clarified the relationship between our SCT analyses and what is probably the best-known set of terminating programs, namely the simply-typed λ-calculus. This is a step toward a deeper understanding of what may and may not be achieved in a static termination analysis such as ours. As remarked by Abel [Abe06, Abe04] in the context of termination checking using sized types, this suggests that static analysis and typing may be complementary for verifying termination.

7.2 Related Work in Termination Analysis

The field of termination analysis is of course a vast one, dating back to the origins of computer science (at least as concerns its theoretical study and in particular undecidability). As such, the approaches to termination analysis are numerous, ranging from purely static-analysis tools (such as our own termination criterion) to the use of a complete theorem prover such as HOL to prove termination.

In this section we shall attempt to give an overview of this field, by describing some of the existing termination checkers. We shall start by describing work on size-change termination in some detail, as this is most closely related to our own work. Next, we present work on termination analysis of logic programs, as this is a prime domain for termination analysis (termination of Prolog programs is difficult to establish, to say the least). We will then describe the use of termination analysis within theorem proving environments, and finally we shall end this section by briefly describing work in termination of term rewriting systems.

7.2.1 Size-Change Termination

The size-change termination criterion was introduced by Lee, Jones and Ben-Amram in 2001 [LJBA01], for first-order purely functional (strict) programs. In this work the algorithm

used to extract size-change graphs from the program is *not* given, so that the focus is on the algorithm used to decide the SCT criterion for an annotated static call graph (as in Chapter 3). It is proved that this is sound, and that the SCT problem (for first-order programs, or equivalently for annotated call graphs) is PSPACE-complete by reduction of the termination problem for boolean programs to SCT. The closure algorithm for deciding size-change termination, in the form that we have presented, appears to be due to Wahlstedt [Wah00].

Implementation Frederiksen [Fre01] describes an implementation of the SCT criterion for first-order languages in more detail. The focus is on a simple abstract interpretation-based analysis of first-order programs to produce an annotated call graph. The use of global size analysis (proving that, for instance, the return value of *filter* is a sublist of its input) to deal with programs such as QuickSort is suggested. An alternative to global size analysis is to use sized type annotations [Par00, CK00]. Finally, the use of more general size-change graph labels (in particular including increasing arrows) is introduced to deal with non-monotonic descent. However, these methods can only handle values that increase *boundedly*.

Partial Evaluation The introduction of size-change termination was motivated by the need for an effective automatic termination criterion for use in program specialisation. Jones and Glenstrup [JG02, GJ05] consider termination of *offline partial evaluation*. In offline partial evaluation, a first phase of the algorithm marks each variable in the program as *static* (known at specialisation time) or *dynamic*. Likewise, each expression is marked as *reduce* (evaluate fully during specialisation) or *specialise*. The partial evaluator then obeys these annotations. An analysis is described that, based on the results of the SCT analysis, decides whether each variable is bounded (takes only finitely many values) and thus may safely be marked static. This work is based on Lee's PhD thesis [Lee01].

Complexity The SCT criterion (for annotated call graphs) is PSPACE-complete. This complexity is somewhat worrying, though we have argued that the maximal bound is rarely reached for real programs. For, it is necessary to use recursive function calls with *permuted* parameters to attain this bound. Lee [Lee02a] describes two approximations to the (PSPACE-complete) SCT criterion, respectively quadratic and cubic in time complexity. It is further shown in the (upcoming) journal version of this paper that these methods are complete, and thus accurate, for large classes of sets of size-change graphs: *stratifiable* size-change graphs in which variables may be ordered (by $\preccurlyeq$ say) such that data can only flow from x to y if $x \preccurlyeq y$; and *fan-out free* (each input variable flows to at most one output) and *strict fan-in* (if more than one variable flows to an output, all corresponding arcs are decreasing) graphs. It is shown empirically that these conditions are not limiting in practice. The same algorithms apply to *finiteness* analysis (as used in the termination of partial evaluation [Lee02b]). As our focus has been on the methods used to extract an annotated call graph we have not

described this in further detail, but this approximation is applicable to our analyses also.

Integers The SCT analysis, as originally stated [LJBA01], suffers from the drawback that only well-founded datatypes can be used. This makes its application to real languages *a priori* impossible, lest we forgo the use of integers. This problem has been studied, and it is possible to remedy it in a natural way. We assume that arrows in size-change graphs are labelled with relations drawn from $\{>, \geq, =, \leq, <\}$ (that is, increases are tracked as well as decreases). In Avery [Ave06], the *convex hull* analysis of Cousot and Halbwachs [CH78] is used to approximate the values of the program state (the tuple of variables appearing in the program) as a convex (n-dimensional) polyhedron. That is, a set of inequalities of the form $0 \leq c + \sum_i a_i x_i$ (where the x_i are the program variables) is deduced. It can then be determined for each size-change graph and each such linear expression if the size-change graph decreases the value of the linear expression. As the values of such expressions are natural numbers, the SCT criterion can be adapted. Once more, this work may be applied in our setting to deal with integer values.

Expressive Power A natural criterion, generalising simple primitive recursion, is *lexicographic descent*: this holds if in any recursive call (wlog from a function $f(x_1, \ldots, x_n)$ to itself), the tuple $(x_1, \ldots, x_n)$ decreases in the lexicographic (dictionary) order. More generally, we may only require that *some* rearrangement of the parameters of f satisfy this. The size-change termination principle generalises lexicographic descent: any program with lexicographic descent is size-change terminating. Jones [LJBA01] conjectured that the class of *functions* expressible as size-change terminating programs is no greater than that accepted by lexicographically decreasing programs (the *multiple recursive* functions [Pét69]), though the class of size-change terminating programs is larger (for instance, functions that permute their arguments can be size-change terminating, but not lexicographically decreasing). This was proved by Ben-Amram [BA02]: any size-change terminating program may be converted to an equivalent program that exhibits lexicographic descent.

Non-monotonic Descent A limitation of the SCT analysis, even if extended to handle increasing parameters, is that only *monotonic* variation in recursive calls can be handled. That is, termination can only be proved if a parameter decreases along every call. It is easy to construct an example where this does not hold, but the program nonetheless terminates: define functions $f(x)$ and $g(x)$ with calls $f(x) \to g(x+1) \to f(x-1)$. Termination is guaranteed because the net effect of any recursive call is to decrease x, but this is not detected as x both increases and decreases along the sequence of calls. Frederiksen [Fre01] suggests recording *bounded* increases (up to some user-specified upper bound), which is a somewhat *ad hoc* solution to this problem. More generally, we may consider size-change graphs with arrows of the form $x \xrightarrow{+k} y$ and $x \xrightarrow{-l} y$ (with respective meanings: $y \leq y + k$ and $y \leq y - l$) for natural numbers k and l, strictly generalising our notion of size-change

graph, and allowing non-monotonic decrease to be expressed. Ben-Amram [BA06] shows that the natural termination criterion for such graphs (which he calls δSCT) is undecidable, by reducing the halting problem for counter programs to δSCT. However, δSCT is decidable for fan-in-free graphs (graphs in which there is at most one arrow to any output). Anderson and Khoo [AK03] consider a related, but more general, extension of the SCT principle. Size-change graphs are replaced by arbitrary *affine relationships* between input and output variables (allowing more complex relationships, such as $x' \leq x+y$ to be considered). However, the termination condition is not decidable, and it is necessary to use a widening operator to control termination of the algorithm.

Runtime Analysis Given an annotated static call graph, it is possible in some cases to deduce properties of the program beyond termination. Consider as a simple example a primitive recursive function such as $fac(n)$, the call graph of which contains a single self-call annotated with the graph $n \stackrel{>}{\longrightarrow} n$. Naturally we may deduce that *fac* terminates, but beyond this we may show that the length of any call sequence in the evaluation of $fac(n)$ is bounded by n (as n decreases at each call), whence it follows that *fac* is a linear time function (assuming the body of *fac* evaluates in constant time).

Frederiksen [Fre02] shows how the SCT analysis can be used to prove that functions evaluate in polynomial time (without giving more precise bounds). The annotated call graph may be used to show that the *call depth* of the program is polynomial. In order to deduce polynomial time complexity for the function being analysed, it is further necessary to show that no *overlapping* recursive calls are made from the function. For instance, the naive (exponential-time) Fibonacci function has linear call depth, but overlapping recursive calls. It may then be deduced that the function has polynomial time complexity. Anderson *et al* [AKAL05] explore the inference of more precise polynomial (as well as a restricted class of exponential) bounds for functional programs using the SCT method. This procedure requires a guess on the form of the runtime bound (in the polynomial case, the maximal possible degree), and produces a set of polynomial equalities with unknown coefficients. A quantifier-elimination procedure due to Tarski allows the coefficients to be found, whence a precise runtime bound can be produced

λ-Calculus Finally, the application of the size-change termination principle to higher-order languages builds on the work of Jones and Bohr [JB04] on termination of the pure untyped λ-calculus. The approach of Jones and Bohr is mirrored in our own: first, an environment-based semantics of the pure call-by-value λ-calculus is defined, with the property that terms occurring in the evaluation are subexpressions of the term being evaluated. A sequentialised form is then introduced. The subtree order on function values is defined, and finally a sound approximation to the exact semantics (annotated with size-change graphs) is given, from which termination can be proved. While our presentation uses many of the same techniques, these are generalised considerably for our setting (in addition to the changes introduced to

extend the input language). In particular, we introduce the sequentialising transformation as a general transform on operational semantics, and define arbitrary-depth size-change graphs to allow for more refined abstractions of the call graph. Finally, we show that different choices of abstraction allow control over the cost / precision tradeoff.

7.2.2 Termination of Logic Programs

There is a substantial body of work focusing on termination analysis of logic programs, and in particular of Prolog programs. This is largely due to the fact that the expressiveness of logic programming, together with Prolog's incomplete (depth-first) evaluation strategy, make it very easy to write nonterminating programs, and indeed Prolog programs typically rely on features such as the *cut* operator to control termination. As an example, consider the following Prolog definition of the list append function:

$$\begin{aligned} &\mathit{append}([], L, L) \\ \mathit{append}([H|L_1], L_2, [H|L_3]) \quad &\leftarrow \quad \mathit{append}(L_1, L_2, L_3) \end{aligned}$$

Evaluation of the query $\mathit{append}(A, B, C)$ depends on the nature of A, B and C: if A and B are ground terms (actual lists), it is easy to see that evaluation terminates, and that the only result for C is the concatenation of A and B. This program likewise terminates if C is ground. However, if both A and C are free variables, then evaluation of the query is nonterminating — we are, after all, solving for all A and C such that $A +\!\!+ B = C$, and this has infinitely many solutions.

This examples illustrates one of the crucial factors in termination of logic programs, namely the *modes* of arguments (ground or free), which any (useful) termination analysis for Prolog must take into account. In the remainder of this section we shall give a brief overview of some work on termination of Prolog programs, before mentioning the use of termination analysis for a different class of logic programs (Datalog).

Termination of Prolog Programs

Apart from the specific issue of argument modes, the main idea in Prolog termination proofs is similar to size-change termination — termination checkers typically attempt to find a parameter that is guaranteed to decrease in size in a recursive call. For instance, in the *append* function defined above, the size of the list in the first argument decreases at each recursive call, *provided* this argument is ground. If the first argument is a variable, then no decrease is observed. In this context, *termination* refers to termination of the Prolog (depth-first, left-to-right) evaluation procedure, for *all* results.

Termination of Function-free Programs The simplest class of Prolog are *function-free* programs. This denotes the class of programs expressed in a logic without function

symbols (*i.e.* constructors in Prolog). Function-free programs can thus only manipulate primitive datatypes such as characters or integers. Brodsky and Sagiv [BS89, BS90] show that appropriate *motonicity constraints* may be used to prove termination. In this setting, primitive predicates are annotated with monotonicity constraints (size-change information), an example of which is $p(X, Y) : X < Y$, with intended meaning: whenever $p(X, Y)$ for any ground atoms X and Y, it is the case that $X < Y$ (in some appropriate well-founded order).

This information is used to produce *argument mappings* for rules. These are analogous to size-change graphs, with nodes representing all argument positions of predicates in (a prefix of) a rule, and arrows either representing equality between nodes, or size-change (deduced from primitive predicate monotonicity constraints). In addition, nodes are coloured *white* or *black* to represent binding modes — the modes for the source nodes (arguments of the head of the rule) are supplied, and the modes of other arguments inferred.

Termination is then detected in a similar way to the (first-order) SCT criterion — namely, the cyclic composition of argument mappings (analogous to the closure of a set of size-change graphs) is computed, and the program is terminating if every cyclic composition has a decreasing cycle. Argument modes are dealt with by specifying a *query pattern* (colouring of nodes in the head of the rule), and restricting compositions to those graphs that are compatible with the query pattern. Sagiv [Sag91] introduces a related test that does not rely on the assumption that the underlying data set is finite (in earlier work [BS90] this case is only handled for a restricted class of well-founded orders), and thus is more readily applicable to programs with function symbols.

Termination of General Prolog Programs Termination analysis for general Prolog programs (with constructors and thus data structures such as lists) fundamentally relies on the same arguments as in the function-free case — finding decreasing parameters in recursive function calls — but the origin of size relationships is different. In general, a notion of the size of a *term* is required. Many such norms have been proposed, for instance the list size norm [UG88] and the total term size norm [SG91]. The relationship between the norm of different terms appearing in a rule can be deduced, and this may be used to prove termination.

The main issue in producing size-change relationships for calls in Prolog programs is that terms may contain free variables, and these must be considered to obtain accurate size-change information. For instance, in the NAIL! system [UG88], *variable/argument graphs* are produced for each rule, giving size relationships between each of the arguments of predicates in the rule, as well as each free variable occurring in the rule. The size constraints implied by such graphs can be used to find decreasing values along recursive calls. Plümer [Plü90b, Plü90a] considers a similar termination criterion, but considers more general size constraints (general linear constraints between sets of variables, rather than simple constraints of the form $X \leq Y + k$).

The Termilog system [LS96, LS97, LSS97] generalises earlier work in termination analysis of Prolog programs. Termilog allows the use of general linear norms on terms, of the form

$\|f(T_1, \ldots, T_n)\| = c + \sum_i a_i \|T_i\|$. This norm may be used for terms that contain free variables, but the result may then be a symbolic expression, rather than a constant. A term is said to be *instantiated enough* if its norm is constant.

Termilog proceeds as the analysis of function-free programs [Sag91]: for each rule, a *rule graph* is constructed to describe size relationships between arguments of predicates in the rule. However, due to the presence of free variables, *weights* are introduced to express more general inequalities: an edge $E_1 \xrightarrow{2+H} E_2$ encodes the equality $\|E_1\| = \|E_2\| + H + 2$ (where H is a logical variable). As in the function-free case, a *query pattern* is provided, but its interpretation is different. In the query pattern, a white node denotes an argument that is not instantiated enough, while a black node indicates that the argument is instantiated enough. As before, this is used to infer further black nodes (*i.e.* decide which arguments are fully ground with respect to the size measure used). The same termination algorithm (similar to SCT) is then used to prove termination.

Datalog Programs and Negation

Datalog is a subset of Prolog, corresponding to the function-free fragment of the logic (*i.e.* constructed datatypes such as lists are not allowed), introduced in the context of deductive databases. The semantics of Datalog programs are more straightforward than that of general Prolog programs — in particular, the denotational semantics of Datalog corresponds more closely to its operational behaviour. For instance, the Datalog definition $p(X) \leftarrow p(X)$ is a *bona fide* definition of the empty relation p, though this is a nonterminating Prolog program. In general, the semantics of a Datalog program is given by the straightforward least fixpoint computation. We have previously introduced termination criteria for the function-free subset of Prolog. It should be emphasised, however, that this is not what is meant by Datalog here, the difference being reflected in the definition of $p(X)$ above. Given a few restrictions (ensuring that relations are finite) it is straightforward that *all* Datalog programs terminate (in the least fixpoint semantics, if not as Prolog programs).

However, the use of *negation* in Datalog programs is problematic, and can give rise to programs which do not define relations. The simplest example is the definition $p(X) \leftarrow \neg p(X)$. Of course, no relation p satisfies $p = \mathscr{V} \setminus p$ (where $\mathscr{V}$ is the universe), and so the relational semantics of this program is undefined[1]. This causes nontermination of the fixpoint computation, due to the fact that the $\neg$ operator is not monotonic.

A subset of Datalog known as *safe* (or *stratified*) Datalog has been introduced to avoid such undefined programs. In stratified Datalog, negation is not allowed in recursive predicate definitions, and thus any stratified Datalog program has a well-defined relational semantics. However, this syntactic restriction is overly coarse, and excludes many potentially useful programs. As a result, wider classes of terminating Datalog programs have been introduced,

[1] The *well-founded semantics* [vGRS91] assigns a meaning to any Datalog program. In this case, the truth value of $p(a)$ for any atom a is simply undefined. A program admits a relational semantics if its well-founded model is two-valued (each ground literal is either true or false).

in particular *modularly stratified* Datalog programs [Ros90, Ros94a], which are terminating programs with desirable implementation properties (subgoal-at-a-time evaluation is possible, and efficient top-down and bottom-up evaluation strategies are known). Unfortunately, the class of modularly stratified Datalog programs is both *semantic* (in the context of deductive databases, this refers to the fact that modular stratification depends on the nature of the data in the database) and undecidable.

Syntactic Conditions for Termination of Datalog Programs Ross [Ros94b] introduces a syntactic condition (*universal constraint stratification*) for termination of Datalog programs. This relies on the use of monotonicity constraints, as introduced by Brodsky and Sagiv [BS89, BS99]. Recursion through negation is then allowed provided that the recursion is well-founded. For instance, suppose a primitive relation $w(X, Y)$ is annotated with the information $w(X, Y) : Y < X$. A rule such as $p(X) \leftarrow w(X, Y), \neg p(Y)$ is then permissible: though this is a negative-recursive rule, no paradoxes can arise, as $p(a)$ can never depend negatively on itself (for any atom a).

Universal constraint stratification is decided in two phases: first, order constraints are inferred on all predicates in the program (from the given constraints); it is then checked that any negative recursive dependency gives rise to an unsatisfiable constraint. Constraint inference for Datalog programs was studied by Brodsky and Sagiv [BS89, BS99]. Surprisingly, the problem is decidable (for negation-free programs): given a set of constraints on primitive predicates, the maximal constraint logically entailed on each defined predicate can effectively be computed. In programs with negation this is undecidable in general, but a good approximation can be obtained by ignoring negated subgoals in the constraint inference process.

The second phase of the algorithm is very similar to the SCT algorithm: each dependency is annotated with a constraint, and for example a dependency $p(X) \leftarrow p(X')$ might be given the constraint $X' < X$. This is analogous to the size-change graph $X \xrightarrow{>} X$. For each negative cyclic dependency, the composition of constraints along the cycle is computed. If this is *cyclically unsatisfiable* (in the SCT formulation this corresponds to a decreasing graph), then the recursion is safe. The constraints considered in the literature on Datalog programs are slightly more general than size-change graphs, however. In particular, constraints of the form $X = a$, where a is a constant, are allowed, leading to improved precision.

Size-Change Terminating Datalog The universal constraint stratification criterion of Ross guarantees termination, but does *not* imply modular stratification. This is unfortunate, as this precludes the use of known implementation mechanisms for modularly stratified Datalog.

This author has proposed a variation on universal constraint stratified Datalog, which we call Size-Change Terminating Datalog. An SCT Datalog program is necessarily modularly stratified (and thus terminating), allowing for a wider range of implementation strategies. The crucial difference between universally constraint stratified Datalog and SCT Datalog lies

in the inference of constraints on dependencies. For instance (this example is due to Ross [Ros94b]), consider the program:

$$\begin{aligned} p(X,Y) &\leftarrow e(X,Y) \\ p(X,Y) &\leftarrow p(X,Z), f(Z,Y), \neg p(Z,Y) \end{aligned}$$

with order constraints $e(X,Y) \Rightarrow Y < X$ and $f(X,Y) \Rightarrow Y < X$. The algorithm of Brodsky and Sagiv [BS89, BS99] infers the constraint $p(X,Y) \Rightarrow Y < X$. This program is then universally constraint stratified: in the dependency $p(X,Y) \leftarrow \neg p(X',Y')$, the constraint $X' < X \wedge Y' < Y$ is deduced, and this is not cyclically satisfiable.

In our SCT criterion for Datalog, we reject the use of constraints on predicates that lie in the same component as a predicate p (that is, predicates that are mutually recursive with p) when deducing constraints on dependencies from p. In particular, in the above example the constraint $X' < X$ is *not* deduced, and so the above program is not size-change terminating — this is desirable, as this is not a modularly stratified program. It is hoped that size-change terminating Datalog will prove a useful termination criterion for Datalog programs, while allowing efficient implementation strategies.

7.2.3 Theorem Proving

A common task in a theorem proving environment such as HOL [GM93] or Isabelle [NPW02] is to define new *function symbols* for use in proofs, to model the use of defined functions in mathematics. In higher-order logics, functions (like other terms) are expressed in (some variant of) the simply-typed λ-calculus. This implies another use of defined functions in theorem proving — if the proof task is to verify a functional program, then it may be possible to express the program directly in the theorem prover's logic, avoiding the difficulties of creating a model of the program.

However, defining functions in theorem provers is fraught with difficulties. The crucial issue is that the definition of *nonterminating* functions may cause logical inconsistencies (rather, the theorems that are deduced from such definitions may be inconsistent). Any sound theorem prover must therefore ensure that any tentative function definition denotes a *bona fide* total function. Due to this requirement, function definition is often restricted to very simple (and limiting) forms, such as primitive recursion, which are guaranteed to terminate.

The TFL system [Sli99], implemented both for HOL and Isabelle, greatly facilitates this task. TFL allows arbitrary functions (written in any suitable functional language, ML being appropriate for HOL) to be defined, and automatically infers *termination conditions* [Sli96]: assertions that the size of the tuple of arguments to a function decreases, in *some* well-founded order. The order may be supplied at function definition time, or left unspecified (in which case termination conditions may need to be discharged at a later stage). TFL then deduces

the existence of the defined function, together with *induction schemes* [Sli97] for reasoning about the newly defined function.

The aims of TFL are very different from those of a fully automatic termination analysis such as ours. In particular, TFL can use the full power of the theorem proving environments it is implemented for to prove termination. This may be used to provide complex well-founded orders (a general measure function can be used to generate a well-founded order). However, this shifts the burden of finding the appropriate well-founded order on the user. For instance, termination of Ackermann's function can be shown automatically, but only provided the lexicographic order has been specified. In many simple cases the order is detected automatically, however.

TFL is limited when dealing with higher-order functions. Slind [Sli99] gives the following example of *higher-order recursion*, to implement an "occurs check" for an element in a tree:

```
type α tree = Node of α × α tree list

let rec occurs x (Node (v, tl)) =
  if x = v then true else
  List.exists (fun t → occurs x t) tl
```

This is trivially size-change terminating. However, it is necessary to manually provide an appropriate *congruence rule* to prove termination in TFL (crucially, to show that the elements of the list *tl* are subtrees of the original tree).

7.2.4 Transition Predicate Abstraction and Terminator

Transition predicate abstraction is a novel technique for proving termination of programs, based on *predicate abstraction* [GS97]. In predicate abstraction, an (infinite-state) program is abstracted to a finite-state *boolean program*, which may be verified by model-checking (of course, the abstract program is an overapproximation of the original program). More precisely, given a set of predicates $\varphi_1, \ldots, \varphi_n$, the state of the abstract program is represented by a tuple of boolean values $(P_1, \ldots, P_n)$. This represents any state in which φ_i is true iff $P_i = \mathrm{T}$. Furthermore, these predicates need not necessarily be introduced by the programmer, as good heuristics are known for deducing useful predicates from the program text [FQ02]. Predicate abstraction has proved very effective in industrial tools for verification such as the SLAM tool, which further uses counterexample-driven abstraction refinement [BMMR01]. However, predicate abstraction is essentially limited to the verification of safety properties, as the abstracted program represents an overapproximation of computation paths in the infinite-state program. Furthermore, the abstract program is nonterminating in all but the simplest cases, so this is not adequate for termination proofs.

Transition predicate abstraction [PR05] extends the idea of predicate abstraction beyond safety properties. As in predicate abstraction, the program is simulated by a finite-state transition system, but *transition predicates* (predicates over *pairs* of states) are used to model

the effect of computation paths. For instance, the predicate $\mathrm{at}(l_0) \wedge \mathrm{at}'(l_0) \wedge y' < y$ holds of those pairs (s, s') of states at the same program location l_0, such that the value of y in s' is less than its value in s. The abstract program is represented as a finite transition system with transition predicates as states — abstracting the infinite-state state system defined by: $(s, s') \xrightarrow{A} (s, s'')$ iff state s' leads to state s' *via* statement A.

It is shown [PR05] that given a (finite) set of transition predicates, an abstract model may be computed from the program. In general this procedure requires the use of a theorem prover to test entailment of transition predicates from exact transitions. The SCT method can be seen as a special case of transition predicate abstraction, where transition predicates are limited to the set of relations of the form $x' < y$ and $x' \leq y$ (so that deducing such predicates may be achieved without theorem proving). Furthermore, both a termination criterion similar to the SCT criterion and a liveness condition can be formulated in this framework [PR04b, PR04a].

Terminator

Transition predicate abstraction is used in the *Terminator* [CPR06a, CPR06b] system for termination proofs, aimed at verifying termination of device drivers. As in the SLAM system, abstraction refinement [CPR05] allows automatic discovery of useful transition predicates. The termination criterion used in Terminator is as follows: a set of well-founded relations $\{R_1, \ldots, R_n\}$ must be found such that for any cyclic path π in the program, the abstract transition predicate for π is contained in some R_i. A spurious counterexample is one whose *exact* transition relation is well-founded (though its abstract transition relation is not seen to be).

The abstraction refinement procedure analyses the program with respect to a given abstraction, and a given set of candidate well-founded relations. If a counterexample is found, it is checked whether this is spurious, leading to two possibilities. The abstract transition predicate may be inadequate, in which case more predicates are added and the analysis restarted; or the set of well-founded relations was not sufficient. In the latter case we attempt to derive a well-founded relation for the spurious counterexample, and use this to analyse the program again. It is thus not clear that this procedure necessarily terminates, though counterexamples are removed at each iteration of the algorithm. Terminator produces good termination results for Windows drivers in practice [CPR06a].

A substantial difficulty in the analysis of imperative programs is proving termination of programs with mutable heap structures. While termination of imperative programs without pointers (or with only boundedly many reachable heap locations) is similar to termination of functional programs, analysing programs with mutable structures is more complex. For instance, an algorithm traversing a linked list may be terminating only if the list is acyclic. Likewise, establishing termination of a loop traversing a circular linked list requires a sophisticated measure function, namely the distance between the current node and the header

node.

Mutant/Terminator [BCDO06] is an extension of the Terminator tool to handle such cases. The representation of the heap in Mutant uses *separation logic*, allowing such properties as acyclicity of a list to be represented. The abstraction used by Mutant deduces the separation logic representation of the heap automatically. Termination is then proved using the Terminator procedure, over an extended representation of program state including the *depth* of heap structures, which is crucial to establish termination. The Mutant/Terminator system appears to provide good results for those Windows drivers for which Terminator is not sufficient [BCDO06].

7.2.5 Term Rewriting Systems

Much of the early work on termination has been concerned with termination of term rewriting systems, and this continues to attract much interest. We shall not describe term rewriting systems in detail, and refer the reader to introductions to the subject [BN98, Ter03] for more information, but shall present a very brief overview of termination proofs for term rewriting systems.

The basic idea behind termination proofs is, as before, the use of a well-founded order. More precisely, a *rewrite order* is a well-founded partial order $<$ on terms that is compatible with substitution and term construction. Then instances of the following result are used to prove termination: a TRS is terminating iff there is a rewrite order $<$ such that $l > r$ whenever there is a rule $l \to r$. The interest lies in the (syntactic) construction of an appropriate rewrite order. Our presentation follows that of Baader and Nipkow [BN98, ch. 5].

It appears that the *polynomial order* [MN70, Lan79] is among the earliest orders studied in this context. In polynomial orders, each function symbol (of arity n say) is assigned a polynomial in the indeterminates $X_1, \ldots, X_n$. By substitution this assigns to each term t a polynomial $\overline{t}$ say in its free variables. This induces an order on terms: polynomials (over a subset of $\mathbb{N} \setminus \{0\}$ in this case) may be compared by comparing their evaluation functions in the pointwise order; terms s and t may then be compared as $\overline{s}$ and $\overline{t}$. This is a powerful method for termination proofs (for instance, it is trivial that term size may be defined as a polynomial, and so this generalises simple size orders), but unfortunately it is not in general decidable for a given polynomial order whether or not $s < t$[2]. Furthermore, it may be shown that the lengths of reduction sequences of programs that can be found terminating using a polynomial order are bounded by a double exponential, and thus Ackermann's function cannot be proved terminating in this way.

A class of rewrite orders that is closely related to our order on closures is the class of *simplification orders* (and in particular the weakest such order, the *homeomorphic embed-*

[2]This is a consequence of undecidability of Hilbert's tenth problem (existence of integer roots of polynomials).

ding order). The homeomorphic embedding order is the closure of the subtree order under substitution and term construction (the former does not apply in our setting, and we do not use closure under term construction). The best-known simplification orders are *recursive path* orders [Pla78b, Pla78a] (with multiset status [Der87] or lexicographic status [KL80]) and *Knuth-Bendix* orders [KB70].

Recursive path orders are defined by an order $<$ on the function symbols of the TRS. Terms are ordered first by their root symbols, then by comparing subterms. The classes of recursive path orders arise through the different ways in which collections of subterms are compared: either as multisets of terms or as sequences (compared in the lexicographic order). Multiset or lexicographic path orders can prove termination of the Ackermann TRS, for which no polynomial order proves termination. Furthermore, these orders are decidable, given the order on function symbols. However it is still necessary to choose this latter order (or try all combinations, which is likely infeasible). Size-change termination (as adapted to term rewriting systems [TG03]) simulates much of the idea of lexicographic or multiset path orders, and is fully automated (without the need to search for an order on function symbols). However, SCT does not subsume recursive path orders, nor is it subsumed by such methods.

Knuth-Bendix orders are defined by assigning each term a *weight* (positive real number). It is assumed that all function symbol and variable are given weights; the weight of a term can then be deduced by adding the weights of all occurrences of function symbols and variables. Terms are then compared similarly to recursive path orders (that is, first comparing root nodes, then subterms), but the *weight* of each subterm is considered, rather than just the order relations between function symbols. Once more the order induced by the weight function is decidable, but this falls short of automation, as weights must still be chosen.

Finally, a more recent approach to termination proofs for term rewriting systems is the use of *dependency pairs* [AG00, GTSKF03, TGSK04]. A dependency pair is essentially the equivalent of a function call (it consists of a pair of terms (l, t), where t occurs in the right-hand side of a rule $l \to r$, and the root node of t is a function symbol). The *dependency graph* contains dependency pairs as nodes and edges $(s, t) \to (s', t')$ if (some substituted instance of) t can be rewritten to s'. Termination is proved by finding an appropriate order on terms such that every *cycle* in the dependency graph contains a dependency pair (s, t) such that $s > t$. The dependency graph is unfortunately not computable, and must be approximated. The search for an appropriate term order in the last step of the algorithm can include any of the orders shown above. The dependency pair method has proved to be a powerful method for termination proofs, and has further been shown to form a useful basis for automated proofs [GTSK05a].

Automatic Termination Analysis Most results on termination of term rewriting systems that we have mentionned above are aimed at providing effective techniques for *manual* termination proofs, and are not appropriate for automation. It is however certainly possible to offer *semi-automatic* termination proofs based on such criteria — for instance, relying on

the user to provide an order on function symbols in order to define the lexicographic path order. The dependency pair method lends itself naturally to automation, and has been implemented in a number of systems, in particular the Tsukuba Termination Checker [HM03] and the tool of Arts [Art00]. Finally, the AProVE system [GTSK05b, GTSKF04] is a recent system that employs dependency pairs together with other methods (including an equivalent of the size-change termination criterion [TG03, TG05]). To conclude this section we shall briefly describe AProVE as an example of systems for automatic TRS termination proofs.

AProVE offers most of the orders that we have described. Recursive path orders are available, represented by the embedding order (the simplest case), the lexicographic path order, the multiset path order, together with some variations (for instance, the lexicographic path order may be generalised to compare terms up to permutations). In these orders it is necessary to find a precedence on function symbols — this is achieved by a simple depth-first or breadth-first search. As the number of function symbols is finite this is guaranteed to terminate, but may be inefficient. Furthermore, the Knuth-Bendix order may be used in AProVE, as well as polynomial orders. For the latter, polynomials may be provided manually, or derived from the TRS by deriving inequalities between polynomials that should be satisfied, and attempting to solve these automatically.

Beyond the methods listed above (which the authors of AProVE dub "direct" termination proofs), AProVE offers termination proofs based on the size-change principle or the method of dependency pairs. The order used in the dependency pair method may be picked from any of the above orders (or rather a set of these may be selected, and the tool can search for an appropriate order). Finally, several power-increasing transformations [AG00, GTSKF03] may be applied to the dependency graph extracted from the program.

The design of AProVE derives from goals very different to those of purely SCT-based tools. In particular, implementations of the SCT principle for functional programs focus on fully automatic termination proofs. Our own method is somewhat intermediate — while the termination criterion is fully automatic, the call graph extraction is parametrised by user decisions (though the choice of abstract interpretation or of the depth parameter k requires less knowledge of the program than the choice of a sophisticated termination ordering). In contrast, due to the plethora of termination orders for term rewriting systems, systems such as AProVE focus on exhaustiveness and the *search* for a successful order (for instance, the choice of order on function symbols).

From Term Rewriting to Functional Programs In recent work [GSSKT06], Giesl *et al* show that termination tools for term rewriting systems, and in particular the AProVE system, can fruitfully be applied to termination of functional programs. In particular, a termination analysis of Haskell, thus handling both higher-order functions and laziness, is defined by an appropriate translation from Haskell programs to term rewriting systems.

Haskell programs are analysed for termination of the start term to normal form (in keeping with the behaviour of Haskell evaluators). Programs such as *take* n (*infinite* m) (where n

and m are free variables) can be handled. However, to prove termination of such programs the naive translation into term rewriting systems is not sufficient, and the authors instead define a *termination graph* from the program. This is obtained by expanding the evaluation tree of the start term, with a heuristic to collapse similar nodes (detecting a node carrying a term that is a substituted instance of a previous node) to guarantee finiteness. The resulting graph appears similar to a 0CFA call graph for the program.

The termination graph is used to generate a dependency pair problem, finiteness (termination) of which may then be proved by existing techniques. While the presence of higher-order functions in the original program can cause the dependency pair problem to be likewise higher-order, this can be transformed into a first-order problem by explicit addition of application symbols. Finally, it is observed (as in our case) that unevaluated arguments make the observation of decreases problematic. The solution proposed is to guarantee that arguments are always evaluated as much as possible (without introducing nontermination), to allow decreases in argument sizes to be recorded.

7.3 Future Work

Let us conclude with a brief discussion of some of the future directions of research that this work may spawn. Three categories of future work may be identified: lifting our analyses to a full termination test for mainstream functional languages, investigating the applications of the SCT analysis for theorem proving, and finally further study of the expressiveness of static termination analysis.

Applications We have argued that our analyses form a powerful and useful basis for termination analysis of functional languages, both strict and lazy. However, as it stands the size-change termination analysis cannot directly be applied to the whole of a language such as Objective Caml. Realising the potential of the analysis requires handling such a language in its entirety, to provide a termination checker for real functional programs.

The work required in applying the SCT analysis to all of OCaml is threefold. First, it is necessary to extend the program transformation techniques that we have used to translate a subset of OCaml into our core language to handle all of OCaml. This is essentially straightforward (though somewhat arduous) — the main language feature that we have ignored from the purely functional core is the module system.

The second requirement points to a vast field of research. The OCaml languages contains *imperative features*: mutable reference cells may be used to implement mutable state. The extension of the SCT analysis to handle mutable cells and assignment for primitive constants such as integers is fairly direct, as the (finite) set of constant reference cells may be modelled in the same way as function parameters. However, the extension to *arbitrary* reference cells is much more involved. This requires a treatment of deep structures in the heap, which may be cyclic, and is conceptually challenging. It is not yet known whether techniques for

termination proofs with mutable heap structures [BCDO06] may be applied to our analysis. Finally, the exception mechanism of ML-like languages substantially complicates the call graph construction.

Finally, a number of practical issues must be resolved. It is essential to incorporate the extension of the SCT analysis to the integers to deal with real programs. Furthermore, efficiency concerns are paramount: some of the analyses that we have presented are clearly too expensive to apply directly to a large program. Open questions include whether the analyses can be made modular in any way, and whether it is possible to apply a more expensive analysis only to parts of the program that require it.

Theorem Proving There is an important connection between termination analysis and theorem proving: the definition of functions in theorem provers requires termination checking, and likewise theorem proving can be a natural environment for automated termination proofs. The integration of an analysis such as SCT in a theorem proving environment thus has many potential benefits, both on the theorem proving side and on the analysis side.

The SCT analysis has clearly defined boundaries: it may easily be seen that certain programs are definitely beyond the scope of this method. This may range from programs such as McCarthy's famous 91 function, whose termination stems from a highly nontrivial partial correctness property; to functions whose termination implies a solution to a current conjecture (the Goldbach conjecture may be used, for example). However, such badly-behaved functions may represent only a small fraction of a program. A practical realisation of the SCT analysis should be able to prove termination of a program, *up to* determining whether a (hopefully small) number of functions terminate. Such a system could generate proof obligations that may then be resolved with the aid of a theorem prover. This approach has the advantage of combining the purely automatic nature of static termination analysis with the flexibility offered by a theorem proving approach to termination checking.

The other benefit of integration within a theorem proving environment is the use of the SCT analysis as a termination checker for defining functions in the theorem prover. The SCT criterion emerges as a natural choice: a fully automatic analysis handling all the simple cases (and, as above, generating proof obligations for failing cases) would greatly simplify the task of defining functions used in a proof. As it stands, examples handled by the first-order SCT analysis often require hints to the termination checker to be handled in a theorem prover. Higher-order programs are even more difficult to handle in this context, and may require extensive proof of termination. The challenge in making the SCT analysis available in a theorem proving environment is that a simple yes / no answer to termination is not sufficient in this case. To ensure the soundness of the theorem proving task, it is crucial that the proof be verifiable, and so the termination analyser should not act as an oracle. Instead it is desirable (or necessary) for the analyser to produce a termination *proof* automatically, essentially specialising the soundness proof for the analysis to a particular program.

Expressiveness Finally, there are many avenues of work open in analysing the expressiveness of the SCT analysis, in particular in relation to well-known classes of terminating programs. Our work has settled a number of questions regarding the relationship of both typed λ-calculi and the class of acyclic attribute grammars to size-change termination. In particular, we have shown that no k-bounded CFA analysis proves termination of all simply-typed λ-expressions (surprisingly, as it was conjectured that 0CFA could capture this), but that every simply-typed (indeed, any terminating) λ-expression was k-bounded CFA terminating for some k. The case of acyclic attribute grammars is different: we have exhibited an acyclic grammar which is not terminating under any control-flow analysis for SCT.

The relationship of the class of simply-typed λ-expressions with SCT remains largely unknown. It is not known whether any SCT analysis can *simultaneously* recognise all simply-typed terms (though each individual term is recognised by some analysis). The question of whether this exists, and whether a natural analysis exists if that is the case, can shed light both on the SCT criterion, as well as simple types. Furthermore, it is interesting to consider restricted subsets of the simply-typed λ-calculus. For instance, is there a naturally occurring such subset which is recognised by the 0CFA analysis? The general problem of characterising such classes as the set of simply-typed terms in 0CFA remains unsolved.

The case of attribute grammars opens different questions on the characterisation of expressiveness. For, not all acyclic (or indeed all absolutely acyclic) attribute grammars are size-change terminating. The difficulty therefore cannot be resolved within our static analysis framework (any static analysis with finite abstract domain, and k-limited graphs). This opens a number of questions, starting with the determination of whether a more refined order on values can be used to prove terminations of such programs (potentially leading to an improvement over our analysis for more general programs also) — it is not clear that any such order can be given. This may point the way to a more general framework than the size-change termination point of view we have adopted. Indeed, it seems that the failure of SCT on attribute grammars is linked to the problem of non-monotonic descent, which is fundamentally difficult to express satisfactorily in the SCT framework (and leads to undecidability of the SCT criterion in general).

Appendix A

Semantic Issues

A.1 Properties of Inference Systems

In Chapter 2, we have informally introduced the notions of *determinism* and *totality* for inference systems. These properties are required for our proof that nontermination may be represented as an infinite sequence of calls, for an appropriate notion of function call. In this section we shall formalise these properties, and define the transformation that defines the call relation.

A.1.1 Inference Rules

We will consider inference systems for defining a relation, say $\Downarrow$ of type $A \times B$ (here $A = \mathit{State}$ and $B = \mathit{Value}$). Such an inference system is composed of inference rules of the form:

$$\frac{\varphi_1(s) \Downarrow v_1 \quad \varphi_2(s, v_1) \Downarrow v_2 \qquad \cdots \qquad \varphi_n(s, v_1, \ldots, v_{n-1}) \Downarrow v_n}{s \Downarrow \psi(s, v_1, \ldots, v_n)}\, P(s) \wedge Q_1(s, v_1) \wedge \cdots \wedge Q_n(s, v_n)$$

where:

1. For each i, $\varphi_i : A \times B^{i-1} \to A$.
2. $\psi : A \times B^n \to B$
3. P is a predicate, or equivalently $P : A \to \mathbb{B}$
4. For each i, $Q_i : A \times B \to \mathbb{B}$

The φ_i and ψ functions may be partial, but we insist that their results are defined whenever all preceding premisses match. More precisely, $\varphi_i(s, v_1, \ldots, v_{i-1})$ is defined whenever $P(s)$ holds and $Q_j(s, v_j)$ holds for each $j < i$. Likewise, we require that $\psi(s, v_1, \ldots, v_n)$ be defined whenever $P(s) \wedge \bigwedge_i Q_i(s, v_i)$ holds.

We will write such an inference rule as a triple $\mathcal{R} = (P, \sigma, \psi)$, where $\sigma = \langle \varphi_i, Q_i \rangle_{i=1}^{n}$ in the following.

The φ_i are the *premiss functions* of $\mathcal{R}$, while the Q_i are the *premiss predicates*. The pair (φ_i, Q_i) is the i^{th} *premiss* of $\mathcal{R}$.

Definition A.1. Let $\mathcal{R} = (P, \sigma, \psi)$ be an inference rule with notation as above. Then $\mathcal{R}$ *matches* at s iff $P(s)$ holds. Furthermore, $\mathcal{R}$ *applies* at $(s, \langle v_1, \ldots, v_n \rangle)$ if $P(s) \wedge \bigwedge_i Q_i(s, v_i)$ holds.

For instance, consider the rule for the **if** construct, where the guard evaluates to **true**:

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

Here (in the above notation) $n = 2$, $P(s) \equiv (\exists e_g, e_t, e_f, \rho) s = \textbf{if } e_g \textbf{ then } e_t \textbf{ else } e_f : \rho$, $\varphi_1(s) = e_g : \rho$ and $\varphi_2(s, v_1) = e_t : \rho$ (where $P(s)$ holds, and $s = \textbf{if } e_g \textbf{ then } e_t \textbf{ else } e_f : \rho$). Also, $Q_1(s, v) \equiv v = \textbf{true}$, while $Q_2(s, v) \equiv \mathrm{T}$. Finally, $\psi(v_1, v_2) = v_2$.

A.1.2 Left-to-Right Evaluation

In our definition of inference systems, the i^{th} premiss function φ_i takes the results of evaluating the first $i - 1$ premisses as parameters. This is necessary to express rules such as the function application rule:

$$\frac{e_1 : \rho \Downarrow (\!| f : \mu |\!) \quad e_2 : \rho \Downarrow w \quad \mathit{Bind}\ (\!| f : \mu |\!)\ w \Downarrow v}{e_1 e_2 : \rho \Downarrow v}\ {\scriptstyle (\!| f:\mu |\!) \in \mathit{Closure}'}$$

where a state whose evaluation is required (here $\mathit{Bind}\ (\!| f : \mu |\!)\ w$) depends on the values that previous states evaluate to.

This flexibility implies a directed view of each inference rule: the premisses are necessarily evaluated in a *left-to-right* order; indeed no other evaluation order is possible, as evaluation of the first $i - 1$ states appearing as premisses is necessary in order to find the i^{th} state to evaluate.

This left-to-right interpretation of each inference rule suggests a focus on *sequential* languages. However, not every semantics definable in our framework gives rise to a language that admits a sequential implementation. This is undesirable, as we shall require a sequential language for our termination criterion (though this is necessary for precision, rather than soundness — in a parallel language, more programs may terminate).

The archetypal example of a language construct that can be given no standard sequential implementation is the *parallel-or* construct $\|$. This is defined by the following:

$$\frac{e_1 : \rho \Downarrow \textbf{true}}{e_1 \| e_2 : \rho \Downarrow \textbf{true}} \qquad \frac{e_2 : \rho \Downarrow \textbf{true}}{e_1 \| e_2 : \rho \Downarrow \textbf{true}} \qquad \frac{e_1 : \rho \Downarrow \textbf{false} \quad e_2 : \rho \Downarrow \textbf{false}}{e_1 \| e_2 : \rho \Downarrow \textbf{false}}$$

An expression $e_1 \| e_2$ is evaluated by evaluating both e_1 and e_2 in *parallel*. If one of these reduces to **true** then $e_1 \| e_2$ reduces to **true** (short-circuit evaluation). It is invalid to attempt to implement this by evaluating either of e_1 or e_2 first, as either of these may be nonterminating. Indeed it is well-known that this operator cannot be implemented in languages such as Haskell.

We wish to explicitly forbid language features such as $\|$ that require parallel evaluation. To this end, we define the *left-to-right* property (LR) for inference systems:

Definition A.2 (LR)**.** An inference system has the *LR* property if the following holds. Suppose that $\mathcal{R} = (P, \langle(\varphi_i, Q_i)\rangle_{i=1}^{n}, \psi)$ and $\mathcal{R}' = (P', \langle(\varphi'_i, Q'_i)\rangle_{i=1}^{m}, \psi')$ are rules that match some state s. Suppose further that for each $i < k$, $Q_i \equiv Q'_i$ and $\varphi_i = \varphi'_i$ (where $k \geq q$). Then $Q_k \equiv Q'_k$.

Note that a consequence is that in an LR system, the *first* premiss functions of all rules that match a same state must be identical.

Let us consider the computational intuition behind this definition. Suppose that s is a state that we aim to evaluate, and suppose that we have evaluated i states $s_j = \varphi_j(s)$ $(j = 1, \ldots, i)$. Then *either* the predicates Q_i of the rules that apply are distinct (so that some choice may be made between the different rules), *or* the next state that we must evaluate is the same for each rule.

In effect, this guarantees that we may evaluate states by evaluating premisses from left to right, as the name implies, across *all* inference rules, rather than just one rule. This intuition is only valid, however, when the inference is also *deterministic*, as will be introduced below.

A.1.3 Determinism

The left-to-right property serves to rule out language constructs that do not admit a sequential implementation. It may still be the case, however, that more than one rule applies to a given state. Again, such cases go against our notion of the computation described by the inference system defining $\Downarrow$, and will prove inconvenient in our definition of the termination criterion.

We therefore introduce the *determinism* property to remedy this. This guarantees that for any state s, there is only ever one rule applying to prove a judgement of the form $s \Downarrow v$. Certainly this property implies that the $\Downarrow$ relation is a (partial) function, but is stronger, as it is easy to define an inference system defining a function $\Downarrow$, but in which several rules may apply to a state

Definition A.3 (Determinism)**.** An inference system with the LR property is *deterministic* if the following holds. Suppose that $\mathcal{R} = (P, \langle(\varphi_i, Q_i)\rangle_{i=1}^{n}, \psi)$ and $\mathcal{R}' = (P', \langle(\varphi'_i, Q'_i)\rangle_{i=1}^{m}, \psi')$ are rules that match some state s. Also, suppose that the first premisses that are not identical in both rules are in position k. Then $Q_k \wedge Q'_k \equiv \mathrm{F}$.

Note that by the LR property, in the above case $\varphi_i = \varphi'_i$ and $Q_i \equiv Q'_i$ for each $i < k$. Furthermore, $\varphi_k = \varphi'_k$. The intention is that there may be no overlap between differing predicates: the first $k-1$ premisses are entirely identical, and the k^{th} premisses have disjoint predicates.

It is easy to see that this definition ensures that no two distinct rules can ever match at a given state. The statement of determinism is perhaps stronger than necessary, as it is possible that a system in which only one rule ever applies fail this test. However, this reflects our intuition about the computational behaviour of an implementation, and a more general notion of determinism will not be required.

As an example, consider the rules for evaluating an **if** expression shown previously. There are four rules that match an **if** case, and these have pairwise disjoint first premiss predicates, as required — the first premiss evaluates the value of the guard to a value v, and the corresponding predicates are $v = \mathbf{true}$, $v = \mathbf{false}$, $v \in \mathit{Value} \setminus \{\mathbf{true}, \mathbf{false}\}$ and $v \notin \mathit{Value}$.

Let us now prove that the determinism property yields our desired result:

Lemma A.4. *Let $\Downarrow$ be defined by an LR deterministic inference system. Then for any $s \in A$, there is at most one proof tree with conclusion of the form $s \Downarrow v$.*

A trivial corollary is the following:

Corollary A.5. *If $\Downarrow$ is defined by an LR deterministic inference system, and $s \Downarrow v$ and $s \Downarrow v'$, then $v = v'$.*

Proof of Lemma A.4. We prove that whenever $s \Downarrow v$ with proof tree Π, there is no proof tree distinct from Π with conclusion $s \Downarrow v'$ (for any $v' \in B$). We proceed by induction on Π.

Consider the last rule used in Π. This has conclusion $s \Downarrow v$. Suppose for a contradiction that there is another proof tree Π' with conclusion $s \Downarrow v'$. We have two cases, depending on whether the last rules used in Π and Π' are identical or not.

Suppose first that the last rules of Π and Π' are identical, say $\mathcal{R} = (P, \sigma, \psi)$. Then as Π and Π' are distinct, some premiss of $\mathcal{R}$ has different proof trees in Π and Π', say the k^{th} premiss is the first such premiss. Then as the first $k-1$ proof trees are identical, the results of evaluating the first $k-1$ states in each rule are pointwise identical, say $v_1, \ldots, v_{k-1}$. Then let $s_k = \varphi_k(s, v_1, \ldots, v_{k-1})$. Then $s_k \Downarrow$ has two distinct proof trees, say Δ in Π and Δ' in Π'. However, Δ is a subtree of Π, and so by induction there is no such tree Δ', a contradiction.

Now suppose that the last rules of Π and Π' are distinct, say $\mathcal{R} = (P, \langle(\varphi_i, Q_i)\rangle_{i=1}^{n}, \psi)$ and $\mathcal{R}' = (P', \langle(\varphi'_i, Q'_i)\rangle_{i=1}^{m}, \psi')$ (resp.). Then both rules match at s. Let k be the index of the first premiss to differ between both rules. Then by determinism $Q_k \wedge Q'_k \equiv \mathrm{F}$, while $\varphi_k = \varphi'_k$.

However, for each i let v_i (resp. v'_i) be the result of evaluating $s_i = \varphi_i(s, v_1, \ldots, v_{i-1})$ (resp. $s'_i = \varphi'_i(s, v'_1, \ldots, v'_{i-1})$). We show that $s_i = s'_i$ and $v_i = v'_i$ for all $i \leq k$ by induction on k. Suppose that this holds for $k-1$. We must show that $s_k = s'_k$ and $v_k = v'_k$. Note that $s_k = \varphi_k(s, v_1, \ldots, v_{k-1}) = \varphi_k(s, v'_1, \ldots, v'_{k-1})$ by the inductive hypothesis. As $\varphi_k = \varphi'_k$, this

gives that $s_k = s'_k$. But $s_k \Downarrow v_k$ with proof tree a subtree of Π, so by the inductive hypothesis (corollary) we have that as $s_k = s'_k \Downarrow v'_k$ also, $v'_k = v_k$.

We have therefore shown that $v'_k = v_k$. But we know that $Q_k(s, v_k) \wedge Q'_k(s, v'_k)$ holds, as both $\mathcal{R}$ and $\mathcal{R}'$ both match. This is a contradiction as $Q_k \wedge Q'_k \equiv \mathrm{F}$. □

A.1.4 Totality

We have introduced the LR and determinism property, which together guarantee that we may give a clear computational interpretation to the reduction rules given. Furthermore, we have shown that an LR deterministic system necessarily defines a (partial) function.

The totality property that we introduce in this section is to some extent dual to the previously defined properties. We aim to show the *existence* of proof trees under certain conditions, rather than their uniqueness.

The use of the word "totality" in this case may be a misnomer, as a total (deterministic) inference system will *not* in general define a total function $\Downarrow$. Indeed in our setting it is most undesirable for a semantics to define a total function — if $\Downarrow$ is total, then every program terminates!

The totality property aims to capture the difference between the semantics shown in Figures 2.2 and 2.3, namely the introduction of *error rules*. In the former semantics, evaluation can get "stuck" on ill-typed or partial operations. In the latter semantics, the only states for which $s \not\Downarrow$ are those states whose evaluation is nonterminating.

Of course, the above is not a mathematical statement, as we do not have an independent definition of nontermination. However, the totality property makes this intuition more precise.

Definition A.6 (Totality)**.** An inference system is *total* if: for any s, if $\varphi \mapsto v_\varphi$ is an assignment of values to the premisses of rules that match s, then some rule $\mathcal{R} = (P, \langle(\varphi_i, Q_i)\rangle_{i=1}^{n}, \psi)$ applies at $(s, \langle v_{\varphi_1}, \ldots, v_{\varphi_n}\rangle)$

This definition encodes the fact that for any state s, whatever assignment of values to the premisses of rules matching s, some rule will apply. This guarantees that if all the states that s depends on reduce to values, then some rule will apply. The connection with nontermination is therefore apparent: the only case in which $s \not\Downarrow$ is when evaluation of s depends on evaluation of s', where $s' \not\Downarrow$.

A.2 Semantics with Error Values

The use of error values to give reductions for ill-typed or partial operations was introduced in Section 2.2.3, excluding *error propagation* rules. For reference, we give the full inference system for evaluation with error rules in this section. The semantics of successful evaluation is shown in Figure A.1. The rules for introducing error values are shown in Figure A.2.

Values, Variables and Constants

$$\frac{}{v \Downarrow v}\ {\scriptstyle v \in Value} \qquad \frac{}{x : \rho \Downarrow \rho(x)} \qquad \frac{}{c : \rho \Downarrow c}$$

Function References and Closures

$$\frac{}{f : \rho \Downarrow (\!|f : \emptyset|\!)} \qquad \frac{Body(f) : \rho \Downarrow v}{(\!|f : \rho|\!) \Downarrow v}\ {\scriptstyle \mathrm{dom}\,\rho = Params(f)}$$

Primitive Operators

$$\frac{(\forall i) e_i : \rho \Downarrow c_i \quad Apply\ op\ \langle c_1, \ldots, c_n \rangle = c}{op(e_1, \ldots, e_n) : \rho \Downarrow c}\ {\scriptstyle (\forall i) c_i \in PrimConst \atop c \notin Error}$$

Constructors and Pattern Matching

$$\frac{(\forall i) e_i : \rho \Downarrow v_i}{\mathsf{C}(e_1, \ldots, e_n) : \rho \Downarrow \mathsf{C}(v_1, \ldots, v_n)} \qquad \frac{e : \rho \Downarrow \mathsf{C}_l(v_1, \ldots, v_{n_l}) \quad e_l : \rho \oplus \{x_j^l \mapsto v_j\}_{j=1}^{n_l} \Downarrow v}{\textbf{match}\ e\ \textbf{with}\ \left\langle \mathsf{C}_i(x_1^i, \ldots, x_{n_i}^i) \to e_i \right\rangle_{i=1}^{k} : \rho \Downarrow v}$$

Conditionals

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

Function Application

$$\frac{e_1 : \rho \Downarrow (\!|f : \mu|\!) \quad e_2 : \rho \Downarrow w \quad Bind\ (\!|f : \mu|\!)\ w \Downarrow v}{e_1 e_2 : \rho \Downarrow v}\ {\scriptstyle (\!|f:\mu|\!) \in Closure' \atop w \in Value}$$

Program Template Variables

$$\frac{}{X : \rho \Downarrow \mathfrak{V}(X)}\ {\scriptstyle \text{with valuation } \mathfrak{V}}$$

Figure A.1: Semantics: Successful Evaluation

Finally, Figure A.3 gives the error propagation rules. These reflect the fact that error values act as uncatchable exceptions, so that if any state evaluates to an error, the execution of the program immediately aborts (returning this error value).

Let us now state the essential properties of this semantics:

Lemma A.7. *The semantics given in Figures A.1, A.2 and A.3 is deterministic and total, and has the left-to-right property.*

One should note that this implies that the semantics in Figure 2.2 is also deterministic, as any subset of a deterministic semantics is deterministic. However this is certainly not a total semantics.

Proof of Lemma A.7. The LR, determinism and totality properties are proved by checking all possible cases, which is lengthy and unilluminating. We first observe that there are only five groups of rules for which more than one rule may match a given state:

Primitive Operators

$$\frac{(\forall j < i) e_j : \rho \Downarrow c_j \in \mathit{PrimConst} \quad e_i : \rho \Downarrow v_i \notin \mathit{PrimConst}}{op(e_1, \ldots, e_n) : \rho \Downarrow \mathit{PrimopErr}}\, i \leq n$$

$$\frac{(\forall i) e_i : \rho \Downarrow c_i \in \mathit{PrimConst} \quad \mathit{Apply\ op}\ \langle c_1, \ldots, c_n \rangle = \mathit{PrimopErr}}{op(e_1, \ldots, e_n) : \rho \Downarrow \mathit{PrimopErr}}$$

Pattern Matching

$$\frac{e : \rho \Downarrow v \in \mathit{Value} \quad (\nexists i, v_j) v = \mathsf{C}_i(v_1, \ldots, v_{n_i})}{\mathbf{match}\ e\ \mathbf{with}\ \left\langle \mathsf{C}_i(x_1^i, \ldots, x_{n_i}^i) \rightarrow e_i \right\rangle_{i=1}^{k} : \rho \Downarrow \mathit{PatMatchErr}}$$

Conditionals

$$\frac{e_g : \rho \Downarrow v \in \mathit{Value} \setminus \{\mathbf{true}, \mathbf{false}\}}{\mathbf{if}\ e_g\ \mathbf{then}\ e_t\ \mathbf{else}\ e_f : \rho \Downarrow \mathit{TypeErr}}$$

Function Application

$$\frac{e_1 : \rho \Downarrow v \in \mathit{Value} \setminus \mathit{Closure}'}{e_1 e_2 : \rho \Downarrow \mathit{TypeErr}}$$

Figure A.2: Semantics: Error Introduction

Primitive Operators

$$\frac{(\forall j < i) e_j : \rho \Downarrow v_j \in \mathit{PrimConst} \quad e_i : \rho \Downarrow e \in \mathit{Error}}{op(e_1, \ldots, e_n) : \rho \Downarrow e}\, i \leq n$$

Constructor Application

$$\frac{(\forall j < i) e_j : \rho \Downarrow v_j \in \mathit{Value} \quad e_i : \rho \Downarrow e \in \mathit{Error}}{\mathsf{C}(e_1, \ldots, e_n) : \rho \Downarrow e}\, i \leq n$$

Pattern Matching

$$\frac{e : \rho \Downarrow e \in \mathit{Error}}{\mathbf{match}\ e\ \mathbf{with}\ \left\langle \mathsf{C}_i(x_1^i, \ldots, x_{n_i}^i) \rightarrow e_i \right\rangle_{i=1}^{k} : \rho \Downarrow e}$$

Conditionals

$$\frac{e_g : \rho \Downarrow e \in \mathit{Error}}{\mathbf{if}\ e_g\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 : \rho \Downarrow e}$$

Function Application

$$\frac{e_1 : \rho \Downarrow e \in \mathit{Error}}{e_1 e_2 : \rho \Downarrow e} \qquad \frac{e_1 : \rho \Downarrow v \in \mathit{Closure}' \quad e_2 : \rho \Downarrow e \in \mathit{Error}}{e_1 e_2 : \rho \Downarrow e}$$

Figure A.3: Semantics: Error Propagation

1. Primitive operator application: successful evaluation, n error propagation rules (where n is the arity of the primtive operator), and the primitive operator evaluation failure case.
2. Constructor application: successful evaluation and the n error propagation rules, where n is the arity of the constructor.
3. Pattern matching: successful match, match failure and error propagation
4. Alternation: **then** branch evaluation, **else** branch evaluation, type error and error propagation
5. Function application: successful application, type error, error propagation (operator) and error propagation (operand).

For each of these groups we must check the LR property, determinism and totality. This is straightforward and thus omitted. □

A.3 Sequentialising Nontermination

A.3.1 The Sequentialising Transformation

We will now introduce the *sequentialising* transformation. Given an arbitrary inference system defining a relation $\Downarrow$ of type $A \times B$, this defines a relation $\rightarrow$ of type $A \times A$. The intended meaning of $s \rightarrow s'$ ("s calls s'") is: in order to evaluate s (that is, in order to deduce $s \Downarrow v$ for some v) it is necessary to evaluate s' (to show that $s' \Downarrow v'$ for some v'). While the sequentialising transformation applies to any inference system, the additional properties of determinism and totality will be necessary for this intended meaning to hold, as we shall see.

Definition A.8. Given an inference system defining $\Downarrow$, the associated *call* relation $\rightarrow$ is defined by an inference system derived as follows. For each rule

$$\frac{\varphi_1(s) \Downarrow v_1 \quad \varphi_2(s, v_1) \Downarrow v_2 \quad \cdots \quad \varphi_n(s, v_1, \ldots, v_{n-1}) \Downarrow v_n}{s \Downarrow \psi(s, v_1, \ldots, v_n)}\, P(s) \wedge Q_1(s, v_1) \wedge \cdots \wedge Q_n(s, v_n)$$

we introduce n rules of the form (for $1 \leq i \leq n$):

$$\frac{\varphi_1(s) \Downarrow v_1 \quad \varphi_2(s, v_1) \Downarrow v_2 \quad \cdots \quad \varphi_{i-1}(s, v_1, \ldots, v_{i-2}) \Downarrow v_{i-1}}{s \rightarrow \varphi_i(s)}\, P(s) \wedge Q_1(s, v_1) \wedge \cdots \wedge Q_{i-1}(s, v_{i-1})$$

For instance, consider the rule for evaluating an **if** expression, where the guard evaluates to **true**. This gives rise to two inference rules:

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

A.3.2 Sequentialisation and Nontermination

The upshot of the definition of the determinism, LR and totality properties, as well as the sequentialising transformation, is that we can now state a general result characterising nontermination as an infinite sequence of calls. This result (which only applies to deterministic and total inference systems) makes precise our intuition that in such systems, if $s \not\Downarrow$ this is necessarily because evaluation of s is nonterminating.

Proposition A.9. *Let $\Downarrow$ be defined by a total inference system, and let $\to$ be the associated call relation. Then for every state s, if $s \not\Downarrow$ then there is an infinite sequence of calls $s = s_0 \to s_1 \to s_2 \to \cdots$.*

We first define some useful notation. If s is a state, we define $\nu(s)$ to be the maximum length of a call sequence $s = s_0 \to s_1 \to \cdots$ starting at s. If there is an infinite call sequence we write $\nu(s) = \infty$.

Proof. Let us prove that whenever $\nu(s)$ is finite, $s \Downarrow$ by induction on $\nu(s)$. Suppose that for all s' with $\nu(s') < \nu(s)$, $s' \Downarrow$. In particular if $s \to s'$ then $\nu(s') < \nu(s)$, so $s' \Downarrow$.

Let S be the set of rules that match s, and for each rule (say $\mathcal{R}_k$) in S let $\left\langle (\varphi_i^k, Q_i^k) \right\rangle_{i=1}^{n_i}$ be the premisses of $\mathcal{R}_k$.

Suppose for a contradiction that no rule in S applies to show that $s \Downarrow v$ (for some v). Then for each rule $\mathcal{R}_k$, there is a premiss $(\varphi_{i_k}^k, Q_{i_k}^k)$ which is not satisfied, wlog i_k is the least such index for each k. Now by minimality of i_k, the first $i_k - 1$ premisses of $\mathcal{R}_k$ are satisfied. Hence we may find sequences $s_i^k = \varphi_i^k(s, v_1^k, \ldots, v_{i-1}^k)$ $(i \leq i_k)$ and v_i^k $(i < i_k)$, such that $s_i^k \Downarrow v_i^k$ for each $i < i_k$. Furthermore, for each $i < i_k$, $Q_i^k(s, v_i^k)$.

For any k, by the above $s \to s_{i_k}^k$. Hence by the inductive hypothesis, $s_{i_k}^k \Downarrow$ for each k. Let X_k be the set of value that $s_{i_k}^k$ reduces to, that is $X_k = \{v \mid s_{i_k}^k \Downarrow v\} \neq \emptyset$. By choice of i_k, for all $v \in X_k$, $Q_{i_k}^k(s, v)$ is false (otherwise the i_k^{th} premiss of $\mathcal{R}_k$ would apply).

Now pick an assignment of values to the premiss functions of rules in S, say $\varphi \mapsto w^\varphi$, such that for each k, $w^{\varphi_{i_k}^k} \in X_k \neq \emptyset$. By totality some rule applies at this assignment, say $\mathcal{R}_k$.

Then all premisses of $\mathcal{R}_k$ apply, so that in particular $Q_{i_k}(s, w^{\varphi_{i_k}^k})$ holds. But $w^{\varphi_{i_k}^k} \in X_k$, contradicting the fact that $Q_{i_k}^k(v)$ is false for all $v \in X_k$. □

Proposition A.9 is all that is required for a sound termination criterion: if we can show that all call sequences are finite, then termination follows. This only relies on totality.

For precision, however, it is desirable to have a converse to this result. For, we expect that an infinite call sequence does imply nontermination — otherwise, we may *a priori* rule out some terminating programs, which is undesirable. The converse of Proposition A.9 is given below. While it does not rely on totality, it does require determinism and the left-to-right property (in fact, this fails in the parallel-or example shown previously).

Proposition A.10. *Let $\Downarrow$ be defined by a left-to-right deterministic inference system, and let $\to$ be the associated call relation. Then for every state s, if there is an infinite sequence of calls $s = s_0 \to s_1 \to s_2 \to \cdots$, then $s \not\Downarrow$.*

Proof. We prove that whenever $s \Downarrow$, $\nu(s)$ is finite by induction on the proof of $s \Downarrow$. Consider the last rule used in the proof, say $\mathcal{R} = (P, \langle(\varphi_i, Q_i)\rangle_{i=1}^{n}, \psi)$. Then as before we may define sequences s_i and v_i such that $s_i = \varphi_i(s, v_1, \ldots, v_{i-1})$ and $s_i \Downarrow v_i$ for each i. Furthermore $Q_i(s, v_i)$ holds for each i.

> **Lemma.** *Whenever $s \to s'$, $s' = s_i$ for some i.*
>
> Suppose that $s \to s'$. Then there is a rule $\mathcal{R}' = (P', \langle\varphi'_i, Q'_i\rangle_{i=1}^{m}, \psi')$ such that the first $p-1$ premisses of the rules are satisfied, and s' is given by the p^{th} premiss. That is, we may define $t_i = \varphi'_i(s, w_1, \ldots, w_{i-1})$ for $i \leq p$ and $t_i \Downarrow w_i$ for each $i < p$, such that $s' = t_p$. Furthermore for each $i < p$, $Q'_i(s, w_i)$ holds.
>
> Now both $\mathcal{R}$ and $\mathcal{R}'$ match at s. Hence by the LR property we may find a maximal common prefix of the premisses of these rules. That is, for some $q \geq 1$, $\varphi_i = \varphi'_i$ for all $i \leq q$, and $Q_i \equiv Q'_i$ for all $i < q$. Furthermore, by determinism (corollary to Lemma A.4) this implies that $s_i = t_i$ for all $i \leq q$, while $v_i = w_i$ for all $i < q$.
>
> We may now apply determinism to $\mathcal{R}$ and $\mathcal{R}'$ to conclude that $Q_q \wedge Q'_q \equiv \mathrm{F}$.
>
> We show that $p \leq q$. Note that this implies that $s' = t_p = s_p$, as required. Suppose for a contradiction that $p > q$.
>
> As $q < p$, $t_q \Downarrow w_q$, where $Q'_q(s, w_q)$ holds. But $t_q = s_q$ and $w_q = v_q$ by determinism as noted above. Therefore $Q'_q(s, v_q)$ holds. But $Q_q(s, v_q)$ holds also. This is a contradiction as $Q_q \wedge Q'_q \equiv \mathrm{F}$. □

Hence if $s \to s'$, then $s' = s_i$ (and indeed $s \to s_i$ for each i). Thus $\nu(s) = 1 + \max_i \nu(s_i)$. But for each i, $s_i \Downarrow v_i$ with smaller proof, so by the inductive hypothesis $\nu(s_i)$ is finite. Hence $\nu(s)$ is finite. □

The sequentialised form of the semantics is given in Figure 2.4 (p. 24).

Appendix B

Well-Founded Relations

Termination proofs, whether using the size-change termination criterion or not, invariably rely on proving infinite sequences of states are impossible as such sequences would cause some value to decrease infinitely in a well-founded order. It is not strictly necessary for the comparison relation used to be a well-founded order in the traditional sense, however. In this appendix we briefly define the minimal properties we require of such comparison relations. These definitions are essentially standard, and are presented for reference only.

B.1 Well-founded Relations

A well-founded order $<$ on a set X is a partial order on X without *infinite decreasing chains*. That is, infinite chains $x_0 > x_1 > x_2 > \cdots$ of inequalities are impossible.

The definition of a well-founded relation generalises the notion of well-founded orders, and provides the minimal property that we require for SCT analysis.

Definition B.1. Let A be a set. A relation $R \subseteq A \times A$ is said to be *well-founded* if there is no infinite sequence $(x_i)_{i\in\mathbb{N}}$ of elements of A such that $x_{i+1}\ R\ x_i$ for each i.

As a consequence of Definition B.1, a well-founded relation is *irreflexive*: $\neg(x\ R\ x)$ for any x. A well-founded relation need not be transitive, however, and therefore need not be a partial order.

Well-founded relations are the analogues of well-founded strict partial orders. We now define a similar generalisation of well-founded *reflexive* partial orders. A reflexive partial order can be defined from its strict counterpart as $x \leq y$ iff $x < y$ or $x = y$. Here we relax this definition by allowing an appropriate equivalence relation in place of equality.

Definition B.2. Let A be a set and R a well-founded relation on A. An equivalence relation $\sim$ on A is *compatible* with R if whenever $x\ R\ y$ and $x \sim x'$, then $x'\ R\ y$.

The pair $(R, \sim)$ is a *non-strict well-founded relation* on A. This gives rise to a relation $R_\sim$ on A, defined by $x\ R_\sim\ y$ iff $x\ R\ y$ or $x \sim y$.

Note that we have only required $\sim$ to be compatible with R on the left — no assertion is made about x and y', where $x\ R\ y$ and $y \sim y'$. This asymmetry may be explained by the fact that the definition of well-foundedness is likewise asymmetrical, since only infinite decreasing chains are considered, not increasing chains. Right-compatibility of $\sim$ and R will not be required.

The upshot of this definition is the following property:

Proposition B.3. *Suppose that $(R, \sim)$ is a non-strict well-founded relation on A. Then there is no infinite sequence x_0, x_1, ... of elements in A such that $x_{i+1}\ R_\sim\ x_i$ for all i, and $x_{i+1}\ R\ x_i$ for infinitely many i.*

Proof. Suppose for a contradiction that $(x_i)_{i\in\mathbb{N}}$ is such a sequence. Let $S = \{i_j\}_{j\in\mathbb{N}}$ be the set of indices such that $x_{i_j+1}\ R\ x_{i_j}$ (wlog $i_0 < i_1 < \cdots$). This set is infinite by assumption. Furthermore, for any $n \notin S$, $x_{n+1} \sim x_n$. Hence if $a = i_j$ and $b = i_{j+1}$, for any $a < n < b$, $x_{n+1} \sim x_n$. By transitivity of $\sim$, if $i_{j+1} \neq i_j{+}1$, then $x_{i_{j+1}} \sim x_{i_j+1}$. Otherwise, $x_{i_{j+1}} = x_{i_j+1}$ and so $x_{i_{j+1}} \sim x_{i_j+1}$.

Hence we have: for all j, $x_{i_{j+1}} \sim x_{i_j+1}\ R\ x_{i_j}$. As $\sim$ is compatible with R this forces $x_{i_{j+1}}\ R\ x_{i_j}$ for each j. This contradicts well-foundedness of R, as required. □

Property B.3 serves to prove the impossibility of certain infinite sequences, and is exactly what is required to apply the size-change termination criterion.

While we have written an arbitrary well-founded relation as R in the above to avoid confusion with well-founded orders, we shall adopt a more cavalier attitude both in the rest of this appendix and in the main text, and use more familiar notation such as $\prec$ and $<$ for such relations. The relation derived from a non-strict well-founded relation $(\prec, \sim)$ may be written $\precsim$.

B.2 Constructing Well-Founded Relations

We have in the above defined well-founded relations (both strict and non-strict) in as much generality as possible. It is tedious to check the conditions of Definitions B.1 and B.2 directly, however, and we now give ways of constructing appropriate well-founded relations.

The first result is unsurprising: any well-founded order is a well-founded relation:

Lemma B.4. *Suppose that $<$ is a well-founded partial order on A. Then $(<, =)$ is a non-strict well-founded relation on A.*

Beyond well-founded orders, a useful criterion is the following, using the machinery of ordinals to characterise well-founded relations:

Lemma B.5. *Let A be a set, and $\prec$ a relation on A. Then $\prec$ is a well-founded relation iff there is a map $f : A \to \mathcal{O}$, where $\mathcal{O}$ is the class of ordinals, such that $f\ x < f\ y$ whenever $x \prec y$.*

Proof. Suppose first that f is such a map. Then if $x_0 \succ x_1 \succ \cdots$, it follows that $f\ x_0 > f\ x_1 > \cdots$ (in the order on ordinals). As ordinals are well-ordered this is impossible, and thus infinitely decreasing chains in the $\prec$ order are impossible.

Conversely, suppose that $\succ$ is well-founded. Then we may define a map $f : A \to \mathcal{O}$ by (well-founded) recursion, as follows:

$$f\ x = \bigcup \{f\ y \mid y \prec x\} + 1$$

This is well defined, as $\neg(x \prec^+ x)$ since $\prec$ is well-founded. Furthermore, the union a of a set S of ordinals is likewise an ordinal, with $a \geq x$ for all $x \in S$. Hence $a + 1 > x$ for any $x \in S$. Thus $f\ x > f\ y$ for all $y \prec x$, as required. □

Such a map f is a *size function* for $\prec$, and we shall often write $|x|$ for $f\ x$. Note that it is necessary to take ordinals for the domain of f — natural numbers will not suffice. An example is given by the set $\{0, 1, 2, \ldots, \top\}$ with order $i \prec i + 1$ for each i, and $i \prec \top$ for all i. This is well-founded, but no finite size can be given for $\top$.

Bibliography

[Abe04] Andreas Abel. Termination checking with types. *Theoretical Informatics and Applications*, 38(4):277–319, 2004. Special Issue: Fixed Points in Computer Science (FICS '03).

[Abe06] Andreas Abel. A polymorphic lambda-calculus with sized higher-order types. PhD Thesis Draft, Ludwig-Maximilians-Universität München. Available online at `http://www.tcs.informatik.uni-muenchen.de/~abel/`, 2006.

[Ack77] Wilhelm Ackermann. On Hilbert's construction of the real numbers (1928). In Jean van Heijenoort, editor, *From Frege to Gödel: A Source Book in Mathematical Logic, 1879–1931*, Source Books in the History of the Sciences, pages 493–507. toExcel Press, 1977.

[AG00] Thomas Arts and Jürgen Giesl. Termination of term rewriting using dependency pairs. *Theoretical Computer Science*, 236(1–2):133–178, 2000.

[AK03] Hugh Anderson and Siau-Cheng Khoo. Affine-based size-change termination. In Atsushi Ohori, editor, *Proceedings of the First Asian Symposium on Programming Languages and Systems (APLAS '03)*, volume 2895 of *Lecture Notes in Computer Science*, pages 122–140. Springer, 2003.

[AKAL05] Hugh Anderson, Siau-Cheng Khoo, Stefan Andrei, and Beatrice Luca. Calculating polynomial runtime properties. In Kwangkeun Yi, editor, *Proceedings of the Third Asian Symposium on Programming Languages and Systems (APLAS '05)*, volume 3780 of *Lecture Notes in Computer Science*, pages 230–246. Springer, 2005.

[Art00] Thomas Arts. System description: The dependency pair method. In Leo Bachmair, editor, *Proceedings of the 11th International Conference on Rewriting Techniques and Applications (RTA '04)*, volume 1833 of *Lecture Notes in Computer Science*, pages 261–264. Springer, 2000.

[Ave06] James Avery. Size-change termination and bound analysis. In Masami Hagiya and Philip Wadler, editors, *Proceedings of the 8th International Symposium on Functional and Logic Programming (FLOPS '06)*, volume 3945 of *Lecture Notes in Computer Science*, pages 192–207. Springer, 2006.

[BA02] Amir M. Ben-Amram. General size-change termination and lexicographic descent. In Torben Mogensen, David Schmidt, and I. Hal Sudborough, editors, *The Essence of Computation: Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones*, volume 2566 of *Lecture Notes in Computer Science*, pages 3–17. Springer, 2002.

[BA06] Amir M. Ben-Amram. Size-change termination with difference constraints. Submitted for publication. Available online at `http://www2.mta.ac.il/~amirben/papers.html#dsct`, 2006.

[BAL06] Amir M. Ben-Amram and Chin Soon Lee. Size-change termination in polynomial time. To appear in ACM Transactions on Programming Languages and Systems (TOPLAS). Available online at `http://www2.mta.ac.il/~amirben/papers.html`. An earlier version can be found in [Lee02a], 2006.

[BCDO06] Josh Berdine, Byron Cook, Dino Distefano, and Peter W. O'Hearn. Automatic termination proofs for programs with shape-shifting heaps. In Thomas Ball and Robert Jones, editors, *Proceedings of the 18th Conference on Computer-Aided Verification (CAV '06)*, volume 4144 of *Lecture Notes in Computer Science*, pages 386–400. Springer, 2006.

[BHA86] Geoffrey L. Burn, Chris Hankin, and Samson Abramsky. Strictness analysis for higher-order functions. *Science of Computer Programming*, 7:249–278, 1986.

[Bir84] Richard S. Bird. Using circular programs to eliminate multiple traversals of data. *Acta Informatica*, 21(3):239–250, 1984.

[Bir98] Richard S. Bird. *Introduction to Functional Programming Using Haskell*. Prentice-Hall, second edition, 1998.

[BMMR01] Thomas Ball, Rupak Majumdar, Todd Millstein, and Sriram K. Rajamani. Automatic predicate abstraction of C programs. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '01)*, pages 203–213. ACM Press, 2001.

[BN98] Franz Baader and Tobias Nipkow. *Term Rewriting and all that*. Cambridge University Press, 1998.

[BS89] Alexander Brodsky and Yehoshua Sagiv. Inference of monotonicity constraints in Datalog programs. In *Proceedings of the Eighth ACM SIGACT-SIGART-SIGMOD Symposium on Principles of Database Systems (PoDS '89)*, pages 190–199. ACM Press, 1989. Journal version available [BS99].

[BS90] Alexander Brodsky and Yehoshua Sagiv. On termination of Datalog programs. In Won Kim, Jean-Marie Nicolas, and Shojiro Nishio, editors, *Proceedings of the First International Conference on Deductive and Object-Oriented Databases (DOOD '89)*, pages 95–112. North-Holland/Elsevier Science Publishers, 1990.

[BS99] Alexander Brodsky and Yehoshua Sagiv. Inference of monotonicity constraints in Datalog programs. *Annals of Mathematics and Artificial Intelligence*, 26:29–57, 1999.

[Bur87] Geoffrey L. Burn. Evaluation transformers — a model for the parallel evaluation of functional languages. In Gilles Kahn, editor, *Proceedings of a Conference on Functional Programming and Computer Architecture (FPCA '87)*, volume 274 of *Lecture Notes in Computer Science*, pages 446–470. Springer, 1987.

[CC77] 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 the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '77)*, pages 238–252. ACM Press, 1977.

[CC91] Patrick Cousot and Radhia Cousot. Relational abstract interpretation of higher-order functional programs. In Michel Billaud, Pierre Castran, Marc-Michel Corsini, Kandina Musumbu, and Antoine Rauzy, editors, *Actes JTASPEFL '91*, volume 74 of *Bigre*, 1991.

[CC94] 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 Computer Society 1994 International Conference on Computer Languages (ICLL '94)*, pages 95–112. IEEE Computer Society Press, 1994.

[CC95] Patrick Cousot and Radhia Cousot. Formal language, grammar and set-constraint-based program analysis by abstract interpretation. In *Proceedings of the Seventh ACM SIGPLAN-SIGARCH-WG2.8 Conference on Functional Programming Languages and Computer Architecture (FPCA '95)*, pages 170–181. ACM Press, 1995.

[CH78] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages (POPL '78)*, pages 84–97. ACM Press, 1978.

[Chu36] Alonzo Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58(2):345–363, April 1936.

[Chu40] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[CK00] Wei-Ngan Chin and Siau-Cheng Khoo. Calculating sized types. In *Proceedings of the 2000 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '00)*, volume 34 of *SIGPLAN Notices*, pages 62–72. ACM Press, 2000.

[CPR05] Byron Cook, Adreas Podelski, and Andrey Rybalchenko. Abstraction refinement for termination. In Chris Hankin and Igor Siveroni, editors, *Proceedings of the 12th International Static Analysis Symposium (SAS '05)*, volume 3672 of *Lecture Notes in Computer Science*, pages 87–101. Springer, 2005.

[CPR06a] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Termination proofs for systems code. In Michael Schwartzbach and Thomas Ball, editors, *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '06)*, pages 415–426. ACM Press, 2006.

[CPR06b] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Terminator: Beyond safety. In Thomas Ball and Robert Jones, editors, *Proceedings of the 18th Conference on Computer-Aided Verification (CAV '06)*, volume 4144 of *Lecture Notes in Computer Science*, pages 415–418. Springer, 2006.

[Dan02] Olivier Danvy. Lambda-lifting in quadratic time. In Zhenjiang Hu and Mario Rodríguez-Artalejo, editors, *Proceedings of the 6th Symposium on Functional and Logic Programming (FLOPS '02)*, volume 2441 of *Lecture Notes in Computer Science*, pages 134–151. Springer, 2002.

[Der87] Nachum Dershowitz. Termination of rewriting. *Journal of Symbolic Computation*, 3(1–2):69–116, 1987.

[DP02] Brian A. Davey and Hilary A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, second edition, 2002.

[FQ02] Cormac Flanagan and Shaz Qadeer. Predicate abstraction for software verification. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '02)*, pages 191–202. ACM Press, 2002.

[Fre01] 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.

[Fre02] Carl Christian Frederiksen. Automatic runtime analysis for first order functional programs. Master's thesis, DIKU, University of Copenhagen, September 2002. Available online at `http://www.diku.dk/topps/bibliography/2002.html`.

[Gan80] Robin O. Gandy. Proof of strong normalization. In J. Roger Hindley and Jonathan P. Seldin, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*. Academic Press, 1980.

[GH93] E. Goubault and C. L. Hankin. A lattice for the abstract interpretation of term graph rewriting systems. In M. R. Sleep, M. J. Plasmeijer, and M. C. J. D. van Eekelen, editors, *Term Graph Rewriting: Theory and Practice*, pages 131–140. John Wiley, 1993.

[Gir71] Jean-Yves Girard. Une extension de l'interprétation de Gödel à l'analyse, et son application à l'élimination des coupures dans l'analyse et la théorie des types. In J.E. Fenstad, editor, *Proceedings of the Second Scandinavian Logic Symposium*, pages 63–92. North-Holland, 1971.

[GJ05] Arne John Glenstrup and Neil D. Jones. Termination analysis and specialization-point insertion in offline partial evaluation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(6):1147–1215, November 2005.

[GLT89] Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and Types*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1989.

[GM93] Mike J. C. Gordon and Tom F. Melham, editors. *Introduction to HOL: A theorem proving environment for higher order logic*. Cambridge University Press, 1993.

[GP02] John P. Gallagher and Germán Puebla. Abstract interpretation over non-deterministic finite tree automata for set-based analysis of logic programs. In Shriram Krishnamurthi and C. R. Ramakrishnan, editors, *Proceedings of the 4th International Symposium on Practical Aspects of Declarative Languages (PADL '02)*, volume 2257 of *Lecture Notes in Computer Science*. Springer, 2002.

[GS97] Susanne Graf and Hassen Sadi. Construction of abstract state graphs with PVS. In Orna Grumberg, editor, *Proceedings of the 9th Conference on Computer-Aided Verification (CAV '97)*, volume 1254 of *Lecture Notes in Computer Science*, pages 72–83. Springer, 1997.

[GSSKT06] Jürgen Giesl, Stephan Swiderski, Peter Schneider-Kamp, and René Thiemann. Automated termination analysis for haskell: From term rewriting to programming languages. In Frank Pfenning, editor, *Proceedings of the 17th International Conference on Rewriting Techniques and Applications (RTA '06)*, volume 4098 of *Lecture Notes in Computer Science*, pages 297–312. Springer, 2006.

[GTSK05a] Jürgen Giesl, René Thiemann, and Peter Schneider-Kamp. The dependency pair framework: Combining techniques for automated termination proofs. In Franz Baader and Andrei Voronkov, editors, *Proceedings of the 11th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR '04)*, volume 3452 of *Lecture Notes in Artificial Intelligence*, pages 301–331. Springer, 2005.

[GTSK05b] Jürgen Giesl, René Thiemann, and Peter Schneider-Kamp. Proving and disproving termination of higher-order functions. In Bernhard Gramlich, editor, *Proceedings of the 5th International Workshop on Frontiers of Combining Systems (FroCoS '05)*, volume 3717 of *Lecture Notes in Artificial Intelligence*, pages 216–231. Springer, 2005.

[GTSKF03] Jürgen Giesl, René Thiemann, Peter Schneider-Kamp, and Stephan Falke. Improving dependency pairs. In Moshe Vardi and Andrei Voronkov, editors, *Proceedings of the 10th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR '03)*, volume 2850 of *Lecture Notes in Artificial Intelligence*, pages 165–179. Springer, 2003.

[GTSKF04] Jürgen Giesl, René Thiemann, Peter Schneider-Kamp, and Stephan Falke. Automated termination proofs with AProVE. In Vincent van Oostrom, editor, *Proceedings of the 15th International Conference on Rewriting Techniques and Applications (RTA '04)*, volume 3091 of *Lecture Notes in Computer Science*, pages 210–220. Springer, 2004.

[Hei94] Nevin Heintze. Set-based analysis of ML programs. In *Proceedings of the 1994 ACM conference on LISP and functional programming*, pages 306–317. ACM Press, 1994.

[Hei95] Nevin Heintze. Control-flow analysis and type systems. In Alan Mycroft, editor, *Proceedings of the Second International Symposium on Static Analysis (SAS '95)*, volume 983 of *Lecture Notes in Computer Science*, pages 189–206. Springer, 1995.

[HM03] Nao Hirokawa and Aart Middeldrop. Tsukuba termination tool. In Robert Nieuwenhuis, editor, *Proceedings of the 14th International Conference on Rewriting Techniques and Applications (RTA '03)*, volume 2706 of *Lecture Notes in Computer Science*, pages 311–320. Springer, 2003.

[JB04] Neil D. Jones and Nina Bohr. Termination analysis of the untyped λ-calculus. In Vincent van Oostrom, editor, *Proceedings of the 15th International Conference on Rewriting Techniques and Applications (RTA '04)*, volume 3091 of *Lecture Notes in Computer Science*. Springer, 2004.

[JG02] Neil D. Jones and Arne Glenstrup. Program generation, termination and binding-time analysis. In Don Batory, Charles Consel, and Walid Taha, editors, *Proceedings of the ACM SIGPLAN / SIGSOFT Conference on Generative*

Programming and Component Engineering (GPCE '02), volume 2487 of *Lecture Notes in Computer Science*, pages 1–31. Springer, 2002.

[Joh85] Thomas Johnsson. Lambda lifting: Transforming programs to recursive equations. In Jean-Pierre Jouannaud, editor, *Proceedings of a Conference on Functional Programming Languages and Computer Architecture (FPCA '85)*, volume 201 of *Lecture Notes in Computer Science*, pages 190–203. Springer, 1985.

[Joh87] Thomas Johnsson. Attribute grammars as a functional programming paradigm. In Gilles Kahn, editor, *Functional Programming Languages and Computer Architecture*, volume 274 of *Lecture Notes in Computer Science*, pages 154–173. Springer, 1987.

[Jon87] 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.

[JW95] Suresh Jagannathan and Stephen Weeks. A unified treatment of flow analysis in higher-order languages. In *Proceedings of 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '95)*, pages 393–407. ACM Press, 1995.

[KB70] Don E. Knuth and Peter B. Bendix. Simple word problems in universal algebras. In John Leech, editor, *Computational Problems in Abstract Algebras*, pages 263–197. Pergamon Press, 1970.

[KL80] Samuel Kamin and Jean-Jacques Levy. Two generalizations of the recursive path ordering. Unpublished Manuscript, University of Illinois. Available online at `http://perso.ens-lyon.fr/pierre.lescanne/not_accessible.html`, 1980.

[Knu68] Don E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, 1968.

[Knu71] Don E. Knuth. Semantics of context-free languages (correction). *Mathematical Systems Theory*, 5(1):95–96, 1971.

[Lan64] Peter J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308–320, January 1964.

[Lan79] Dallas Lankford. On proving term rewriting systems are noetherian. Technical Report MTP-3, Louisiana Technical University, Ruston, LA, USA, 1979.

[LDG+04] Xavier Leroy, Damien Doligez, Jacques Guarrigue, Didier Rémy, and Jérôme Vouillon. *The Objective Caml System*, 2004. Available at `http://caml.inria.fr/`.

[Lee01] Chin Soon Lee. *Program Termination Analysis and Termination of Offline Partial Evaluation*. PhD thesis, University of Western Australia, March 2001. Available online at `http://www.diku.dk/topps/bibliography/2002.html#D-475`.

[Lee02a] Chin Soon Lee. Finiteness analysis in polynomial time. In Don Batory, Charles Consel, and Walid Taha, editors, *Proceedings of the ACM SIGPLAN / SIGSOFT Conference on Generative Programming and Component Engineering (GPCE '02)*, volume 2487 of *Lecture Notes in Computer Science*, pages 218–235. Springer, 2002. Subsumed by [BAL06].

[Lee02b] Chin Soon Lee. Finiteness analysis in polynomial time. In Manuel Hermenegildo and German Puebla, editors, *Proceedings of the 9th International Symposium on Static Analysis (SAS '02)*, volume 2477 of *Lecture Notes in Computer Science*, pages 493–508. Springer, 2002.

[LJBA01] Chin Soon Lee, Neil D. Jones, and Amir M. Ben-Amram. The size-change principle for program termination. In Chris Hankin and Dave Schmidt, editors, *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '01)*, pages 81–92. ACM Press, 2001.

[LS96] Naomi Lindenstrauss and Yehoshua Sagiv. Checking termination of queries to logic programs. Available online at `http://www.cs.huji.ac.il/~naomil/`, 1996.

[LS97] Naomi Lindenstrauss and Yehoshua Sagiv. Automatic termination analysis of logic programs. In Lee Naish, editor, *Proceedings of the Fourteenth International Conference on Logic Programming (ICLP '97)*, pages 63–77. MIT Press, 1997.

[LSS97] Naomi Lindenstrauss, Yehoshua Sagiv, and Alexander Serebrenik. TermiLog: A system for checking termination of queries to logic programs. In Orna Grumberg, editor, *Proceedings of the 9th Conference on Computer-Aided Verification (CAV '97)*, volume 1254 of *Lecture Notes in Computer Science*, pages 444–447. Springer, 1997.

[MH98] Pasquale Malacaria and Chris Hankin. A new approach to control flow analysis. In Kai Koskimies, editor, *Proceedings of the 7th International Conference on Compiler Construction (CC '98)*, volume 1383 of *Lecture Notes in Computer Science*, pages 95–108. Springer, 1998.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, April 1978.

[MN70] Zohar Manna and Steven Ness. On the termination of Markov algorithms. In *Proceedings of the Third Hawaii International Conference on System Science*, pages 798–792, 1970. Available online at `http://perso.ens-lyon.fr/pierre.lescanne/not_accessible.html`.

[MTHM97] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, May 1997.

[MV06] Panagiotis Manolios and Daron Vroon. Termination analysis with calling context graphs. In Thomas Ball and Robert Jones, editors, *Proceedings of the 18th Conference on Computer-Aided Verification (CAV '06)*, volume 4144 of *Lecture Notes in Computer Science*, pages 401–414. Springer, 2006.

[Myc80] Alan Mycroft. The theory and practice of transforming call-by-need into call-by-value. In Bernard Robinet, editor, *International Symposium on Programming: Proceedings of the Fourth* Colloque International sur la Programmation, volume 83 of *Lecture Notes in Computer Science*, pages 269–281. Springer, 1980.

[NNH99] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.

[Nöc90] Eric Nöcker. Strictness analysis based on abstract reduction. In *Proceedings of the Second International Workshop on Implementation of Functional Languages on Parallel Architectures*, pages 297–321, 1990.

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

[Par00] Lars Pareto. *Types for Crash Prevention*. PhD thesis, Chalmers University of Technology, 2000. Available online at `http://www.cs.chalmers.se/~pareto/`.

[Pét69] Rózsa Péter. *Recursive Functions*, chapter 10. Academic Press, 1969.

[PJ87] Simon Peyton-Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, 1987. Available from `http://research.microsoft.com/users/simonpj/papers/slpj-book-1987/`.

[PJ03] Simon Peyton-Jones. *Haskell 98 language and libraries: the Revised Report*. Cambridge University Press, 2003. Online version at `http://haskell.org/definition`.

[Pla78a] David A. Plaisted. A recursively defined ordering for proving termination of term rewriting systems. Technical Report 78-943, University of Illinois at Urbana-Champaign, September 1978. Available online at `http://perso.ens-lyon.fr/pierre.lescanne/not_accessible.html`.

[Pla78b] David A. Plaisted. Well-founded orderings for proving termination of systems of rewrite rules. Technical Report 78-932, University of Illinois at Urbana-Champaign, July 1978. Available online at `http://perso.ens-lyon.fr/pierre.lescanne/not_accessible.html`.

[Plo75] Gordon Plotkin. Call-by-name, call-by-value and the lambda calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[Plü90a] Lutz Plümer. *Termination Proofs for Logic Programs*, volume 446 of *Lecture Notes in Artificial Intelligence*. Springer, 1990.

[Plü90b] Lutz Plümer. Termination proofs for logic programs based on predicate inequalities. In David H. D. Warren and Péter Szeredi, editors, *Proceedings of the 7th International Conference on Logic Programming (ICLP '90)*, pages 634–648. MIT Press, 1990.

[PR04a] Andreas Podelski and Andrey Rybalchenko. A complete method for the synthesis of linear ranking functions. In Bernhard Steffen and Giorgio Levi, editors, *Proceedings of the 5th International Conference on Verification, Model Checking and Abstract Interpretation*, volume 2937 of *Lecture Notes in Computer Science*, pages 239–251. Springer, 2004.

[PR04b] Andreas Podelski and Andrey Rybalchenko. Transition invariants. In *Proceedings of the 19th IEEE Symposium on Logic in Computer Science (LICS '04)*, pages 32–41. IEEE Computer Society, 2004.

[PR05] Andreas Podelski and Andrey Rybalchenko. Transition predicate abstraction and fair termination. In *Proceedings of the 32nd ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '05)*, pages 132–144. ACM Press, 2005.

[PS95] Jens Palsberg and Michael I. Schwartzbach. Safety analysis versus type inference. *Information and Computation*, 118(1):128–141, 1995.

[Rey72] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM Annual Conference*, volume 2, pages 717–740. ACM Press, 1972.

[Rey74] John C. Reynolds. Towards a theory of type structure. In *Paris Colloquium on Programming*, volume 19 of *Lecture Notes in Computer Science*, pages 408–425. Springer, 1974.

[Ros90] Kenneth A. Ross. Modular stratification and magic sets for Datalog programs with negation. In *Proceedings of the Ninth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PoDS '90)*, pages 161–171. ACM Press, 1990. Journal version available [Ros94a].

[Ros94a] Kenneth A. Ross. Modular stratification and magic sets for Datalog programs with negation. *Journal of the ACM*, 41(6):1216–1266, 1994.

[Ros94b] Kenneth A. Ross. A syntactic stratification condition using constraints. In Maurice Bruynooghe, editor, *Proceedings of the 1994 International Symposium on Logic programming (ISLP '94)*, pages 76–90. MIT Press, 1994.

[Sag91] Yehoshua Sagiv. A termination test for logic programs. In Vijay A. Saraswat and Kazunori Ueda, editors, *Proceedings of the 1991 International Symposium on Logic Programming (ISLP '91)*, pages 518–532. MIT Press, 1991.

[Sch76] Helmut Schwichtenberg. Definierbare funktionen im λ-kalkül mit typen. *Arch. Math. Logik*, 17:113–114, 1976.

[SG91] Kirack Sohn and Allen Van Gelder. Termination detection in logic programs using argument sizes. In *Proceedings of the Tenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PoDS '91)*, pages 216–226. ACM Press, 1991.

[She05] Tim Sheard. Putting Curry-Howard to work. In *Proceedings of the 2005 ACM SIGPLAN workshop on Haskell*, 2005. Available from the ACM Digital Library at `http://doi.acm.org/10.1145/1088348.1088356`.

[Shi88] Olin Shivers. Control-flow analysis in Scheme. In R. L. Wexelblat, editor, *Proceedings of the 2nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '88)*, pages 164–174. ACM Press, June 1988.

[Shi91] Olin Shivers. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, Carnegie Mellon University, May 1991.

[SJ05] Damien Sereni and Neil D. Jones. Termination analysis of higher-order functional programs. In Kwangkeun Yi, editor, *Proceedings of the Third Asian Symposium on Programming Languages and Systems (APLAS '05)*, volume 3780 of *Lecture Notes in Computer Science*, pages 281–297. Springer, 2005.

[Sli96] Konrad Slind. Function definition in higher-order logic. In Joakim von Wright, Jim Grundy, and John Harrison, editors, *Proceedings of the 9th International Conference on Theorem Proving in Higher-Order Logics (TPHOLs '96)*, volume 1125 of *Lecture Notes in Computer Science*, pages 381–397. Springer, 1996.

[Sli97] Konrad Slind. Derivation and use of induction schemes in higher-order logic. In Elsa L. Gunter and Amy P. Felty, editors, *Proceedings of the 10th International Conference on Theorem Proving in Higher-Order Logics (TPHOLs '97)*, volume 1275 of *Lecture Notes in Computer Science*, pages 275–290. Springer, 1997.

[Sli99] Konrad Slind. *Reasoning about Terminating Functional Programs.* PhD thesis, Technischen Universität München, 1999. Available online at `http://www.cl.cam.ac.uk/~kxs/papers/phd.html`.

[Tai67] William W. Tait. Intensional interpretation of functionals of finite type I. *Journal of Symbolic Logic*, 32:198–212, 1967.

[Ter03] Terese, editor. *Term Rewriting Systems.* Cambridge University Press, 2003.

[TG03] René Thiemann and Jürgen Giesl. Size-change termination for term rewriting. In Robert Nieuwenhuis, editor, *Proceedings of the 14th International Conference on Rewriting Techniques and Applications (RTA '03)*, volume 2706 of *Lecture Notes in Computer Science*, pages 264–278. Springer, 2003.

[TG05] René Thiemann and Jürgen Giesl. The size-change principle and dependency pairs for termination of term rewriting. *Applicable Algebra in Engineering, Communication and Computing*, 16(4):229–270, 2005.

[TGSK04] René Thiemann, Jürgen Giesl, and Peter Schneider-Kamp. Improved modular termination proofs using dependency pairs. In David Basin and Michael Rusinowitch, editors, *Proceedings of the 2nd International Joint Conference on Automated Reasoning (IJCAR '04)*, volume 3097 of *Lecture Notes in Artificial Intelligence*, pages 75–90. Springer, 2004.

[Tur36] Alan M. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, 2(42):230–265, November 1936.

[UG88] Jeffrey D. Ullman and Allen Van Gelder. Efficient tests for top-down termination of logical rules. *Journal of the ACM*, 35(2):345–373, 1988.

[vEGHN93] Marko van Eekelen, Eric Goubault, Chris Hankin, and Eric Nöcker. Abstract reduction: Towards a theory via abstract interpretation. In M. R. Sleep, M. J. Plasmeijer, and M. C. J. D. van Eekelen, editors, *Term Graph Rewriting: Theory and Practice*, pages 117–130. John Wiley, 1993.

[vGRS91] Allen van Gelder, Kenneth A. Ross, and John S. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM*, 38(3):620–650, July 1991.

[Wad87] Philip Wadler. Strictness analysis over non-flat domains (by abstract interpretation over finite domains). In Samson Abramsky and Chris Hankin, editors, *Abstract Interpretation.* Ellis Horwood, 1987.

[Wah00] David Wahlstedt. Detecting termination using size-change in parameter values. Master's thesis, Göteborgs Universitet, 2000. Available online at `http://www.cs.chalmers.se/~davidw/`.

[Wah04] David Wahlstedt. Type theory with first-order data types and size-change termination. Master's thesis, Chalmers University of Technology, 2004. Licenciate Thesis 36L. Available online at `http://www.cs.chalmers.se/~davidw/`.

[Wel99] J. B. Wells. Typability and type checking in system F are equivalent and undecidable. *Annals of Pure and Applied Logic*, 98(1–3):111–156, 1999.