Programming Research Group

SIZE-CHANGE TERMINATION OF HIGHER-ORDER FUNCTIONAL PROGRAMS

Damien Sereni

PRG-RR-04-20

Oxford University Computing Laboratory
Wolfson Building, Parks Road, Oxford OX1 3QD

Abstract

The *size-change termination* principle [18] is a simple criterion for program termination, originally described for first-order functional programs. A program is size-change terminating if *any infinite call sequence would result in an infinite decrease of a well-founded parameter*. This is a powerful and simple termination analysis, so that it identifies termination for a substantial and natural class of programs. In addition, its extensional power is the class of Péter's mutual recursive functions [1].

The ideas underlying the size-change termination analysis have been applied recently to develop a termination analysis for the pure untyped λ-calculus [12]. Again, this is a powerful method, proving termination of λ-expressions containing even the Y fixpoint combinator.

We bridge the gap between the two analyses by developing a termination analysis of a call-by-value, higher order functional language. This is an application of the ideas used in the λ-calculus analysis, which it subsumes. It is also a useful step in applying this analysis to real functional programs.

Contents

Notation

Some useful notation that we will use without comment in the sequel is given below:

Sequences A (finite) *sequence* of elements $x_1, \ldots, x_n$ will be written using the angled brackets notation:

$$xs = \langle x_1, \ldots, x_n \rangle$$

Note that we use the convention that xs denotes a sequence of elements x_i when there can be no confusion. The length of a sequence is written as follows:

$$|xs| = |\langle x_1, \ldots, x_n \rangle| = n$$

For concision, a sequence of elements generated in a uniform way will sometimes be written using the indexed notation, with an optional *guard*. The intended meaning is equivalent to ZF set comprehensions:

$$xs = \langle x_i \mid P(x_i) \rangle_{i=1}^{n}$$

The *empty sequence* is written ϵ. Furthermore, we write:

$$\begin{array}{rcl}
\langle x_1, \ldots, x_n \rangle +\!\!+ \langle y_1, \ldots, y_m \rangle & = & \langle x_1, \ldots, x_n, y_1, \ldots, y_m \rangle \\
x_1 :: \langle x_2, \ldots, x_n \rangle & = & \langle x_1, x_2, \ldots, x_n \rangle
\end{array}$$

Standard Functions We will use the standard functional programming functions *map*, *take* and *filter* on lists in programs, or on sequences in mathematics. These are defined as follows (in the sequence notation, informally):

$$\begin{array}{rcl}
\mathit{map}\ f\ \langle x_1, \ldots, x_n \rangle & = & \langle f\ x_1, \ldots, f\ x_n \rangle \\
\mathit{filter}\ p\ \langle x_1, \ldots, x_n \rangle & = & \langle x_i \mid p\ x_i = \mathbf{true} \rangle_{i=1}^{n} \\
\mathit{take}\ m\ \langle x_1, \ldots, x_n \rangle & = & \left\{ \begin{array}{ll} \langle x_1, \ldots, x_m \rangle & \text{if } m \leq n \\ \langle x_1, \ldots, x_n \rangle & \text{otherwise} \end{array} \right.
\end{array}$$

In words, *map* applies a function to each element of a sequence, *filter* picks out elements satisfying a given property, and *take* chops off a list at a given position.

1 Introduction

1.1 Aims

We describe a termination analysis for a call-by-value higher-order functional programming language. The language that we aim to analyse is a substantial fragment of purely functional ML. The analysis is fully automatic, in the sense that the programmer does not need to specify any information (such as lexicographic orders) in order to analyse the program.

Our termination analysis is based on the *size-change termination principle* [18, 6] (SCT). The SCT analysis was originally developed for first-order functional programs. More recently, a termination analysis based on the SCT principle was described for the pure untyped λ-calculus by Neil Jones and Nina Bohr [12].

We aim to use the ideas from both the first-order and λ-calculus analyses for analysing higher-order functional programs. Our presentation follows [12], although the present description should be self-contained.

This is joint work with my supervisor Oege de Moor, and Neil Jones (from DIKU, University of Copenhagen).

1.2 Size-Change Termination

1.2.1 Well-Founded Datatypes

We assume throughout that we are working over *well-founded datatypes*. That is, we assume that each datatype is partially ordered with no infinite decreasing chains. Well-founded datatypes include the natural numbers with their usual ordering, as well as any algebraic datatype with the structural ordering.

Note that an important datatype that is not well-founded is the type of integers. We will return to this point at the end of this section.

1.2.2 Size-Change Termination

The size-change termination principle is based on the observation that (with the above assumption of well-foundedness), there can be no infinite decreasing chain of data values appearing in any execution of the program. We can therefore prove termination of programs by showing that any infinite run would result in such an infinite decreasing chain. This leads to the following informal statement of the SCT principle:

> A (first-order) program P is *size-change terminating* if any infinite path following the control flow of P would cause an infinite descent in some well-founded value in P.

We can illustrate this with a (trivial) example, the following factorial function (on natural numbers):

$$\begin{aligned} \mathit{fac}\ n \quad &= \quad \textbf{if}\ n == 0\ \textbf{then}\ 1 \\ & \qquad\quad \textbf{else}\ n * \mathit{fac}\ (n-1) \end{aligned}$$

This program is immediately seen to be terminating. For, suppose for a contradiction that evaluation of $\mathit{fac}\ n$ does not terminate for some n. This must result in an infinite sequence of calls $\mathit{fac} \to \mathit{fac}$ by inspection. However, each such call decreases the natural number parameter x. This results in an infinite decreasing chain of values of x, a contradiction by well-foundedness.

1.2.3 Size-Change Graphs

The previous example was trivial in that it had only one function of one argument. As a more complex example, consider the Ackermann function, written again in a hypothetical simple functional programming language:

$$\begin{array}{rcl} ack\; m\; n & = & \textbf{if } m == 0 \textbf{ then } n+1 \\ & & \quad \textbf{else if } n == 0 \textbf{ then } {}^{1}ack\ (m-1)\ 1 \\ & & \qquad \textbf{else } {}^{2}ack\ (m-1)\ ({}^{3}ack\ m\ (n-1)) \end{array}$$

where we have labelled the three recursive calls. How can we show that *ack* is size-change terminating? Following [18], we can define for each recursive call a *size-change graph*. The graph for a call i from a function f to a function g is a bipartite graph with the parameters of f as input nodes and the parameters of g as output nodes. An arc from node x to node y indicates a size relationship between the parameter x in f and the value of y passed to g in call i. It is labelled with $=$ to indicate that the value of y is no greater than the value of x, and with $\downarrow$ to indicate that the value of y is smaller.

To make this concrete, we exhibit a possible set of size-change graphs for the Ackermann program above:

$$\begin{array}{ccc} 1 : ack \to ack & \qquad 2 : ack \to ack & \qquad 3 : ack \to ack \\ \begin{array}{ccc} m & \xrightarrow{\downarrow} & m \\ n & & n \\ & G_1 & \end{array} & \qquad \begin{array}{ccc} m & \xrightarrow{\downarrow} & m \\ n & & n \\ & G_2 & \end{array} & \qquad \begin{array}{ccc} m & \xrightarrow{=} & m \\ n & \xrightarrow{\downarrow} & n \\ & G_3 & \end{array} \end{array}$$

It is clear by inspection of the program that all the size change information in the above graphs is correct. We can now prove termination of *ack* easily.

Suppose for a contradiction that there is an infinite chain of calls in some run of the program. We consider the concatenation of the size-change graphs for each of the calls, and prove that there is an infinite path of decreasing arrows. Note that this is a contradiction, as this implies that there is an infinite decrease in either m or n.

Suppose first that there are infinitely many graphs labelled 1 or 2. Then there are infinitely many decreasing arrows $m \overset{\downarrow}{\to} m$. Furthermore, all other graphs are labelled 3 and so have an arrow $m \overset{=}{\to} m$. We therefore have an infinite path of arrows $m \to m$, with infinitely many of these arrows labelled $\downarrow$, a contradiction.

Otherwise, necessarily some tail of the sequence of labels is 3^ω. But graph 3 contains an arrow $n \overset{\downarrow}{\to} n$, and therefore we have found an infinite decreasing path in n, a contradiction again. This establishes termination.

1.2.4 The Algorithm

The size-change termination analysis consists of two essentially independent phases:

- Extract a set of size-change graphs from the program P such that all the size-change information in the graphs is guaranteed to be correct in any run of P.

- Show that in any infinite sequence of the above graphs that follows the control flow of P, there is an infinite path with infinitely many $\downarrow$ edges.

The second phase of the analysis is described in detail in [18], and we give a brief summary in Section 3. This algorithm is complete, in that given the control flow graph and the set of graphs, it can decide precisely whether or not every infinite sequence of graphs must contain an infinitely decreasing path. The unavoidable imprecision of the analysis therefore only arises from the first phase of the algorithm.

1.2.5 Expressive Power

The size-change termination analysis is a simple and fully automatic criterion for program termination. In spite of its simplicity, its expressive power is quite high, as is in fact its intrinsic complexity. The complexity of the SCT problem was established in [18], where the problem is shown to be complete for PSPACE.

The expressive power of the SCT analysis can be characterised in two ways. The *extensional* power of the analysis is the set of *functions* that it can prove terminating. This is a natural and substantial class of functions, namely Péter's *multiple recursive functions* [22, 1]. The *intensional* power of the analysis is the set of *programs* that it proves terminating. This is by nature harder to describe, but it seems that it includes many natural programs.

1.2.6 The Higher-Order Analysis

Recall that the SCT graph analysis is independent of the actual program analysis (other than the control flow graph to be extracted from the program). Our work in developing the SCT analysis for higher-order functional programs is therefore confined to extracting safe size-change graphs from a higher-order program. However, there are a number of issues to be addressed here:

- The (well-founded) size order on atomic values is as before, but what kind of well-founded order can be placed on function values?
- The control flow of a first-order functional program is trivial, but the control flow of a general higher-order program is much more complicated.
- The notion of function call needs to be modified to extend to higher-order programs.

These issues are discussed in [12] for the λ-calculus, and we provide an analogous treatment for higher-order functional programs.

1.3 Limitations

There are a number of issues related to size-change termination that we will not seek to address in these notes. However, we briefly mention them here for completeness.

1.3.1 Well-Foundedness

We make the assumption throughout that all datatypes are well-founded. As mentioned previously, the type of integers is an important datatype that does not satisfy this assumption. Consequently, we use natural numbers rather than integers throughout. However, it is possible to apply the size-change termination analysis to programs with integer values. It suffices to show that there exists a lower bound on the actual values that an integer variable can take (for example, if it can be shown that an integer variable is always nonnegative it really behaves like a natural number). This can be achieved automatically for many programs by existing analyses, and we will ignore this aspect here for simplicity.

1.3.2 Error Values

We should note that a size-change terminating program need not terminate *successfully*. For example, consider the following "program":

$$f\ x = f\ (\mathsf{pred}\ x)$$

where x is a natural number parameter. This *is* size-change terminating, as x decreases at each call $f \to f$. However, any evaluation of $f\ x$ for an actual value of x will eventually entail evaluation of $\mathsf{pred}\ 0$, which results in a run-time error. We do consider such an execution to be terminating, however, as it involves no infinite execution sequence (so that the SCT principle *is* valid).

Again, it would be easy to prove that this can never happen for certain programs by safety analysis. If this is combined with the SCT analysis, we can guarantee that a program terminates without run-time errors. However, we will not concern ourselves with the details of such analyses here.

1.4 Outline

A brief outline of our development of the size-change termination analysis for higher-order functional programs is as follows:

- We define the language that we analyse and give it an operational semantics in Section 2.
- We give the notion of size change that we use in Section 3.
- We define an instrumentation of the operational semantics to produce size-change graphs (to record size-change information) in Section 4.
- We introduce an abstract interpretation of the instrumented semantics to produce a safe approximation to the set of possible size-change graphs in Section 5.
- We show how the simple language defined in Section 2 can be used to encode a substantial subset of pure ML in Section 6, thus establishing that our analysis can be used on a nontrivial existing language.
- Finally, we give some example of size-change terminating programs in Section 7 and conclude in Section 8.

Finally, note that most of the proofs that arise in these notes are straightforward but long. Many proofs are therefore not included in the main text but are given in Appendix A for completeness.

2 The Language

2.1 Programs

We begin by defining the syntax and semantics of the language that we will base the analysis on (though it could of course be extended to many similar languages). As previously mentioned, this language is a simple call-by-value purely functional language. A program is given in *named combinator form*, that is as a list of combinator definitions. A combinator definition is a simple function definition of the form

$$f\ x_1\ x_2\ \ldots\ x_n = e$$

for some identifiers $x_1, \ldots, x_n$ and some expression e. In particular, there is no pattern matching in the definition.

Each program ends with a single expression to be evaluated in the context of the preceding combinator definitions. The language contains constants, which may be either values of primitive datatypes or primitive operations on these values.

2.2 Syntax

The abstract and concrete syntax of the language is given by the following ML-like declaration:

$$
\begin{array}{rcll}
\textit{Program} & = & \textit{Combinator list} \times \textit{Expr} & c_1\ c_2 \cdots c_n; e \\
\textit{Combinator} & = & \textit{Name} \times \textit{Name list} \times \textit{Expr} & f\ x_1 \cdots x_n = e \\
\textit{Expr} & = & \ \ \textit{Name} & x \\
 & & \mid \textit{FunctionName} & f \\
 & & \mid \textit{Const}\ \mathbf{of}\ \textit{Constant} & c \\
 & & \mid \textit{App}\ \mathbf{of}\ \textit{Expr} \times \textit{Expr} & e_1\ e_2 \\
 & & \mid \textit{If}\ \mathbf{of}\ \textit{Expr} \times \textit{Expr} \times \textit{Expr} & \mathbf{if}\ e\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2
\end{array}
$$

Note that the expression e appearing in rule *Program* must be closed. Furthermore, any variable x appearing in the body of a combinator definition must be one of the parameters of f. Finally, a function name f is necessarily the name of some combinator in the program.

We assume that each expression is labelled with a *unique* identifier, so that different occurrences of the same expression are distinct. Thus an expression is really a *pair* (e, l) where e is the syntactic form of the expression, and l is the label. If we need to emphasise the dependence on the label l, we will write this as e^l, but when no confusion can arise we will write simply e.

Furthermore, this implies that every subexpression of the original program lies in precisely one combinator definition. We let $\textit{CombDef}(e)$ be the name of the combinator in which e appears, and $\sharp e$ the number of parameters of $\textit{CombDef}(e)$. Also, we write $e^{(f)}$ for the body (right-hand side) of f.

Note also that we will write **if** e **then** e_1 **else** e_2 as $(e \to e_1 \mid e_2)$ in the semantics to save space.

A few general notes about the syntax:

- We make a *syntactic* distinction between a function name f and a variable name x. This is simply done to simplify the description of the semantics later.

- Prominently missing are: λ-abstractions, **case** statements and pattern matching. This is done primarily to simplify the presentation, and does not affect expressiveness much — lambda lifting can convert abstractions to combinator definitions, **case** statements can be translated into chained **if** statements using the primitive tests on datatypes, and finally pattern matching can be converted into **case** statements. We will return to this in Section 6.

2.3 Constants

We now describe the set *Constant* of constants that is used by our analysis. We wish to include data values that contain higher-order values (such as a list of functions) and thus the set of constants is necessarily defined by mutual recursion with the set of all values (constant and higher-order), introduced later.

We do not wish to fix a specific set of constants, so that the analysis can be applied to different contexts. However, we must of course rely on certain properties. The crucial property for size-change termination is:

The set *Constant* has a well-founded order $<$.

2.3.1 Kinds of Constants

The set *Constant* can be partitioned into three different kinds of constants:

- *Primitive Operators*: constants that represent functions. Examples are succ (the successor function on $\mathbb{N}$) and cons (the constructor on lists).
- *Atomic Constants*: constants such as natural numbers and booleans which are assumed to be primitive. These cannot contain higher-order values.
- *Algebraic Constants*: constants built using constructors introduced in algebraic datatype declarations. These can contain functions.

We write

$$Constant = PrimOp \cup Atom \cup AlgConst$$

where the union is assumed disjoint. In addition, we partition the set *PrimOp* into operations on atoms and operations on algebraic constants:

$$PrimOp = AtomOp \cup AlgOp$$

Finally, we assume defined a function $\sharp : Constant \to \mathbb{N}$, such that $\sharp c$ is the *arity* of c for all $c \in Constant$: for c a primitive operator this is the number of arguments that c expects, and $c \notin PrimOp$, $\sharp c = 0$.

Each primitive operator $op \in PrimOp$, with $\sharp op = n$, is identified with a (total) function

$$op : Value^n \to Value_{Err}$$

where *Value* is the set of (constant and higher-order) values, defined later; and $Value_{Err} = Value \cup \{Err\}$, where *Err* is a special values assumed distinct from every other value.

The *Err* value is used to represent partial operations. For example, the operations $0 - 1$ (on natural numbers) and $\mathsf{null}\ 1$ (where null is the test for the empty list) are undefined. However, it is inconvenient in the sequel to deal with partial functions, whence we define both of these to equal *Err*.

2.3.2 Atomic Constants

We do not specify the set of atomic constants exactly, so that this may be chosen as apropriate for the language to analyse. However, we must assume two properties of the set of atomic constants:

- *Atom* has a well-founded order $<_A$
- Function values cannot be stored in atomic constants. The following condition ensures that this holds:

  $$(\forall op \in AtomOp)(\forall vs \in Value^{\sharp op}) \quad op(vs) \in Atom_{Err}$$

  Strictly speaking, the above states that function values can never be retrieved from atomic constants, but the two are equivalent in our context.

- $AtomOp$ is finite.

Appendix B.2 gives one possible choice for this set, satisfying the above requirements. Again, the axioms above are all we will use in the sequel.

2.3.3 Algebraic Constants

We now describe the set of *algebraic constants*. Unlike the atomic constant case, we will define this set precisely. For, we wish to allow algebraic constants containing function values, and this can be a source of nonterminating behaviour. We must therefore have enough information about such constants to integrate them into our analysis.

The set of algebraic constants is the set of values than can be defined in an ML-like datatype declaration. This has the form:

$$\begin{array}{rcl} \textbf{type}\ (\alpha_1, \ldots, \alpha_k)\ t & = & C_1\ \textbf{of}\ t^1_1 \times t^1_2 \times \cdots \times t^1_{k_1} \\ & | & \ldots \\ & | & C_n\ \textbf{of}\ t^n_1 \times t^n_2 \times \cdots \times t^n_{k_n} \end{array}$$

where the C_i are *constructors*, and the α_l are *type variables* on which the types t^j_i can depend. Recall however that our core language is untyped. We therefore regard such a declaration as merely introducing the constructors $C_1, \ldots, C_n$ and their arities ($\sharp C_i = k_i$). This causes no loss of generality.

An example of such a datatype declaration is the type of lists:

$$\textbf{type}\ \alpha\ \mathit{list} = \mathit{Nil} \mid \mathit{Cons}\ \textbf{of}\ (\alpha \times \alpha\ \mathit{list})$$

The set of values defined by the declaration of type t above is:

$$\mathit{AlgConst}_t = \bigcup_{i=1}^{n} \{t.C_i(v_1, \ldots, v_{k_i}) \mid (\forall j \leq k_i)\ v_j \in \mathit{Value}\}$$

This defines a value of type t as a finite *tree* with nodes labelled with constructors. This relies on the fact that our language uses call-by-value and does not allow mutable data structures, making it impossible to build graph structures, and in particular cyclic data values.

Given a set *Types* of datatype declarations, the set of algebraic constants is then:

$$\mathit{AlgConst}_{\mathit{Types}} = \bigcup_{t \in \mathit{Types}} \mathit{AlgConst}_t$$

Note that the constructors appearing in distinct datatype declarations are assumed distinct (and are labelled with the type declaration in which they appear).

We now define a well-founded order on $\mathit{AlgConst}_{\mathit{Types}}$. This is entirely as expected, using the subtree order. Formally,

- $x \leq y$ iff $x < y$ or $x = y$, and
- $x < t.C_i(v_1, \ldots, v_n)$ iff for some v_j, $v_j \in \mathit{AlgConst}$ and $x \leq v_j$.

This order is trivially well-founded, since data values are finite trees. The usefulness of this order is that *destructors* (such as tail on lists) always return a value that is less than their argument in the above order. For the datatype of natural numbers,

$$\textbf{type}\ \mathit{Nat} = \mathit{Zero} \mid \mathit{Succ}\ \textbf{of}\ \mathit{Nat}$$

this is just the usual order on $\mathbb{N}$.

We have defined the set of algebraic constants and its order, given a set *Types* of datatype declarations. We now define the primitive operations on these values. There are three kinds of primitive operations: constructors, destructors and tests. We fix a type t in what follows:

1. *Constructors.* The constructor functions $\mathsf{C}_1, \ldots, \mathsf{C}_n$ are simply defined by $\sharp\mathsf{C}_i = k_i$ and

$$\mathsf{C}_i : (v_1, \ldots, v_{k_i}) \mapsto C_i(v_1, \ldots, v_{k_i})$$

   for all values v_j.

2. *Destructors.* The destructor functions of type t extract a particular component of a value. We write these as $\mathsf{getC_i.j}$. Then $\sharp\mathsf{getC_i.j} = 1$, and

$$\mathsf{getC_i.j}(x) = \begin{cases} c_j & \text{if } x = C_i\ (c_1, \ldots, c_{k_i}) \\ Err & \text{otherwise} \end{cases}$$

   Examples in the case of *list* are head and tail. Note that if $y := \mathsf{GetC_i.j}(x) \neq Err$, then $y < x$ in the subexpression order introduced above. Destructors therefore always decrease the size of their argument in non-error cases.

3. *Tests.* The test functions of type t allow guards based on the constructor of a value. We write these as $\mathsf{isC_i}$ for $1 \leq i \leq n$. Then $\sharp\mathsf{isC_i} = 1$, and

$$\mathsf{isC_i}(x) = \begin{cases} T & \text{if } x = C_i\ (c_1, \ldots, c_{k_i}) \\ F & \text{if } x = C_j\ (c_1, \ldots, c_{k_j}), j \neq i \\ Err & \text{otherwise} \end{cases}$$

   An example in the case of *list* is null.

Note that these are enough to compile any pattern-matching expression into **if** statements (provided that we identify matching failure with a selection operator returning *Err*). There is hence no loss of generality in excluding pattern matching and considering only these operators.

2.3.4 Putting it all together

We have specified the sets $Atom$ and $AtomOp$ of atomic constants and operators on atomic constants, and defined the sets $AlgConst_{Types}$ and $AlgOp_{Types}$ of algebraic constants and operations, given a set *Types* of datatype declarations. In addition, we have given well-founded orders $<_A$ on $Atom$ and $<$ on $AlgConst_{Types}$.

It is entirely straightforward to put these together to define the set *Constant*. It should simply be noted that this depends on the program to be analysed, as the set *Types* depends on the program (there are infinitely many possible datatype declarations, but only finitely many appear in the program). Also, note that there are only finitely many primitive operators in *PrimOp*.

Finally, the order $<$ on *Constant* is easily defined given the orders on *AlgConst* and *Atom*: it is simply the union of these orders, together with the trivial order on *PrimOp* (there is no need to order primitive operators, and no obvious way to do it). This is well-founded, as required.

2.4 Values and States

We will soon proceed to give an operational semantics for the core language described previously. In order to do this, we must define the set of *states* that can occur in the evaluation of a program, and the set of *values* that a state can evaluate to.

Before we can define these, we must state a property that will hold of the operational semantics: all the expressions that arise in the evaluation of the program are *subexpressions* of the original program P (although constants can also arise in the evaluation of P). We will prove that this actually holds. In the meantime, we note that this implies that each expression e has an associated arity $\sharp e$, as defined previously (the arity of the unique combinator definition in which e appears).

States and values are defined by mutual recursion:

$$\begin{aligned} State &= \Big\{(lhs, vs) \mid lhs \in Expr \cup Constant \wedge vs \in Value^* \wedge |vs| \leq \sharp lhs\Big\} \\ Value &= \Big\{(lhs, vs) \in State \mid |vs| < \sharp lhs \vee (lhs \in Constant \wedge \sharp lhs = 0)\Big\} \end{aligned}$$

A state is a thus a pair (lhs, vs) where lhs is either an expression or a constant, and vs is a list of values. The intention is that vs is a representation of the environment. Variables are therefore identified by their position in the list of parameters of a combinator definition. This makes it slightly easier to state the semantics later. Also, it allows for a uniform treatment of primitive operators — there are no variable *names* associated with an operator such as $+$, but there are two variable positions.

A state of the form (e, vs), where $|vs| < \sharp e$ corresponds to a *closure* (a function value). Under call-by-value, closures are never reduced and hence are included in the set *Value*. We also include partially evaluated primitive operators as closures. We let *Closure* denote the set of closures:

$$Closure = \{(lhs, vs) \in Value \mid |vs| < \sharp lhs\}$$

In the sequel, we will write $lhs : vs$ for the pair (lhs, vs) for clarity. We will also occasionally identify the value $c : \epsilon$ with the constant c when no confusion can arise.

2.5 Operational Semantics

Given the previous definitions, the (big-step) operational semantics of the language are now straightforward (Figure 1).

Note. In *all* the rules that follow apart from the Value rule, we assume an implicit side condition that the state to be reduced is not an incomplete closure. That is, if the state is $lhs : vs$, the side condition is

$$\sharp lhs = |vs|$$

This ensures that the only rule that applies to an incomplete closure is Value, and thus that closures only reduce to themselves.

Note also that a rule applying to (say) an application state $e_1 e_2 : vs$ really means "any state of the form $(e_1 e_2)^l : vs$ for some expressions e_1, e_2, some label l and some values vs (recall that each expression has a unique label identifying it). An illustration of these semantic rules in action is given in Appendix B.3.

We will shortly define two more versions of the operational semantics, to add more features that we will need later. These will be *compatible* with the semantics in Figure 1, in the sense that programs evaluate to the same results under all semantics. More precisely, we assert the following properties of all semantics:

- Determinism: If $s \Downarrow v$ and $s \Downarrow v'$, then $v = v'$. Also, $v \in Value$
- Compatibility: If $s \Downarrow v$ under the semantics of Figure 1, then $s \Downarrow v$ under all other semantics.

Value

$$\frac{}{s \Downarrow s}\ {\scriptstyle s \in \mathit{Value}}$$

Variable

$$\frac{}{x_i : vs \Downarrow vs_i}$$

Function

$$\frac{e^{(f)} : \epsilon \Downarrow v}{f : vs \Downarrow v}$$

Constant

$$\frac{}{\mathit{Cst}\ c : vs \Downarrow c : \epsilon}\ {\scriptstyle c \in \mathit{Constant}}$$

PrimOp

$$\frac{}{op : \langle v_1, \dots, v_n \rangle \Downarrow op(v_1, \dots, v_n)}\ {\scriptstyle n = \sharp op}$$

App

$$\frac{e_1 : vs \Downarrow e_b : ws \qquad e_2 : vs \Downarrow v \qquad e_b : (ws +\!\!+ \langle v \rangle) \Downarrow w}{e_1 e_2 : vs \Downarrow w}\ {\scriptstyle |ws| < \sharp e_b}$$

If-T

$$\frac{e \Downarrow T : \epsilon \qquad e_1 : vs \Downarrow v}{(e \to e_1 \mid e_2) : vs \Downarrow v}$$

If-F

$$\frac{e \Downarrow F : \epsilon \qquad e_2 : vs \Downarrow v}{(e \to e_1 \mid e_2) : vs \Downarrow v}$$

Figure 1: Operational Semantics

The proofs of the above are trivial and unilluminating; they are therefore omitted. Note that the second condition (compatibility) is stated as an implication rather than an equivalence as the next semantics will in fact add reductions.

2.5.1 Error Productions

The above semantics is intended to be clear, but it suffers from a major drawback — if an expression e does not reduce successfully, this is only characterised as the *absence* of a proof tree for $e \Downarrow v$. There is no explicit characterisation of nontermination. Furthermore, we wish to distinguish nontermination from abnormal termination (for example, the state $(1 \to e_1 \mid e_2) : vs$ does not reduce to anything with the above semantics, as the evaluation gets stuck). We remedy the latter problem first.

To avoid the problem of stuck computations, we update the semantics to produce the value *Err* previously introduced whenever an illogical operation is performed. The updated semantics producing the value *Err* to indicate abnormal termination are given in Figure 2.

As mentionned previously, this semantics is deterministic and compatible with the original semantics. The role of the *Err* value is shown by the following result (proof omitted):

Lemma 1. $s \Downarrow v$ (semantics in Fig 1) iff $s \Downarrow v$ (Fig 2) and $v \neq \mathit{Err}$. Also, if $s \Downarrow \mathit{Err}$ (Fig 2), then $s \not\Downarrow$ (Fig 1).

2.6 Extending the Operational Semantics

We now seek to extend the operational semantics given in the previous section to represent nontermination explicitly. Specifically, we introduce a "calls" relation $\to$ on states. The desired

Value

$$\frac{}{s \Downarrow s}\; {\scriptstyle s \in Value_{Err}}$$

Variable

$$\frac{}{x_i : vs \Downarrow vs_i}$$

Function

$$\frac{e^{(f)} : \epsilon \Downarrow v}{f : vs \Downarrow v}$$

Constant

$$\frac{}{Cst\ c : vs \Downarrow c : \epsilon}$$

PrimOp

$$\frac{}{op : vs \Downarrow op(vs)}\; {\scriptstyle |vs| = \sharp op}$$

AppErr1

$$\frac{e_1 : vs \Downarrow v}{e_1 e_2 : vs \Downarrow Err}\; {\scriptstyle v \notin Closure}$$

AppErr2

$$\frac{e_1 : vs \Downarrow s \qquad e_2 : vs \Downarrow Err}{e_1 e_2 : vs \Downarrow Err}$$

App

$$\frac{e_1 : vs \Downarrow e_b : ws \qquad e_2 : vs \Downarrow v \qquad e_b : (ws +\!\!+ \langle v \rangle) \Downarrow w}{e_1 e_2 : vs \Downarrow w}\; {\scriptstyle |ws| < \sharp e_b}$$

If-T

$$\frac{e \Downarrow T : \epsilon \qquad e_1 : vs \Downarrow v}{(e \to e_1 \mid e_2) : vs \Downarrow v}$$

If-F

$$\frac{e \Downarrow F : \epsilon \qquad e_2 : vs \Downarrow v}{(e \to e_1 \mid e_2) : vs \Downarrow v}$$

If-Err

$$\frac{e \Downarrow v}{(e \to e_1 \mid e_2) : vs \Downarrow Err}\; {\scriptstyle e \notin \{T:\epsilon, F:\epsilon\}}$$

Figure 2: Operational Semantics, with Error Rules

property of $\to$ is that $s \to s'$ whenever s' appears as a subgoal in the proof tree of $s \Downarrow$. The upshot of this is the following, dubbed the *nontermination is sequential* property after [12]:

Lemma 2 (NIS). $s \not\Downarrow$ iff there exists an infinite call sequence

$$s = s_0 \to s_1 \to s_2 \to \cdots$$

As the name reflects, this property allows nontermination to be characterised by the existence of one (linear) infinite sequence. Note that the NIS property allows termination to be proved by showing that such a sequence is impossible.

The operational semantics with calls is defined in Figure 3. Note that it is useful to split the call relation $\to$ into several cases. We have:

Definition 1. The call relation $\to$ is defined to be $\rightarrowtail \cup \Rightarrow$.

The $\Rightarrow$ relation is intended to represent "proper" calls, representing one step in the reduction of a state. By contrast, the $\rightarrowtail$ relation represents calls to evaluate subexpressions.

Again, this semantics is deterministic and compatible with the previously defined semantics.

We write $\to^*$ for the reflexive transitive closure of $\to$. Also, given a program $P = (cs, e)$, we define the state P_{start} to be the state $e : \epsilon$ (the *start state* of P).

2.7 Control States

The $\to$ relation gives a satisfactory notion of call, by the NIS property (Lemma 2). However, we still need to identify a finite number of *control states* that can occur during the evaluation of an expression. This guarantees that, in any nonterminating sequence of calls, at least one control state will occur infinitely often. We can therefore focus on *self-loops* $pp \to^* pp$ (for some program point pp) where meaningful size-change information can be obtained.

There is of course a trivial solution to this problem, namely defining one (vacuous) control state and regarding all calls as being from and to this state. However, the precision of the analysis is increased if the control flow can be constrained in some way.

The crucial observation is that the *expressions* that arise during evaluation of a program are in fact all subexpressions of this program, whereas the *environments* actually contain values that can depend on the computation. We assume that we are working with a fixed program P in the sequel, as the call and evaluation relations implicitly depend on P anyway.

Definition 2. The set of subexpressions of an expression e, including e itself, is denoted by *subexps*(e). The set of subexpression of a program P is

$$\mathit{subexps}(P) = \mathit{subexps}(e) \cup \bigcup_{i=1}^{k} \mathit{subexps}(e^{(f_i)})$$

where $P = (\langle f_1, \ldots, f_k \rangle, e)$.

We now define a function that takes a state to the collection of all expressions that appear in that state:

Definition 3. Let $s = \mathit{lhs} : \mathit{vs}$ be a state. We define the *expression support* of s to be:

$$\mathit{expsup}(\mathit{lhs} : \mathit{vs}) = \mathit{expsup}_{lhs}(\mathit{lhs}) \cup \bigcup_{i=1}^{n} \mathit{expsup}(v_i) \text{ where } \mathit{vs} = \langle v_1, \ldots, v_n \rangle$$

Value

$$\frac{}{s \Downarrow s}\ {\scriptstyle s \in Value_{Err}}$$

Variable

$$\frac{}{x_i : vs \Downarrow vs_i}$$

Function

$$\frac{}{f : vs \Rightarrow e^{(f)} : \epsilon}$$

Constant

$$\frac{}{Cst\ c : vs \Downarrow c : \epsilon}\ {\scriptstyle c \in Constant}$$

PrimOp

$$\frac{}{op : \langle v_1, \ldots, v_n \rangle \Downarrow op(v_1, \ldots, v_n)}\ {\scriptstyle n = \sharp op}$$

AppErr1

$$\frac{e_1 : vs \Downarrow v}{e_1 e_2 : vs \Downarrow Err}\ {\scriptstyle v \notin Closure}$$

AppErr2

$$\frac{e_1 : vs \Downarrow s \qquad e_2 : vs \Downarrow Err}{e_1 e_2 : vs \Downarrow Err}$$

CApp-1

$$\frac{}{e_1 e_2 : vs \rightarrowtail e_1 : vs}$$

CApp-2

$$\frac{e_1 : vs \Downarrow e_b : ws}{e_1 e_2 : vs \rightarrowtail e_2 : vs}\ {\scriptstyle |ws| < \sharp e_b}$$

CApp-Call

$$\frac{e_1 : vs \Downarrow e_b : ws \qquad e_2 : vs \Downarrow v}{e_1 e_2 : vs \Rightarrow e_b : ws \mathbin{+\!\!+} \langle v \rangle}\ {\scriptstyle |ws| < \sharp e_b}$$

CIf-1

$$\frac{}{(e \to e_1 \mid e_2) : vs \rightarrowtail e : vs}$$

If-Err

$$\frac{e \Downarrow v}{(e \to e_1 \mid e_2) : vs \Downarrow Err}\ {\scriptstyle e \notin \{T:\epsilon, F:\epsilon\}}$$

CIf-T

$$\frac{e : vs \Downarrow T : \epsilon}{(e \to e_1 \mid e_2) : vs \Rightarrow e_1 : vs}$$

CIf-F

$$\frac{e : vs \Downarrow F : \epsilon}{(e \to e_1 \mid e_2) : vs \Rightarrow e_2 : vs}$$

Step

$$\frac{s \Rightarrow s' \qquad s' \Downarrow v}{s \Downarrow v}$$

Figure 3: Operational Semantics: Error Rules and Calls

where $expsup_{lhs}$ is defined as follows:

$$\begin{array}{rcl} expsup_{lhs}(e) & = & subexps(e) \text{ if } e \text{ is an expression} \\ expsup_{lhs}(c) & = & \emptyset \text{ if } c \in Atom \\ expsup_{lhs}(C_i(v_1, \ldots, v_n)) & = & \bigcup_{i=1}^{n} expsup(v_i) \end{array}$$

Definition 4. Let $s = e : vs$ be a state. We define a predicate $SUB(s)$ by:

$$SUB(s) \quad \equiv \quad s = Err \vee expsup(s) \subseteq subexps(P)$$

Lemma 3. Suppose that $s \rightarrow s'$ or $s \Downarrow s'$. Then $SUB(s) \Rightarrow SUB(s')$.

Corollary 1 (The Subexpression Property). If $P_{start} \rightarrow^* e : vs$, then $e \in subexps(P)$ or $e \in Constant$.

Proof. As $P_{start} \rightarrow^* e : vs$, either $e : vs = P_{start}$ or there exists a sequence

$$P = s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow \cdots \rightarrow s_n = e : vs$$

Now $P_{start} = e_P : \epsilon$ by definition, so $expsup(P) = subexps(e_P) \subseteq subexps(P)$. Hence $SUB(s_0)$ holds. If $e : vs = P_{start}$ then $SUB(e : vs)$ holds. Otherwise, by induction, as $SUB(s_i) \Rightarrow SUB(s_{i+1})$ for each i, $SUB(s_n)$ holds, again as required.

Hence $expsup(e : vs) \subseteq subexps(P)$. If $e \in Constant$ then the result is vacuous. Otherwise, $e \in subexps(e) \subseteq expsup(e : vs) \subseteq subexps(P)$, as required. □

Certainly $subexps(P)$ is finite. We can therefore define the control states to be the set of subexpressions of P. Note that we need not include constants or Err as control states. Recall that we are concerned with identifying infinite sequences of calls of the form

$$P = s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow \cdots$$

In any such sequence, each s_i must be a state of the form $e : vs$ for some expression e. States of the form Err or $c : vs$ (for some constant c) are *terminal* (they do not call any further states) by the semantics. As no state is terminal in an infinite sequence of calls, such states cannot occur. The relevant control states are therefore only subexpressions of P.

3 Size Changes

In the previous section, the operational semantics of the language has been defined, including a *call* relation $\rightarrow$. We have shown that whenever a program is nonterminating, there is an infinite call sequence. In order to apply the size-change termination principle to show such infinite sequences impossible, we must define a well-founded order on states. This uses the order $<$ on constants (recall that this is well-founded), but also includes another notion of size change that extends the idea of size-change termination to higher-order functions.

3.1 An Ordering on States

We recall that a state s is a pair $lhs : vs$, where lhs is either an expression *or* a constant. Furthermore, each value v_i in vs is itself a state (as $Value \subseteq State$). This gives rise to a finite *tree* of states with root s. We can therefore regard a state s' as being "smaller" than s if it appears as a subtree of s. This is the crucial notion of size for higher-order values in our analysis.

Another notion of size is obtained by comparing the lhs components in the above. Consider two states $e : vs$, $e' : vs$ (where e, e' are expressions) with identical environments vs. If e' is a subexpression of e, we can regard the state $e' : vs$ as smaller than $e : vs$. Similarly, the same applies to states $c : vs$, $c' : vs$ (where c, c' are constants) when c' is less than c in the order on constants.

We will now make precise the informal description of the size order on states, and show that it is well-founded. Its usefulness will, we hope, become apparent when the analysis is further described and examples are shown.

The following definition defines the set of substates of a state (*ie* the set of its subtrees):

Definition 5. The *support* of a state s is defined to be:

$$\begin{aligned} support(Err) &= \emptyset \\ support(e : \langle v_1, \ldots, v_n \rangle) &= \{e : vs\} \cup \bigcup_{i=1}^{n} support(v_i) \end{aligned}$$

Note that this is closely related to $expsup(s)$, with the difference that $support(s)$ is a set of *states*, rather than a set of expressions.

We therefore have three distinct notions of size change between states:

1. Atomic Constants: $c_2 : vs \preccurlyeq_1 c_1 : vs$ whenever $c_2 \leq c_1$ (the $\leq$ order on constants was introduced in Section 2 and is well-founded)
2. Subtree relationship: $s_2 \preccurlyeq_2 s_1$ if $s_2 \in support(s_1)$
3. Subexpression relationship $s_2 \preccurlyeq_3 s_1$ if $s_1 = e_1 : vs$, $s_2 = e_2 : vs$ and $e_2 \in subexps(e_1)$.

Finally, there is another case to deal with in which we can no longer keep the orders on constants and states independent. This is the case of algebraic datatypes containing function values, where we would like to have, for example, $f \leq [f, g]$ (note that this is not given by the existing order on constants). We therefore define:

$$s \prec_4 C_i(x_1, \ldots, x_n) \iff (\exists i) s = x_i \vee s \prec_4 x_i$$

It is easy but tedious to verify:

Lemma 4. $\preccurlyeq_i$ is a partial order ($i = 1, 2, 3, 4$).

These four distinct orders can now be combined into the final order relation $\sqsubseteq$, which we then show is well-founded (and so appropriate for the SCT analysis):

Definition 6. The $\sqsubseteq$ relation on states is defined to be the transitive closure of the union of the $\preccurlyeq_i$ ($i = 1, 2, 3, 4$).

Lemma 5. The $\sqsubseteq$ relation on states is a well-founded partial order.

Proof. We proceed by defining a *size* $|s|$ for any state s, such that $|s| \in \mathbb{N}$ for each s. We show that whenever $s_1 \sqsubset s_2$, then $|s_1| < |s_2|$. This guarantees that $\sqsubset$ is well-founded, as the natural numbers are well-ordered by $<$.

We first define the size of *atomic* constants:

$$|c| = |\{d \mid d <_A c\}| \text{ for all } c \in Atom$$

The size of an atomic constant is the number of constants that are less than it, a well-defined natural number by well-foundedness of $<_A$. The size of a primitive operator can be arbitrary, say 1.

The size of *algebraic* constants is defined as expected:

$$|C(v_1, \ldots, v_n)| = 1 + \sum_{i=1}^{n} |v_i|$$

The size of an *expression* e is defined as usual and denoted $|e|$.

Finally, we define the size of a state to be:

$$|lhs : \langle v_1, \ldots, v_n \rangle| = |lhs| + 1 + \sum_{i=1}^{n} |v_i|$$

Note that *lhs* is either a constant or an expression, and so $|lhs|$ is a natural number by the above. The measure $|s|$ is then a well-defined natural number since states are finite *trees*.

Note that if $s_1 \prec_2 s_2$, then $s_1 \in support(v_i)$ for some i, where $s_2 = lhs : \langle v_1, \ldots, v_n \rangle$. Therefore clearly

$$|s_1| \leq |v_i| \leq \sum_{j=1}^{n} |v_j| < |s_2|$$

A formal proof of the above would proceed by induction on the depth at which s_1 occurs in the tree s_2.

If $s_1 \prec_1 s_2$, then $s_1 = c_1 : \epsilon$, $s_2 = c_2 : \epsilon$ with $c_1 <_A c_2$, whence $|c_1| < |c_2|$ and so $|s_1| < |s_2|$. The reasoning is similar if $s_1 \prec_3 s_2$.

Finally, if $s_1 \prec_4 s_2$, then $s_2 = C(v_1, \ldots, v_n)$, where $s_1 = v_i$ for some i. In this case $|s_2| > |v_i| = |s_1|$, as required again.

Hence whenever $s_1 \prec_i s_2$ $(i = 1, 2, 3, 4)$, then $|s_1| < |s_2|$. This concludes the proof, as $\sqsubset$ is the transitive closure of the union of the $\preccurlyeq_i$ ☐

We have therefore defined an appropriate notion of size change on states. In the next section we will define the notion of size-change graph, which allows several size-change relations between substates to be recorded in each transition.

3.2 Size-Change Graphs

3.2.1 Definitions

We now wish to define *size-change graphs* for pairs of states. A size-change graph for states (s, s') should track size-change relationships not only between the states s and s', but also between all the substates that make up s and s'.

A size-change graph from a state s to a state s' will be defined as a bipartite graphs, with *input nodes* corresponding to the substates of s and *output nodes* to those of s'. The nature of this correspondence will be detailed first. The actual size-change information is then recorded by arrows from input nodes to output nodes. Note that this is a generalisation of the first-order case, where the nodes of a size-change graphs were simply the parameters of the caller and callee functions.

Graph Bases Recall that any state is a tree, all of whose subtrees (corresponding to internal nodes) are states. Leaf nodes correspond to states with no further environment part. The components, or substates, of a state are defined to be all of its substrees.

We can therefore identify a component of a state by the (unique) path from the root of the tree to the corresponding subtree. The set of all such paths is called the *graph basis* of a state:

Definition 7. The *graph basis* of a state s is defined by:

$$\begin{array}{rcl} gb(Err) & = & \emptyset \\ gb(lhs : \langle v_1, \ldots, v_n \rangle) & = & \{\epsilon\} \cup \bigcup_{i=1}^{n} \{i :: p \mid p \in gb(v_i)\} \end{array}$$

Given a state s and a path p in $gb(s)$, we can find the state identified by p as follows:

Definition 8. Let s be a state. The *valuation function* at s is the map $\overline{s} : gb(s) \to State$ defined by:

$$\begin{array}{rcl} \overline{lhs : vs}(\epsilon) & = & lhs : vs \\ \overline{lhs : vs}(i :: p) & = & \overline{vs_i}(p) \end{array}$$

This is well-defined by definition of $gb(s)$.

Size-Change Graphs We can now define the syntax of size-change graphs and the operations on them. A size-change graph from a state s to a state s' is a bipartite labelled graph, with nodes from $gb(s)$ on the input side and nodes from $gb(s')$ on the output side:

Definition 9. A *size-change graph* γ is a tuple (s, S, s'), where s and s' are states. Such a graph has *type* $s \to s'$. The set $S \subseteq gb(s) \times \{\stackrel{\downarrow}{\to}, \stackrel{=}{\to}\} \times gb(s')$ is a set of *labelled edges*.

We note an important property of size-change graphs, for later use:

Lemma 6. The set of graphs of a given type $s \to s'$ is finite.

Proof. Any graph $\gamma : s \to s'$ is determined by its set of arrows. Let $n = |gb(s)|$, $m = |gb(s')|$. Then there are nm tuples in $gb(s) \times gb(s')$. For every pair $(x, y) \in gb(s) \times gb(s')$ there are only three possibilities: there is no arrow from x to y in S, there is a nonincreasing arrow or there is a decreasing arrow. Hence the number of possible sets S is finite, bounded above by 3^{nm}. □

The import of this is that when we discuss the abstract interpretation that our analysis relies on, the set of types appearing will be finite, and so this guarantees that the total set of graphs is finite.

3.2.2 Operations on Graphs

It will be useful in the sequel to have a number of basic graphs and operations on graphs defined. We group these here, with their basic properties, for future reference.

Basic graphs The basic graphs are the *empty graph* and the *identity graph*.

Definition 10. The *empty graph* from state s to s' is the graph $\emptyset_{s,s'} := (s, \emptyset, s')$.

We will often omit the subscript s, s' when the appropriate states are clear from context. The identity graph on a state is defined as expected:

Definition 11. Let s be a state. Then the identity graph on s, 1_s, is the graph $(s, \{p \overset{=}{\to} p \mid p \in gb(s)\}, s)$.

Composition The most important operation on size-change graphs is *composition*. This is defined as expected: the composition of graphs γ_1 and γ_2 is obtained by pasting the graphs together and collapsing the arrows. The basic operation on sets of arrows is shown below:

Definition 12. Let S_1 and S_2 be sets of arrows. We define S_1S_2 to be the set:

$$S_1S_2 = \{(p, l_1; l_2, q) \mid (\exists r)\ (p, l_1, r) \in S_1 \wedge (r, l_2, q) \in S_2\}$$

where

$$l_1l_2 = \begin{cases} \overset{\downarrow}{\to} & \text{if } l_1 = \overset{\downarrow}{\to} \vee l_2 = \overset{\downarrow}{\to} \\ \overset{=}{\to} & \text{otherwise} \end{cases}$$

Howzever, there is a slight refinement to be made here. The operation above can introduce *redundant edges*. An edge is redundant if is it of the form $p \overset{=}{\to} q$, where the graph also contains the edge $p \overset{\downarrow}{\to} q$. Such edges add no information and complicate matters by making it possible for more than one edge to join two nodes. We therefore wish to maintain the invariant that no graph contains redundant edges. To this end, we assume a function *Simplify* (easily defined) that removes redundant edges from a set of arrows.

The definition of composition for sets of arrows can now be completed:

Definition 13. Let S_1 and S_2 be sets of arrows. We define:

$$S_1; S_2 = \mathit{Simplify}(S_1S_2)$$

The composition operation for size-change graphs is then defined as expected: its effect is just that of composition of arrows, but we ensure that the types match in the natural way.

Definition 14. Let $\gamma_1 = (s_1, S_1, t_1)$ and $\gamma_2 = (s_2, S_2, t_2)$, with $t_1 = s_2$. Then the composition of γ_1 and γ_2 is:

$$\gamma_1; \gamma_2 = (s_1, S_1; S_2, t_2)$$

We then have the easy

Lemma 7. Graph composition is associative.

Union Another fundamental operation on graphs is *union*. Again, this is entirely as expected: to form the union of two graphs, we form the union of their sets of arrows. However, as in the case of composition we ensure that no redundant edges are created, and we ensure that the types match. The precise definition is then:

Definition 15. Let $\gamma_1 = (s_1, S_1, t_1)$ and $\gamma_2 = (s_2, S_2, t_2)$, with $s_1 = s_2$ and $t_1 = t_2$. The union of γ_1 and γ_2 is:

$$\gamma_1 \cup \gamma_2 = (s_1, t_1, \mathit{Simplify}(S_1 \cup S_2))$$

3.3 Multipaths and Termination

The idea of size-change termination can be formulated in the abstract setting of sets of size-change graphs, without reference to any program. This is very briefly described below; details can be found in [18]. The upshot of all of this will be an algorithm for, given a finite set of size-change graphs, determining *precisely* whether an infinite sequence of graphs (a "multipath") is possible or not. This will be what is needed to prove termination of a program, provided that we can "cover" all the calls that can occur with graphs.

Definition 16. A *multipath* $\mathscr{M} = \gamma_0\gamma_1\gamma_2\cdots$ is a finite or infinite sequence of graphs such that $\gamma_i : a_i \to a_{i+1}$ for some sequence of states $a_0, a_1, \ldots$ (that is, the graphs have the appropriate types to be composed).

Definition 17. A *thread* in a multipath $\mathscr{M} = \gamma_0\gamma_1\gamma_2\cdots$ is a finite or infinite connected path of arcs:

$$x_0 \overset{\gamma_N}{\to} x_1 \overset{\gamma_{N+1}}{\to} x_2 \overset{\gamma_{N+2}}{\to} x_3 \cdots$$

Note that a thread need not start at the first graph of $\mathscr{M}$.

Definition 18. A thread is *infinitely descending* if there are infinitely many decreasing ($\overset{\downarrow}{\to}$) arrows along it.

An infinitely decreasing thread is evidently infinite. The assumption that there is at most one edge between two nodes in a size-change graph (no unnecessary edges) guarantees that this is a property of the sequence of graphs (γ_i) and nodes (x_i) in the above, as these determine the arrows.

This is the crucial notion for size-change termination:

Definition 19. Let S be a set of size-change graphs. Then S is *size-change terminating* if for any multipath $\mathscr{M}$ of graphs in S, any infinite thread in $\mathscr{M}$ is infinitely descending.

This condition is the intuitive size-change termination property for sets of size-change graphs. We will show in the next section that, under the assumption that the set S of graphs is appropriate for a program, this implies termination of the program.

3.4 Safety

We first define the notion of adequacy of a graph for a single call. We call a graph γ from state s to s' *safe* if the size-change information it conveys is correct:

Definition 20. Let s and s' be states. A graph $\gamma = (s, S, s')$ is *safe* (for the pair (s, s')) if:

- Whenever $p \overset{=}{\to} q \in S$, then $\overline{s'}(q) \sqsubseteq \overline{s}(p)$
- Whenever $p \overset{\downarrow}{\to} q \in S$, then $\overline{s'}(q) \sqsubset \overline{s}(p)$

To simplify notation, we define:

Definition 21. We write $x \xrightarrow[l]{\gamma} y$ for the statement $(x \xrightarrow{l} y) \in \gamma$, where l is a *label*, one of $\downarrow$ and $=$.

Before proceeding further we note the relationship between the operations on graphs and safety:

Lemma 8. Suppose that γ_1 is safe for (s_1, s_2) and γ_2 is safe for (s_2, s_3). Then $\gamma_1; \gamma_2$ is safe for (s_1, s_3).

Proof. Suppose that $p \xrightarrow[l]{\gamma} q$. Then $p \xrightarrow[l_1]{\gamma_1} r \xrightarrow[l_2]{\gamma_2} q$ for some $r \in gb(s_2)$, with $l_1 l_2 = l$. By safety of γ_1 for (s_1, s_2) and of γ_2 for (s_2, s_3), $\overline{s}(p) \sqsupseteq \overline{s'}(r) \sqsupseteq \overline{v}(q)$, whence $\overline{s}(p) \sqsupseteq \overline{v}(q)$. Also, if $l = \downarrow$, then one of l_1 and l_2 is $\downarrow$, so that by safety one of the inequalities is strict, so $\overline{s}(p) \sqsupset \overline{v}(q)$, as required. □

Lemma 9. Suppose that γ_1 and γ_2 are safe for (s, s'). Then so is $\gamma_1 \cup \gamma_2$.

Proof. Any arc in $\gamma_1 \cup \gamma_2$ is either in γ_1 or in γ_2. □

We can now define safety of a set of graphs for a *program*:

Definition 22. A set G of size-change graphs is *safe for program P* if whenever $P_{start} \to^* s \to s'$, then some graph $\gamma \in G$ is safe for (s, s').

The following result is the crucial connection between size-change graphs and programs:

Proposition 1. Let P be a program, and G be a safe set of size-change graphs for P. If G is size-change terminating, then $P_{start} \Downarrow$.

Proof. Suppose that $P_{start} \not\Downarrow$. Then by the NIS property (Lemma 2) there is an infinite sequence of calls

$$P = s_0 \to s_1 \to s_2 \to \cdots$$

As G is safe for P_{start}, there is a sequence of graphs $\gamma_0, \gamma_1, \ldots$ such that γ_i is safe for (s_i, s_{i+1}) for each i. Let $\mathscr{M}$ be the multipath $\gamma_0\gamma_1\gamma_2\cdots$.

Suppose that $x_0 \to x_1 \to x_2 \to \cdots$ is an infinite thread in $\mathscr{M}$. Then $x_i \xrightarrow{\gamma_{i+N}} x_{i+1}$ for all i (where $N \geq 0$ is fixed — the thread need not start at γ_0).

But by safety of each γ_j for (s_j, s_{j+1}) this implies that

$$\overline{s_N}(x_0) \sqsupseteq \overline{s_{N+1}}(x_1) \sqsupseteq \overline{s_{N+2}}(x_2) \sqsupseteq \cdots$$

As the partial order $\sqsupseteq$ is well-founded, only finitely many of these can be strict. By safety again, only finitely many $\overset{\downarrow}{\to}$ arrows can exist along the infinite thread $x_0 \to x_1 \to \cdots$. Hence G is *not* size-change terminating, as required. □

3.5 The Algorithm

The size-change termination condition above corresponds naturally to the intended semantic meaning of size-change termination, but is not immediately computable. In this section we describe (one of the) algorithms for deciding this. It is surprisingly simple (although very expensive computationally). Full details can be found in [18] again.

Definition 23. Let S be a set of size-change graphs. The *closure* of S is the set

$$\mathrm{Cl}(S) = \{\gamma_1; \gamma_2; \ldots; \gamma_n \mid (\forall i)\gamma_i \in S \wedge \gamma_i; \gamma_{i+1} \text{ exists}\}$$

Trivially, $\mathrm{Cl}(S)$ is the smallest set containing S and closed under composition.

The algorithm is based on the following simple observation:

Lemma 10. Let S be a finite set of size-change graphs. Then $\mathrm{Cl}(S)$ is finite.

Proof. There are only finitely many size-change graphs of any given type $s \to s'$ (Lemma 6). But by definition of $\mathrm{Cl}(S)$, if $\gamma \in \mathrm{Cl}(S)$ and $\gamma : s \to s'$, then there exist graphs $\gamma_1 : s \to t$ and $\gamma_n : t' \to s'$ in S for some t, t'. As S is finite (and each graph has precisely one type), there are only finitely many possible such types and thus $\mathrm{Cl}(S)$ is finite. □

This ensures that the following condition (which we do not prove, see [18]) is decidable:

Definition 24. Let γ be a size-change graph. Then γ is *idempotent* iff $\gamma^2 = \gamma; \gamma$ exists and equals γ. γ has a *in-situ descent* if $x \xrightarrow[\downarrow]{\gamma} x$ for some x.

Theorem 1. A finite set S of size-change graphs is size-change terminating iff each *idempotent* graph in $\mathrm{Cl}(S)$ has an *in-situ* descent.

This completes our description of the size-change termination algorithm. It is straightforward to compute the closure $\mathrm{Cl}(S)$, and to check the idempotent graphs in it.

We note that the closure $\mathrm{Cl}(S)$ can have size exponential in the size of S, and thus in the size of the program. That this is an expensive test is in fact (probably) unavoidable, by the following result (again, see [18]):

Theorem 2. The size-change termination problem for finite sets of size-change graphs is complete for PSPACE.

The above algorithm decides whether a given set of size-change graphs is terminating, and we have shown above that termination of a safe set of size-change graphs implies termination of the program.

It therefore merely remains to find a way of producing a safe set of size-change graphs for a program P. In the next section, we will describe how the semantics can be instrumented to produce safe size-change graphs for each call. We will then give an abstract interpretation that allows an approximation of this to be computed.

4 Producing Size-Change Graphs

4.1 Instrumented Semantics

In this section we add to the operational semantics given in Figure 3 to generate safe graphs with each call. The judgements become $s \to s', \gamma$ and $s \Downarrow v, \gamma$ where the intention is that γ is safe for (s, s') or (s, v) (resp.). The semantics is given in Figure 4.

The different graphs appearing in Figure 4 are defined as follows (where in each case, s represents the left-hand side state and s' the right-hand side state):

- Variable Rule

The graph $1_s^{i \mapsto \epsilon}$ is defined as having the set of arrows:

$$\{i :: p \overset{=}{\to} p \mid p \in gb(s)\}$$

For, taking path $i :: p$ in the state $x_i : vs$ is the same as taking path p in the state vs_i.

Lemma 11. $1_s^{i \mapsto \epsilon}$ is safe for $(x_i : vs, vs_i)$ for any vs (where $s = v_i : vs$).

Proof. Let $s = x_i : vs$, and $v = vs_i$. All the arcs in $1^{i \mapsto \epsilon}$ are of the form $i :: p \overset{=}{\to} p$. For any such arc, $\overline{s}(i :: p) = \overline{x_i : vs}(i :: p) = \overline{vs_i}(p) = \overline{v}(p)$ as required. □

Value

$$\frac{}{s \Downarrow s, 1_s}\ {\scriptstyle s \in Value_{Err}}$$

Variable

$$\frac{}{x_i : vs \Downarrow vs_i, 1^{i \mapsto \epsilon}}$$

Function

$$\frac{}{f : vs \Rightarrow e^{(f)} : \epsilon, \emptyset}$$

Constant

$$\frac{}{Cst\ c : vs \Downarrow c : \epsilon, 1_{c:\epsilon}}\ {\scriptstyle c \in Constant}$$

PrimOp

$$\frac{}{op : \langle v_1, \ldots, v_n \rangle \Downarrow op(v_1, \ldots, v_n), Op^{op}(vs)}\ {\scriptstyle |vs| = \sharp op}$$

AppErr1

$$\frac{e_1 : vs \Downarrow v, \gamma}{e_1 e_2 : vs \Downarrow Err, \emptyset}\ {\scriptstyle v \notin Closure}$$

AppErr2

$$\frac{e_1 : vs \Downarrow s, \gamma_1 \qquad e_2 : vs \Downarrow Err, \gamma_2}{e_1 e_2 : vs \Downarrow Err, \emptyset}$$

CApp-1

$$\frac{}{e_1 e_2 : vs \rightarrowtail e_1 : vs, 1_{\downarrow}}$$

CApp-2

$$\frac{e_1 : vs \Downarrow e_b : ws, \gamma_1}{e_1 e_2 : vs \rightarrowtail e_2 : vs, 1_{\downarrow}}\ {\scriptstyle |ws| < \sharp e_b}$$

CApp-Call

$$\frac{e_1 : vs \Downarrow e_b : ws, \gamma_1 \qquad e_2 : vs \Downarrow v, \gamma_2}{e_1 e_2 : vs \Rightarrow e_b : ws +\!\!+ \langle v \rangle, AppG_i(\gamma_1, \gamma_2)}\ {\scriptstyle i := |ws| < \sharp e_b}$$

CIf-1

$$\frac{}{(e \to e_1 \mid e_2) : vs \rightarrowtail e : vs, 1_{\downarrow}}$$

IfErr

$$\frac{e \Downarrow v, \gamma}{(e \to e_1 \mid e_2) : vs \Downarrow Err, \emptyset}\ {\scriptstyle e \notin \{T:\epsilon, F:\epsilon\}}$$

CIf-T

$$\frac{e : vs \Downarrow T : \epsilon, \gamma}{(e \to e_1 \mid e_2) : vs \Rightarrow e_1 : vs, 1_{\downarrow}}$$

CIf-F

$$\frac{e : vs \Downarrow F : \epsilon, \gamma}{(e \to e_1 \mid e_2) : vs \Rightarrow e_2 : vs, 1_{\downarrow}}$$

Step

$$\frac{s \Rightarrow s', \gamma_1 \qquad s' \Downarrow v, \gamma_2}{s \Downarrow v, (\gamma_1 ; \gamma_2)}$$

Figure 4: Operational Semantics: with Graphs

- Subexpression Rules

These are all the rules of the form $e : vs \to e' : vs$, where e' is a subexpression of e. The graph is then

$$1_s^{\downarrow} = \{\epsilon \xrightarrow{\downarrow} \epsilon\} \cup \{p \xrightarrow{=} p \mid p \in gb(s)\}$$

For, the ϵ node corresponds to the entire state, and $e' : vs$ is *less* than $e : vs$ by the subexpression ordering. All other paths are mapped identically, as the environment is not affected.

Lemma 12. Let e' be a subexpression of e. Then $1_s^{\downarrow}$ is safe for $(e : vs, e' : vs)$, where $s = e : vs$.

Proof. Let $s = e : vs$, $s' = e' : vs$. Then $\overline{s}(\epsilon) = e \succ_3 e' = \overline{s'}(\epsilon)$, so that arc $\epsilon \xrightarrow{\downarrow} \epsilon$ is safe. Also, for all p with $|p| > 0$, $p = i :: q$ for some q, so $\overline{s}(p) = \overline{s}(i :: q) = \overline{vs_i}(q) = \overline{s'}(i :: q) = \overline{s'}(p)$, as required. □

- Constant Evaluation Rule

The constant evaluation rule yields judgements such as $cons : \langle x, y\rangle \Downarrow cons(x, y) : \epsilon$ (where x and y are constant values). That is, it evaluates a primitive operator that has been supplied with all its arguments. The size-change graph for the judgement

$$op : \langle v_1, \ldots, v_n\rangle \Downarrow op(v_1, \ldots, v_n) : \epsilon$$

is defined to be the graph with sets of input nodes $gb(s)$ and output nodes $gb(s') = \{\epsilon\}$ (resp.), and with arrows:

$$\begin{array}{rcl} Op^{op}(v_1, \ldots, v_n) & = & \{\langle i\rangle \xrightarrow{\downarrow} \epsilon \mid v_i > v\} \cup \{\langle i\rangle \xrightarrow{=} \epsilon \mid v_i = v\} \\ & & \text{where } v = op(v_1, \ldots, v_n) \end{array}$$

if $v \in Constant$. If $v \notin Constant$ then op is necessarily a destructor and we take $AbstOp^{op} = \{\langle 1\rangle \xrightarrow{\downarrow} \epsilon\}$.

Lemma 13. $Op^{op}(vs)$ is safe for $(op : vs, op(vs))$.

Proof. If $v \notin Constant$ in the above, then safety is clear as op is a destructor. Now assume that $v \in Constant$. Suppose that $p \xrightarrow{\downarrow} q$ (resp. $\xrightarrow{=}$). Then $q = \epsilon$ and $p = \langle i\rangle$ for some i necessarily, with $v_i > v := op(vs)$ (resp. $v_i \geq v$). Hence $\overline{s}(\langle i\rangle) = v_i \sqsupset v = \overline{v}(\epsilon)$, as required. □

- Call Rule

The call rule is the most complicated rule, to account for the data flow in the call

$$e_1 e_2 : vs \to e_b : ws +\!\!+ \langle v\rangle \ \text{ where } \begin{array}{l} e_1 : vs \Downarrow e_b : ws, \gamma_1 \\ e_2 : vs \Downarrow v, \gamma_2 \end{array}$$

The graph γ_1 records size-change information from the state $s_1 = e_1 : vs$ to $e_b : ws$. As the state $s = e_1 e_2 : vs$ is *greater* than s_1, this is again valid for the pair $(s, e_b : ws)$. As $e_b : ws$ and $s' = e_b : ws +\!\!+ \langle v\rangle$ agree on all variables in which both environments are defined, this is again valid for the pair (s, s'). However, more can be said, as any arc $\epsilon \xrightarrow{=} p$ can be transformed to $\epsilon \xrightarrow{\downarrow} p$ (since s_1 is smaller than s). This can be written concisely as $1^{\downarrow}; \gamma_1$

Lemma 14. Suppose that γ_1 is safe for $(e_1 : vs, e_b : ws)$. Then $1^{\downarrow}; \gamma_1$ is safe for $(e_1 e_2 : vs, e_b : ws +\!\!+ \langle v \rangle)$.

Proof. $1^{\downarrow}$ is safe for $(e_1 e_2 : vs, e_1 : vs)$ and γ_1 is safe for $(e_1 : vs, e_b : ws)$. By lemma 8, $1^{\downarrow}; \gamma_1$ is safe for $(e_1 e_2 : vs, e_b : ws)$. Let $s' = e_b : ws +\!\!+ \langle v \rangle$ and $s = e_b : ws$. Then $\overline{s}(p) = \overline{s'}(p)$ for all $p \in gb(s)$. Hence $1^{\downarrow}; \gamma_1$ is safe for $(e_1 e_2 : vs, s')$ also, as required. □

For the second component of the graph for the function call, we must trace the dataflow from the operand. The graph γ_2 describes the size-change information from the state $s_2 = e_2 : vs$ to the value v. However, the value v is not part of the *top-level* state s', but occurs "one level down". If $|ws| = i$, then we must go down the edge labelled $i + 1$ in the tree corresponding to state s' to find v. This yields the definition:

$$\gamma_2^{\epsilon \mapsto i} = \{p \xrightarrow{l} (i+1) :: q \mid p \xrightarrow{l} q \in \gamma_2\}$$

As before, the graph that we need to use is $1^{\downarrow}; \gamma_2^{\epsilon \mapsto i}$ as e_2 is a subexpression of $e_1 e_2$.

Lemma 15. Suppose that γ_2 is safe for $(e_2 : vs, v)$. Then $1^{\downarrow}; \gamma_2^{\epsilon \mapsto i}$ is safe for $(e_1 e_2 : vs, e_b : ws +\!\!+ \langle v \rangle)$ whenever $|ws| = i$.

Proof. It suffices to show that $\gamma_2^{\epsilon \mapsto i}$ is safe for $(e_2 : vs, e_b : ws +\!\!+ \langle v \rangle)$. Suppose that $p \xrightarrow{l} q$ in $\gamma_2^{\epsilon \mapsto i}$. Then $q = (i+1) :: r$ for some r, with $p \xrightarrow{l} r$ in γ_2. By safety of γ_2, $\overline{v}(r) \overset{l}{\sqsubset} \overline{s_1}(p)$ where $s_1 = e_1 : vs$. Hence if $s' = e_b : ws +\!\!+ \langle v \rangle$,

$$\overline{s'}(q) = \overline{s'}((i+1) :: q) = \overline{e_b : ws +\!\!+ \langle v \rangle}((i+1) :: q) = \overline{v}(q) \overset{l}{\sqsubset} \overline{s_1}(p)$$

as required (as $|ws| = i$). □

We now define a shorthand for the compound graph arising in an application:

$$AppG_i(\gamma_1, \gamma_2) = (1^{\downarrow}; \gamma_1) \cup (1^{\downarrow}; \gamma_2^{\epsilon \mapsto i})$$

We deduce:

Corollary 2. If γ_1 is safe for $(e_1 : vs, e_b : ws)$ $(|ws| = i)$ and γ_2 is safe for $(e_2 : vs, v)$, then $AppG_i(\gamma_1, \gamma_2)$ is safe for $(e_1 e_2 : vs, e_b : ws +\!\!+ \langle v \rangle)$.

Proof. Union preserves safety (Lemma 9). □

- Reduction Rule

The reduction rule is simple: if $s \to s', \gamma_1$ and $s' \Downarrow v, \gamma_2$, then the graph for the combined reduction $s \Downarrow v$ is just $\gamma_1; \gamma_2$. This is valid by Lemma 8.

It is now trivial to prove:

Lemma 16. Suppose that $s \to s', \gamma$ or $s \Downarrow s', \gamma$. Then γ is safe for (s, s').

Proof. By induction on the proof that $s \to s', \gamma$ or $s \Downarrow s', \gamma$, trivial by the above results. □

Note that trivially (since the graph components merely augment the rules),

Lemma 17. $s \to s'$ iff $s \to s', \gamma$ for some γ and $s \Downarrow v$ iff $s \Downarrow v, \gamma$ for some γ.

The two results above guarantee that we have achieved an *instrumentation* of the original semantics, generating safe size-change graphs alongside the ordinary calls in the evaluation of a a program.

4.2 Approximations to Size-Change Graphs

The graphs described above are in a sense the most precise size-change graphs, given our notion of size-change. They describe the dataflow occurring at arbitrary depth in a state transition. However, as a consequence of this the number of nodes of a graph depends on the depth of the environments occurring in a state. This may be unbounded, which implies that there is no *a priori* finite set of graphs from which all the graphs that can arise during execution are drawn.

In the next section, we will describe an abstract interpretation that generates a computable approximation to the set of graphs occurring in a computation. However, this relies crucially on the existence of such an *a priori* finite set. We therefore seek an approximation to size-change graphs, such that the infinite set of all size-change graphs gets mapped to a finite set of graphs, while preserving safety. This will be achieved by restricting the *length* of the paths in the graph bases of states. This approximation is described below.

Let $n \geq 0$ be a natural number. We define the graphs CUT_n and $FORGET_n$ to be:

$$\begin{array}{rcl} CUT_n(s) & = & \{p \stackrel{=}{\to} p \mid p \in gb(s) \wedge |p| \leq n\} \\ & & \cup \{take\ n\ p \stackrel{\downarrow}{\to} p \mid p \in gb(s) \wedge |p| > n\} \\ FORGET_n(s) & = & \{p \stackrel{=}{\to} p \mid p \in gb(s) \wedge |p| \leq n\} \end{array}$$

Note that the *input* nodes (paths) of $CUT_n(s)$ have length at most n, while the *output* nodes of $FORGET_n(s)$ have length at most n.

Lemma 18. $CUT_n(s)$ and $FORGET_n(s)$ are safe for (s, s).

Proof. It suffices to show this for $CUT_n(s)$, as $FORGET_n(s)$ is a subgraph of it. Suppose that $p \stackrel{=}{\to} q \in CUT_n(s)$. Then $p = q$ necessarily, so trivially $\overline{s}(p) = \overline{s}(q)$, as required.

Now suppose that $p \stackrel{\downarrow}{\to} q \in CUT_n(s)$. Then $q = p +\!\!+ r$ for some r necessarily, with $|r| > 0$. Hence

$$\overline{s}(q) = \overline{s}(p +\!\!+ r) = \overline{\overline{s}(p)}(r) \in support(\overline{s}(p))$$

Hence certainly $\overline{s}(q) \sqsubseteq \overline{s}(p)$. Furthermore, as $r \neq \epsilon$ these are distinct (q is a proper substate of p) and thus $\overline{s}(q) \sqsubset \overline{s}(p)$, as required. □

Definition 25. Let γ be a size-change graph. Then define

$$[\gamma]_n := CUT_n; \gamma; FORGET_n$$

(where the implicit input and output states of CUT_n and $FORGET_n$ are chosen to agree with those of γ. *in* and *out* sets of γ).

Corollary 3. Suppose that γ is safe for (s, s'). Then $[\gamma]_n$ is safe for (s, s') (for any $n \geq 0$).

Proof. Composition preserves safety (Lemma 8). □

Note that as there is an upper bound on the branching factor of any node in a state (the maximum number of parameters of any function in the program), there are boundedly many sequences of length at most n in the graph basis of any state s. We can therefore deduce that there are only finitely many possible graphs of the form $[\gamma]_n$ for any graph γ occurring in the execution of given program P. Recall that this was the property that we required to use this approximation in the abstract interpretation.

The approximation to graphs that we have described here will in fact be used with the abstract interpretation to be described in the next section, where it will be crucial that there are only finitely many different graphs appearing. However, it should be noted that it would be possible to replace all graphs γ appearing in the instrumented semantics of Figure 4 by $[\gamma]_n$ (for any fixed n), maintaining correctness of the instrumentation (but losing precision).

An Aside We observe that the operation $\gamma \mapsto [\gamma]_n$ can *introduce* decreasing edges $\stackrel{\downarrow}{\to}$. This is a crucial property for size-change termination. The attentive reader will have observed that all decreasing edges introduced in the instrumented semantics of Figure 4 arise from the constant ($\preccurlyeq_1$) and subexpression ($\preccurlyeq_3$) orders (see Section 3 for definitions). No decreases are introduced in the *substate* order ($\preccurlyeq_2$). This is because the instrumented semantics tracks dataflow at arbitrary depths, and thus instead of saying that a state s is a substate of a state t, we can make the more precise statement that s is equal to a specific state t', which will occur as a substate of t. However, as soon as a finite depth limit is set, decreases in this order are introduced (in the above, t' could occur at too high a depth, and so we cannot use the precise statement in general). This is encapsulated in the $\gamma \mapsto [\gamma]_n$ operation.

5 Abstract Interpretation

We have described an instrumentation of the semantics of our core language that generates safe size-change graphs alongside each call (Figure 4). However, the set of graphs generated in a run of a given program P is, of course, not computable. We are therefore not done yet if we wish to use this as a termination analysis.

We introduce an *abstract interpretation* that allows a safe approximation to be effectively computed. That is, the set of graphs produced in the abstract interpretation should be a superset of the set of graphs produced in the exact semantics. This will allow us to apply the SCT principle to prove termination.

The abstract interpretation is made computable by restricting the states that can appear to be drawn from a finite set. This implies that the inference rules can be exhaustively applied, with this process eventually converging to an overapproximation. We first discuss the definition of the finite set of *abstract states*, then describe the abstract interpretation informally. Finally, we give a precise definition of the abstract interpretation and prove it sound.

5.1 Abstract States

We recall that a state is either *Err*, or is of the form $lhs : vs$ where vs is a list of values (also states) and lhs is:

1. *Either* a subexpression of the program P
2. *Or* an atomic constant

Furthermore, $|vs| \leq \sharp lhs$.

A state is therefore a finite tree, with nodes labelled by subexpressions of P or constants, and with branching factor bounded by some constant B (the maximal arity of any function definition in P). Now the set of all possible states is infinite for two reasons:

1. There are infinitely many possible labels for nodes. This is because there are infinitely many *constants*, as there are only finitely many subexpressions of P

2. The depth of the tree can become arbitrarily large in an execution of P.

Recall that the abstract interpretation will be based on a finite state space. We will therefore define a finite set of *abstract states*, by placing restrictions on both aspects above. This will of course entail a certain loss of precision, but we endeavour to make the precision adjustable (at the cost of a larger state space and so a slower analysis).

We first deal with the issue of constants. We will define a finite set of *abstract constants* to approximate real constants. We will then deal with the depth issue, by defining *abstract states* in which there is an *a priori* upper bound on the depth of the state. Note that our analysis is therefore essentially a k-CFA [24].

5.1.1 Abstract Constants

We must specify a finite set of *abstract constants.* We do not fix a specific set of such constants, but we will require several properties for it. Instances satisfying these properties are easy to construct, and one possible choice is shown in Appendix B.

Abstract Constants Recall that the set of constants could be written as a disjoint union:

$$Constant = Atom \cup AlgConst \cup PrimOp$$

where $Atom$ is the set of atomic constants, $AlgConst$ is the set of algebraic constants and $PrimOp$ is the set of primitive operators. We define

$$BasicConst = Atom \cup AlgConst$$

Recall that $PrimOp$ was assumed to be finite. The set $BasicConst$ can be infinite, however.

We assume the existence of a *finite* set of abstract constants. This can be written:

$$AbstConst = AbstBasic \cup AbstPrimOp$$

where the union is assumed disjoint. Each constant c in $AbstConst$ is assumed to have a unique arity $\sharp c$, and $AbstBasic$ is the set of abstract constants of arity 0 (corresponding to the set $BasicConst$).

The *abstraction function* α_C takes a constant to an abstract constant:

$$\alpha_C : Constant \to AbstConst$$

Our first assumption is that α_C preserves arities:

$$\forall c \in Constant \qquad \sharp(\alpha_C(c)) = \sharp c \tag{AbstConst-1}$$

To simplify notation, for primitive operators op, we will write op^α for $\alpha_C(op)$. Recall that primitive operators are functions from tuples of values to values. We restrict abstract primitive operators to return constants only, as the case of operators returning other values will be treated separately:

$$\forall op \in PrimOp \qquad op^\alpha : Value^{\sharp op} \to AbstBasic_{Err} \tag{AbstConst-2}$$

We will require an assumption that op^α is consistent with op, but we return to this in a moment.

The Information Order To improve the precision of the abstract interpretation, it is desirable to retain *some* information about constants (rather than taking the set *AbstConst* to be a one-point set). We can view an abstract constant as a constraint on constants, restricting the possible range of values; equivalently, an abstract constant represents a set of real constants.

For example, it is possible (and useful) to include the boolean values T and F in the domain *AbstConst*. However, tests such as equality ($=^\alpha$) on abstract values cannot always yield a "precise" answer — the test $c =^\alpha d$, where c and d represent overlapping sets of constants, cannot be determined exactly. We therefore need to assume the existence of a value that is *less defined* than both T and F

To this aim, we assume defined a partial order $\trianglelefteq$ on the set *AbstBasic*. The intention is that $c \trianglelefteq d$ if c is less defined than d. Furthermore, we require:

$$\textit{AbstBasic} \text{ is a } \textit{join semilattice} \text{ under the } \trianglelefteq \text{ order} \qquad (\mathsf{AbstConst\text{-}3})$$

This guarantees that any set S of abstract atomic constants (necessarily finite) has a greatest lower bound $\bigsqcap S$, the most precise value that is consistent with all the values in S. As a consequence, the set *AbstBasic* has a least element $* = \bigsqcap \textit{AbstBasic}$ (we do not use the symbol $\perp$ to avoid confusion with nontermination).

$$* \text{ is the least element of } \textit{AbstBasic} \text{ in the } \trianglelefteq \text{ order} \qquad (\mathsf{AbstConst\text{-}4})$$

We can now state the consistency requirement for abstract primitive operators:

$$(\forall i.d_i \trianglelefteq \alpha_n(c_i)) \Rightarrow op^\alpha(d_1, \ldots, d_n) \trianglelefteq \alpha(op(c_1, \ldots, c_n)) \vee op(c_1, \ldots, c_n) = \mathit{Err}$$
$$\text{whenever } op(c_1, \ldots, c_n) \in \mathit{Constant} \qquad (\mathsf{AbstConst\text{-}Op})$$

That is, computing a value from approximations to the input values yields a value that is less defined (and so consistent with) the value obtained in the exact evaluation. There are two exceptions:

- We do not require abstract operations to produce error values whenever exact ones do. This is not necessary, as error values force termination
- This rule does not apply to primitive operations returning functions. This will be dealt with by a separate mechanism, shown later.

Extensions We have so far presented the set of abstract constants as a means of restricting the state space of the abstract interpretation to a finite set. However, we shall also wish to use abstract constants to define *abstract programs* representing (potentially infinite) classes of programs. This is achieved by substituting abstract constants for concrete constants in expressions. For example, given a program defining a function f, we could prove that f is terminating *on any input* by showing that the expression $f\ *$ terminates. To help with this it will be useful to record a few definitions here.

We first extend the information order $\trianglelefteq$ to abstract primitive operators. This is defined pointwise, as expected:

Definition 26. Let $op_1, op_2 \in \mathit{AbstPrimOp}$. Then $op_1 \trianglelefteq op_2$ iff $\sharp op_1 = \sharp op_2 = n$ say, and for all $vs \in \mathit{AbstBasic}^n$, $op_1(vs) \trianglelefteq op_2(vs)$.

We use the same symbol $\trianglelefteq$ for both orders, and note that we can regard $\trianglelefteq$ as a single order on the disjoint union $AbstBasic \cup AbstPrimOp$.

It is trivial to deduce:

$$(\forall i.d_i \trianglelefteq \alpha_C(c_i)) \wedge op' \trianglelefteq op^\alpha \Rightarrow op'(d_1, \ldots, d_n) \trianglelefteq \alpha(op(c_1, \ldots, c_n)) \quad \text{provided } op(c_1, \ldots, c_n) \in Constant \qquad \text{(AbstConst-Op')}$$

We can now introduce the *least defined* primitive operator for a fixed arity:

Definition 27. We assume the set $AbstPrimOp$ contains

$$op_*^n \quad : \quad (c_1, \ldots, c_n) \mapsto * \qquad \forall (c_1, \ldots, c_n)$$

for all n such that there is an operator of arity n in $PrimOp$ (this is a finite set).

Trivially, $op_*^n \trianglelefteq op$ for any operator op of arity n.

Finally, we define a set of *both* abstract and concrete constants. The intention is that in abstract programs (as introduced above), constants may be drawn either from the set of concrete constants, or the set of abstract constants, as needed.

Definition 28. The set of abstract and concrete constants is defined to be $AllConst = AbstConst \cup Constant$, where the union is assumed disjoint.

It is straightforward to extend the information order $\trianglelefteq$ to the set $AllConst$. We merely add concrete constants as maximal elements, and distinct concrete constants are incomparable. Finally, a concrete constant dominates all the abstract constants that it is consistent with.

Definition 29. The order relation $\trianglelefteq$ is extended to an order $\trianglelefteq'$ on the set $AllConst$ by:

$$c_1 \trianglelefteq' c_2 \iff \begin{cases} c_1 \trianglelefteq c_2 & \text{if } c_1, c_2 \in AbstConst \\ c_1 \trianglelefteq \alpha_C(c_2) & \text{if } c_1 \in AbstConst \wedge c_2 \in Constant \\ c_1 = c_2 & \text{otherwise} \end{cases}$$

We write $\trianglelefteq$ for both relations in the sequel, as $\trianglelefteq'$ is a simple extension of $\trianglelefteq$.

Finally, it is useful to extend the abstraction function α_C to both abstract and concrete constants for ease of notation. This is just the identity on abstract constants:

Definition 30. The map α'_C extends the map α_C to the set $AllConst$:

$$\alpha'_C(c) = \begin{cases} \alpha_C(c) & \text{if } c \in Constant \\ c & \text{if } c \in AbstConst \end{cases}$$

Again, we denote both maps by α_C in the sequel.

This operation is monotonic:

Lemma 19. Suppose that $c_1, c_2 \in AllConst$, with $c_1 \trianglelefteq c_2$. Then $\alpha_C(c_1) \trianglelefteq \alpha_C(c_2)$.

Proof. By cases on c_1, c_2. If $c_1, c_2 \in AbstConst$, then $\alpha_C(c_1) = c_1$ and $\alpha_C(c_2) = c_2$, so this is trivial. If $c_1 \in AbstConst$ and $c_2 \in Constant$, then $\alpha_C(c_1) = c_1 \trianglelefteq \alpha_C(c_2)$ by definition of $\trianglelefteq$. Finally, in other cases, $c_1 = c_2$ by definition of $\trianglelefteq$ and so this is trivial. ☐

5.1.2 Abstract Expressions and Abstract Programs

We have described abstract constants, and have defined the information order on the set *AllConst* of all (abstract and concrete) constants. This suffices for the purpose of restricting the state space to a finite set, and hence we could move on to discuss the restriction on the depth of states which will be crucial in achieving this. However, as mentioned before we will want to prove terminating a whole class of programs with one run of the analysis, in particular to prove that a function is terminating on any input. We record relevant definitions for this purpose here, before resuming our discussion of abstract states in the next section.

We will define *abstract expressions* and *abstract programs*. The intention is that an abstract expression (resp., an abstract program) is the same as a normal expression, except that some concrete constants may have been replaced by abstract constants. The analysis will then proceed on abstract programs (which merely extends the set of concrete programs, so there is no loss of generality). Note that an abstract program may not be executable (any abstract constants appearing prevent this).

Definition 31. The set *AbstExpr* of *abstract expressions* is defined by the same grammar as the set *Expr* of expressions, except that constants are drawn from the set *AllConst* rather than *Constant*.

We define an information order $\trianglelefteq_e$ on the set *AbstExpr*, by structural recursion:

Definition 32. Let $e_1, e_2 \in AbstExpr$. Then

$$e_1 \trianglelefteq_e e_2 \iff \begin{cases} e_2 = c_2 \wedge c_1 \trianglelefteq_c c_2 & \text{if } e_1 = c_1 \\ e_2 = e_1 & \text{if } e_1 = x \\ e_2 = e_1 & \text{if } e_1 = f \\ e_2 = a_2 b_2 \wedge a_1 \trianglelefteq a_2 \wedge b_1 \trianglelefteq b_2 & \text{if } e_1 = a_1 b_1 \\ e_2 = (a_2 \to b_2 \mid c_2) \wedge x_1 \trianglelefteq x_2, x = a, b, c & \text{if } e_1 = (a_1 \to b_1 \mid c_1) \end{cases}$$

where for clarity we have written $\trianglelefteq_c$ for the information order on *AllConst*.

Informally, $e_1 \trianglelefteq e_2$ iff e_1 can be obtained from e_2 by replacing constants by less defined constants in place. In particular, e_1 and e_2 have exactly the same syntax tree.

Now recall that the expressions that appear on the left-hand side of states are *labelled* with a unique identifier, to distinguish between distinct occurrences of syntactically identical expressions. We extend the $\trianglelefteq$ order to such labelled expressions, in the following straightforward manner:

Definition 33. Let $e_1^{l_1}$, $e_2^{l_2}$ be labelled expressions. Then $e_1^{l_1} \trianglelefteq e_2^{l_2}$ iff $l_1 = l_2$ and $e_1 \trianglelefteq_e e_2$.

That is, only expressions that have the same label are comparable.

We can finally define abstract programs and the order relation $\trianglelefteq$ on programs.

Definition 34. The set *AbstProg* of *abstract programs* is defined by the same grammar as the set *Prog* of programs, but with expressions drawn from *AbstExpr* rather than *Expr*.

The order relation allows programs to be compared:

Definition 35. Let P, Q be (abstract) programs, and say that the combinator definitions of P (resp. Q) are

$$\langle f_i\ x_1\ \cdots\ x_{k_i} = e_i \rangle_{i=1}^{n} \text{ and } \langle g_i\ x_1\ \cdots\ x_{l_i} = d_i \rangle_{i=1}^{m}$$

Also suppose that the expression to be evaluated in P (resp. Q) is e_P (resp. e_Q).

Then $P \trianglelefteq Q$ iff $n = m$, $k_i = l_i$ for all i, $e_i \trianglelefteq d_i$ for all i and $e_P \trianglelefteq e_Q$.

In particular, programs P and Q are only comparable if they have exactly the same syntactical structure (up to labels of expressions, as $\trianglelefteq$ only compares expressions that have the same label). Again, we observe that P can be obtained from Q by substituting constants by less defined constants at arbitary points in the syntax tree. This will be useful in order to prove termination of functions on arbitrary inputs.

Finally, we can define the admissible left-hand side components of abstract states:

Definition 36. Let $P \in \mathit{AbstProg}$. Then the P-LHS components are precisely the abstract constants and the subexpressions of P.

The only differences between the above definition and the left-hand side components of ordinary states are that constants are replaced by abstract constants, and that the program P itself is allowed to contain abstract constants.

Finally, we introduce an abstraction function to take a concrete LHS component to an abstract LHS component. This acts as α_C on constants, and as the identity on subexpressions of P:

Definition 37. Let *lhs* be either a subexpression of a program P or a constant. Then

$$\alpha_{lhs}(lhs) = \begin{cases} \alpha_C(lhs) & \text{if } lhs \in \mathit{AllConst} \\ lhs & \text{otherwise} \end{cases}$$

Note that this is a P-LHS component.

5.1.3 Abstract States

We can now conclude our description of abstract states by defining the abstraction of the environment component of a state. Of course, the environment can be arbitrarily deep, which means that there can be unboundedly many states. We must therefore restrict these. As with our discussion of approximations of graphs, we describe this in general, with a *depth* parameter that can be adjusted to yield better precision if required.

The *depth*-1 approximation to a state forgets the environment component entirely, with the exception that it keeps track of the *length* of the list of values representing it. Note that this is bounded above by $\sharp e$, where e is the LHS component of the state, and so that there are only finitely many possible values. It is useful to keep track of the length of the list to distinguish between closures and states that must be reduced. We can therefore define the level-1 abstraction function on states to be:

$$\alpha_1(e : vs) = \alpha_{lhs}(e) : |vs|$$

(all abstraction functions take *Err* to *Err* and so we omit mention of this). The above function has a finite range by the previous remarks. The simplistic depth-1 approximation appears to be precise enough for many purposes, but there is nothing to be gained by not going the whole way. We can define depth-n abstraction functions recursively for arbitrary n:

$$\alpha_n(e : \langle v_1, \ldots, v_k \rangle) = \alpha_{lhs}(e) : \langle \alpha_{n-1}(v_1), \ldots, \alpha_{n-1}(v_k) \rangle$$

A depth-n abstract state is therefore a tree in which all the immediate children of the root node are depth-$(n-1)$ abstract states, as expected. Note that by trivial induction, there are only finitely many depth-n abstract states for any fixed n.

We have defined the depth-1 states as special cases. However, we contend that this is natural in our formulation. For, the intention is that a depth-1 state forgets *all* information about the environment. This is *not* achieved by defining depth-1 states to be just the left-hand side component, as it is necessary to keep track of the number of bound variables. For, consider a function definition $f\ x\ y = e$. The state $e : 0$ represents an abstraction $\lambda xy.e$, the state $e : 1$ represents $\lambda y.e$ and the state $e : 2$ represents e itself. We certainly wish to distinguish between these three objects, justifying our definition of depth-1 states.

Note. In fact, the special case can be taken to be the depth-0 interpretation (which we have left undefined). For, a depth-1 state is a pair $e : k$ where $k \geq 0$. We can identify k with the list of length k over some dummy domain with only one value ($\bullet$ say). This corresponds to the general definition, provided that the only depth-0 state is $\bullet$. We therefore define:

$$\alpha_0(s) = \bullet \text{ for all states } s$$

This is just a notational convenience, to make the treatment of certain cases more uniform.

We now extend the information order $\trianglelefteq$ to all states. Note that this is still only concerned with *constants* — in particular, we do not attempt to define an order where (say) a depth-1 state is "less defined" than a depth-2 state. The informal definition of the $\trianglelefteq$ order on states is as follows:

> $s_1 \trianglelefteq s_2$ if the state s_2 can be obtained from the state s_1 by replacing constant values appearing in s_1 by more defined constant values.

In particular, s_1 and s_2 are required to have exactly the same tree structure to be comparable. We have already defined the information order on left-hand side values, and hence we can define:

Definition 38. Let $s_1 = \mathit{lhs}_1 : \mathit{vs}$ and $s_2 = \mathit{lhs}_2 : \mathit{ws}$. Then $s_1 \trianglelefteq s_2$ iff $\mathit{lhs}_1 \trianglelefteq_{\mathit{lhs}} \mathit{lhs}_2$, $|\mathit{vs}| = |\mathit{ws}|$ and $\mathit{vs}_i \trianglelefteq \mathit{ws}_i$ for all i, $1 \leq i \leq |\mathit{vs}|$.

Finally, note that this applies equally to abstract states, provided we also assume $\bullet \trianglelefteq \bullet$ (for the depth-0 case).

5.2 Abstract Interpretation

Given the abstraction function α_n (for some fixed n), we can define an abstract semantics on abstract states. This should also produce size-change graphs, in such a way that at least one safe graph is produced for each call. This is enough to guarantee the validity of the size-change termination method. We define the abstract interpretation in terms of two relations $\rightarrow_n$ and $\Downarrow_n$. These relate depth-n states to depth-n states.

Many of the rules carry over straightforwardly to the abstract interpretation. However, three rules present challenges. We describe these informally below:

5.2.1 The Application Rule

We recall the application rule:

$$\textsf{CApp-Call}\quad \frac{e_1 : \mathit{vs} \Downarrow e_b : \mathit{ws}, \gamma_1 \qquad e_2 : \mathit{vs} \Downarrow v, \gamma_2}{e_1 e_2 : \mathit{vs} \Rightarrow e_b : \mathit{ws} +\!\!+ \langle v \rangle\ , \mathit{AppG}_i(\gamma_1, \gamma_2)}\ \begin{array}{l}{\scriptstyle |\mathit{vs}| = \sharp e_1 e_2}\\ {\scriptstyle |\mathit{ws}| < \sharp e_b}\end{array}$$

An essential characteristic of this rule is that it can produce closures of arbitrary depth, by storing the value v inside the closure $e_b : \mathit{ws} +\!\!+ \langle v \rangle$. As the abstract interpretation relies on a definite upper bound on the depth of closures, this rule must be adapted, with some (inevitable) loss of information.

More precisely, suppose that (in the abstract interpretation) we have $e_1!vs \Downarrow_n e_b : ws$ and $e_2 : vs \Downarrow_n v$, where $e_b : ws$ and v are abstract states of depth n. Then we cannot deduce $e_1 e_2 : vs \Downarrow_n e_b : ws +\!\!+ \langle v \rangle$, as the depth of the latter abstract state is $n+1$. It is hence necessary to *restrict* the depth of the resulting state to n (note that the above applies equally to depth-1 states, using our convention about depth-0 states).

We therefore define an operation $Restrict_n$ on abstract states. Informally, for any state v of depth n, $Restrict_n(v)$ should be the state of depth $n-1$ obtained from v by forgetting information at depth n.

This is easily defined by recursion on n:

$$\begin{aligned} Restrict_1(s) &= \bullet \\ Restrict_{n+1}(e : vs) &= e : (map\ Retrict_n\ vs) \end{aligned}$$

We will require a slightly more general property:

Lemma 20. Let s be a state, and suppose that $t \trianglelefteq \alpha_n(s)$. Then $Restrict_n(t) \trianglelefteq \alpha_{n-1}(s)$.

5.2.2 The Variable Lookup Rule

In the exact semantics, the variable rule is defined as:

$$\textsf{Variable} \qquad \frac{}{x_i : vs \Downarrow vs_i, 1^{i \mapsto \epsilon}}\ {\scriptstyle |vs| = \sharp x_i}$$

The behaviour of the Variable rule is dual to that of the CApp-Call rule — it *decreases* the depth of closures. In particular, if $x_i : vs$ is a depth-n abstract state, then vs_i has depth $n-1$. However, it is desirable for the $\Downarrow_n$ relation to produce only depth-n states.

It is thus necessary to define an inverse operation to the *Restrict* operation defined earlier. Note however that this inverse cannot be functional, as *Restrict* is certainly many-to-one. We therefore define a *relation* $\ll_n$ such that whenever s is a state of depth $n-1$, $s \ll_n t$ for all states t of depth n that are possible extensions of s. The abstract Variable rule can then be defined as expected. The obvious definition is:

$$\textsf{AbstVariable}_1 \qquad \frac{vs_i \ll_n v}{x_i : vs \Downarrow_n v, 1^{i \mapsto \epsilon}}\ {\scriptstyle |vs| = \sharp x_i}$$

Note however that this ignores the *context* in which vs_i appears (namely, that it is the i^{th} actual parameter of a particular combinator). This leads to an unacceptable loss of precision. In particular, in the depth-1 case, each value vs_i is the dummy value $\bullet$, and so there is no information available. We can correct this by amending the rule to

$$\textsf{AbstVariable} \qquad \frac{x_i : vs \ll_{n+1} x_i : ws}{x_i : vs \Downarrow_n ws_i, 1^{i \mapsto \epsilon}}\ {\scriptstyle |vs| = \sharp x_i}$$

The intuitive interpretation of this rule is that we first promote the entire depth-n state to a depth-$(n+1)$, which has the effect of increasing the depth of each of the substates *in context*. We then only keep the i^{th} substate. This is the simplest description of this rule, but for efficiency it is obviously necessary to avoid performing expensive computations on all the substates that will be ignored. This is a minor implementation point, however, and need not concern us here.

5.2.3 The Operator Evaluation Rule

The operator evaluation rule has the form (in the exact semantics):

$$\mathsf{PrimOp}\qquad \frac{}{op : \langle v_1, \ldots, v_k\rangle \Downarrow op(v_1, \ldots, v_k), Op^{op}(v_1, \ldots, v_k)}$$

where the v_i are constants. There are two cases to consider:

- If $op(v_1, \ldots, v_k)$ is a constant c, then we must ensure that the corresponding rule in the abstract interpretation returns a constant that is at least as defined as $\alpha_C(c)$.
- If $op(v_1, \ldots, v_k)$ is a higher-order value v, then (at depth n) the abstract op rule must return some value that is at least as defined as $\alpha_n(v)$.

These two aspects are essentially different and will be treated independently.

Constant Return Values We first consider the case where the return value is constant. The rule for producing a return value in the abstract interpretation (depth > 1) is formally identical to the PrimOp rule in the exact semantics. For, recall that for each operator op we assume defined an operator $op^\alpha := \alpha_c(op)$ which satisfies Equation AbstConst-Op. This is precisely what is required above, and the abstract interpretation is defined so that the only values that can appear on the left-hand side of a rule are abstract (so an operator appearing in the abstract semantics is op^α, as required).

We must treat the depth-1 case as a special case, as we have no information about the arguments to op in that case. The following reduction is always valid:

$$\frac{}{op : k \Downarrow op(*, \ldots, *)}\ {\scriptstyle k=\sharp op}$$

by Equation AbstConst-Op again.

Size-Change Graphs We do not define the size-change graph to be produced in the abstract interpretation PrimOp rule, to maintain the separation between the treatment of constant values and the rest of the analysis.

We merely assume defined for each operator op an abstract size-change graph construction:

$$AbstOp^{op}(av_1, \ldots, av_k) \text{ where } av_i \in AbstConst$$

The required property of this is that:

If $av_i \trianglelefteq \alpha_C(v_i)$ for each i, then $AbstOp^{op}(av_1, \ldots, av_k)$ is safe for the above reduction

That is, $AbstOp^{op}(avs)$ is a conservative approximation to the graph produced by $Op^{op}(vs)$ whenever the avs are conservative approximations to the vs.

Again, for the depth-1 case the appropriate graph is $AbstOp^{op}(*, \ldots, *)$.

It is easy to define suitable graph constructions $AbstOp^{op}$, in particular in the case of algebraic datatypes. We invite the reader to refer to Appendix B for a description of one possible choice of set of abstract constants, and the resulting definition of the graph $AbstOp^{op}$ for each operator op.

Higher-Order Return Values Finally, we must consider the case where the return value of the primitive operator evaluation is a higher-order value. By assumption this can only occur if *op* is a destructor for an algebraic datatype. In this case $\sharp op = 1$ necessarily, and the reduction has the form (in the exact semantics):

$$op : \langle C(v_1, \ldots, v_n) \rangle \Downarrow v_i \text{ for some } i$$

The graph $AbstOp^{op}(av)$ is safe for this reduction for *any* abstract value *av*, so that there is no need to treat the graph construction as a special case.

However, we must produce a return value for the reduction in the abstract semantics. We deal with this in a very simple-minded manner – we assume that the reduction can produce *any function value that could have been stored in any constant.* This is very crude and could lead to substantial loss of precision, but is sound (and could be improved upon fairly straightforwardly). Note that values potentially stored in constants are easy to identify, as the arguments to primitive operators (necessarily constructors)

We refer to this as *escape analysis*, and say that a function value that is stored in a constant *escapes*. The rule for the depth-1 case is similar to closure analysis:

$$\frac{e_1 e_2 \in subexps(P) \qquad e_1 : \sharp e_1 \Downarrow_1 op' : k \qquad e_2 : \sharp e_2 \Downarrow_1 v}{op : n \Downarrow_1 v, AbstOp_1^{op}(n)} \; \begin{array}{l} n{=}\sharp op \\ op \text{ is a destructor} \\ op' \text{ is a constructor} \end{array}$$

For the depth-n case, we just use the depth-1 rule to produce all possible depth-1 values, and promote those to depth-n states.

We will conclude our description of the abstract interpretation in the next section, where the $\ll_n$ relation is described in detail.

5.3 The Abstract Interpretation: Precise Description

Having described the issues and motivation behind the abstract interpretation, we are now in a position to define it precisely. This is shown in Figures 6 and 7 (appendix B). We note that it is *not* necessary to include the error rules in the abstract interpretation. For, error states are final states and therefore cannot appear in a nonterminating run, which is what we must identify.

5.3.1 The $\ll_n$ Relation and Closure Analysis

We must define the $\ll_n$ relation on states. Recall that the intention is that whenever s is a state of depth $n-1$, then $s \ll_n t$ for all states t of depth n that could give rise to s under restriction. More precisely, we will require the following property:

For any reachable state s, if $t \trianglelefteq \alpha_{n-1}(s)$, then $t \ll_n w$ for some $w \trianglelefteq \alpha_n(s)$.

where the definition of reachability will be stated formally later. That is, whenever t is a valid depth-$(n-1)$ approximation to s, then $t \ll_n w$ for some w a valid depth-n approximation to s.

The $\ll_n$ relation is defined by recursion on n. The inductive case is simple:

$$\frac{(\forall i) \qquad v_i \ll_{n-1} w_i}{lhs : \langle v_1, \ldots, v_k \rangle \ll_n lhs : \langle w_1, \ldots, w_k \rangle} \; n{>}2$$

That is, to increase the depth of a state of depth at least 2, we increase the depth of each of its substates.

The base case is $\ll_2$. This relation takes a depth-1 state on the left-hand side, and returns all possible depth-2 extensions of it. Now a depth-1 state is of the form $lhs : k$ for some $k > 0$. There are two cases: either $lhs = e'$ for some $e' \trianglelefteq e \in subexps(P)$ or $lhs = c$ for some abstract constant c. The latter case breaks down into two subcases: c an atomic constant (in which case $vs = \epsilon$) or c a primitive operator. We deal with each of these cases independently.

If $s = c : \epsilon$ for c an atomic or algebraic constant, there is nothing to do:

$$\frac{}{c : 0 \ll_2 c : 0}\; {\scriptstyle c \in AbstBasicConst}$$

In the other two cases, the left-hand side is unchanged, but we must find an approximation to the set of possible values for each component of the right-hand side. We introduce a relation $\multimap$ to do the actual work. The $\ll_n$ is defined as:

$$\frac{(\forall i) \qquad CombDef(e), i \multimap v_i}{e : k \ll_2 e : \langle v_1, \ldots, v_k \rangle} \qquad \frac{(\forall i) \qquad op, i \multimap c_i}{op : k \ll_2 e : \langle c_1, \ldots, c_k \rangle}$$

There are therefore two cases for the $\multimap$ relation:

- $f, i \multimap v$, where f is a combinator definition. This should hold whenever v is the depth-1 approximation to a possible value of the i^{th} parameter of f.
- $op, i \multimap v$, where op is a primitive operator. This is assumed to hold whenever v is less defined than *any* value v_i such that $op(v_1, \ldots, v_n) \neq Err$ for some v_j. This depends only on the operator op. Note that we can take $v = *$ for all op, i — this certainly satisfies the above requirements.

For the $f, i \multimap v$ case, we use *closure analysis*. We observe that if v was bound to the i^{th} variable of f, then this must have been the result of evaluating an expression $e_1 e_2$, where e_1 reduces to f with $i - 1$ arguments, and e_2 reduces to v. By the subexpression property, these must have been subexpressions of the program P. This gives:

$$\frac{e_1 e_2 \in subexps(P) \qquad e_1 : \sharp e_1 \Downarrow_1 e^{(f)} : (i-1) \qquad e_2 : \sharp e_2 \Downarrow_1 v}{f, i \multimap v}$$

This completes our description of the $\ll_2$ relation. Note that as mentioned previously, this depends on $\Downarrow_2$, and hence the two relations are mutually recursive.

5.3.2 Escape Analysis

Escape analysis has been described above for the depth-1 case and is straightforward. The general case (depth-n) is obtained by putting:

$$\frac{e_1 e_2 \in subexps(P) \qquad e_1 : \sharp e_1 \Downarrow_1 op' : k \qquad e_2 : \sharp e_2 \Downarrow_1 v \qquad v \ll_n^* w}{op : vs \Downarrow_1 w, AbstOp_1^{op}(n)} \; \begin{array}{l} {\scriptstyle |vs| = \sharp op} \\ {\scriptstyle op \text{ is a destructor}} \\ {\scriptstyle op' \text{ is a constructor}} \end{array}$$

where $\ll_n^*$ takes a depth-1 state to a set of depth-n states, defined by:

$$s \ll_n^* t \iff s = s_1 \ll_2 s_2 \ll_3 \cdots \ll_{n-1} s_{n-1} \ll_n s_n = t \text{ for some } s_i$$

It is convenient to have some notation for the escape analysis, and so we split up the above judgement into two inference rules:

$$\frac{e_1 e_2 \in \mathit{subexps}(P) \quad e_1 : \sharp e_1 \Downarrow_1 op' : k \quad e_2 : \sharp e_2 \Downarrow_1 v}{\mathit{ESCAPES}(v)} \; {\scriptstyle op' \text{ is a constructor}}$$

$$\frac{\mathit{ESCAPES}(v) \quad v \ll_n^* w}{op : vs \Downarrow_n w} \; {\scriptstyle |vs| = \sharp op \atop op \text{ is a destructor}}$$

The complete specification of the abstract interpretation can be found in Figures 6 and 7 (Appendix B). To illustrate these ideas in a slightly simpler form, however, we show the depth-1 case in Figure 5 (this section). Note that the $\ll_2$ relation does *not* appear in this, as its use for closure analysis has been incorporated directly into the variable lookup rule. The $\ll$ relation is of course used for the general depth-n case.

5.3.3 Soundness of the Abstract Interpretation

We will now prove that the abstract interpretation defined in this section is *sound*, in the sense that the set of graphs generated by the abstract interpretation (computable) overapproximates the set of graphs generated by the instrumented semantics (uncomputable). We will show that this gives a sound criterion for program termination, as required.

We have set up the abstract interpretation to operate over *abstract programs* – programs with concrete constants replaced by abstract constants. This essentially allows us to prove termination of a *class* of programs with one analysis, and in particular we can prove termination of a function on arbitrary input values. This complicates the statement of the soundness theorem somewhat, but is worth the extra effort.

Throughout this section, we will let P denote a concrete program and AP denote an abstract program.

Definition 39. Let AP be an abstract program. The result of the depth-n abstract interpretation is defined to be the set of graphs:

$$S_n(AP) = \{\gamma \mid (\exists t, t') AP_{start} \rightarrow_n^* t \rightarrow_n t', \gamma\}$$

The result that we are aiming for is:

Theorem 3 (Soundness). Let P be a (concrete) program, and $AP \trianglelefteq \alpha(P)$. Then $S_n(AP)$ is safe for P.

Note. As a technical aside, the definition of safety needs to be modified slightly from Definition 22, see Definition 45 below.

That is, the set of graphs generated by the abstract interpretation for AP is a safe approximation to the exact set of graphs generated by the execution of *any* program P that AP could represent.

This is the required result, by Proposition 1:

Corollary 4. If $S_n(AP)$ is size-change terminating, and $AP \trianglelefteq \alpha(P)$, then $P \Downarrow$.

The proof of Theorem 3 proceeds by showing that any call in the exact semantics is mirrored by a call in the abstract interpretation, by induction on the computation tree leading to that call. The details are left to Appendix A.2, and we merely state the main definitions and results here.

We first introduce the notion of a state being valid for a particular program:

Value

$$\frac{}{s \Downarrow_1 s, 1_s}\ {\scriptstyle s \in AbstValue_{Err}}$$

Function

$$\frac{}{f : n \Rightarrow_1 e^{(f)} : 0, \emptyset}$$

Variable

$$\frac{e_1 e_2 \in subexps(P) \qquad e_1 : \sharp e_1 \Downarrow_1 f_{x_i} : i - 1 \qquad e_2 : \sharp e_2 \Downarrow_1 v}{x_i : n \Downarrow_1 v, 1^{i \mapsto \epsilon}}$$

Constant

$$\frac{}{Cst\ c : n \Downarrow_1 c^{\alpha} : 0, 1_{c:\epsilon}}\ {\scriptstyle c \in Constant}$$

PrimOp

$$\frac{}{op : n \Downarrow_1 op(*, \ldots, *), AbstOp_1^{op}(n)}$$

PrimOpEscape

$$\frac{e_1 e_2 \in subexps(P) \qquad e_1 : \sharp e_1 \Downarrow_1 op' : k \qquad e_2 : \sharp e_2 \Downarrow_1 v}{op : n \Downarrow_1 v, AbstOp_1^{op}(n)}\ \begin{array}{l}{\scriptstyle n = \sharp op} \\ {\scriptstyle op \text{ is a destructor}} \\ {\scriptstyle op' \text{ is a constructor}}\end{array}$$

CApp-1

$$\frac{}{e_1 e_2 : n \rightarrowtail_1 e_1 : n, 1_{\downarrow}}$$

CApp-2

$$\frac{e_1 : n \Downarrow_1 e_b : k, \gamma_1}{e_1 e_2 : n \rightarrowtail_n e_2 : n, 1_{\downarrow}}\ {\scriptstyle k < \sharp e_b}$$

CApp-Call

$$\frac{e_1 : n \Downarrow_1 e_b : k, \gamma_1 \qquad e_2 : n \Downarrow_1 v, \gamma_2}{e_1 e_2 : n \Rightarrow_1 e_b : k + 1, [AppG_i(\gamma_1, \gamma_2)]_1}\ {\scriptstyle k < \sharp e_b}$$

CIf-1

$$\frac{}{(e \rightarrow e_1 \mid e_2) : n \rightarrowtail_1 e : n, 1_{\downarrow}}$$

CIf-T

$$\frac{e : n \Downarrow_1 v : 0, \gamma \qquad v \leqslant T^{\alpha}}{(e \rightarrow e_1 \mid e_2) : n \Rightarrow_1 e_1 : n, 1_{\downarrow}}$$

CIf-F

$$\frac{e : n \Downarrow_1 v : 0, \gamma \qquad v \leqslant F^{\alpha}}{(e \rightarrow e_1 \mid e_2) : n \Rightarrow e_2 : n, 1_{\downarrow}}$$

Step

$$\frac{s \Rightarrow_1 s', \gamma_1 \qquad s' \Downarrow_1 v, \gamma_2}{s \Downarrow_1 v, (\gamma_1 ; \gamma_2)}$$

Figure 5: Abstract Interpretation, Depth-1

Definition 40. Let AP be an abstract program. An abstract state $s = \mathit{lhs} : \mathit{vs}$ is an *AP-abstract state* if lhs is either a subexpression of AP or an abstract constant, and each element vs is an AP-abstract state. We then write $\mathit{Valid}_{AP}(s)$.

The only nontrivial part of the soundness proof is the proof that closure analysis, and escape analysis, successfully approximate the set of values that could occur in the evaluation of a program. Recall that we prove soundness by showing that each call $s \to s'$ in the exact semantics is mirrored by some call $t \to t'$ in the abstract interpretation (with conditions on t, t'). However, as the abstract interpretation depends on the program AP to be analysed, this does not hold for *any* states s, s', but only for states that could occur in the evaluation of P.

To deal with this issue, we introduce the following notion of *reachability* for a (concrete) state. First, define

Definition 41. Let v be a value, f a function name and c a constant. We define:

$$\begin{aligned}
\mathit{ClosureReach}_{AP}(f, i, v) &\equiv (\exists w \trianglelefteq \alpha_1(v))\ f, i \multimap_{AP} w \wedge \mathit{Valid}_{AP}(w) \\
\mathit{EscapeReach}_{AP}(v) &\equiv v \in \mathit{Constant} \vee ((\exists w \trianglelefteq \alpha_1(v))\ \mathit{ESCAPES}(w) \wedge \mathit{Valid}_{AP}(w))
\end{aligned}$$

We can now define reachability for a state:

Definition 42. Let AP be an abstract program, $AP \trianglelefteq \alpha(P)$ and let $s = \mathit{lhs} : \mathit{vs}$ be a P-state. We define a predicate $\mathit{Reachable}_{AP}(s)$ by cases: if $\mathit{lhs} = e$ is an expression, then

$$\mathit{Reachable}_{AP}(e : \mathit{vs}) \equiv (\forall i \leq |\mathit{vs}|)\, \mathit{ClosureReach}_{AP}(\mathit{CombDef}(e), i, \mathit{vs}_i) \wedge \mathit{Reachable}_{AP}(\mathit{vs}_i)$$

If $\mathit{lhs} = \mathit{op}$ is a primitive operator, then

$$\mathit{Reachable}_{AP}(\mathit{op} : \mathit{vs}) \equiv (\forall v \in \mathit{vs})(\mathit{Reachable}_{AP}(v) \wedge \mathit{EscapeReach}_{AP}(v))$$

If $\mathit{lhs} = c = C(x_1, \ldots, x_k)$ then

$$\mathit{Reachable}_{AP}(c : \epsilon) \equiv (\forall i) \mathit{Reachable}_{AP}(v_i) \wedge \mathit{EscapeReach}_{AP}(v_i)$$

Finally, if $\mathit{lhs} = c$ is an atomic constant then $\mathit{Reachable}_{AP}(c)$.

Thus, a state s is reachable iff its substates (and higher-order values in constants) could be approximated by closure analysis and escape analysis. We shall show that whenever $P \to^* s$, then $\mathit{Reachable}_{AP}(s)$, but this must be done in parallel with the proof of soundness of the depth-1 abstract interpretation.

To simplify notation in what follows, we collapse the three different kinds of reduction judgements ($\rightarrowtail$, $\Rightarrow$ and $\Downarrow$) into one:

Definition 43. We define $\xrightarrow{s} = \rightarrowtail$ (a "subexpression" judgement), $\xrightarrow{c} = \Rightarrow$ (a "call" judgement) and $\xrightarrow{e} = \Downarrow$ (an "evaluation" judgement). The notation $\xrightarrow{l}$ stands for any of the three, with $l = s$, c or e as appropriate.

We can now formulate the precise property that we can prove by induction on the computation tree:

Definition 44. The depth-n analysis of an abstract program AP is said to be correct with respect to P if $AP \trianglelefteq \alpha(P)$, and the following condition holds:

Suppose that $P_{start} \rightarrow_P{}^* s \xrightarrow{l}_P s'$, with $s' \neq \mathit{Err}$. Let t be an AP-abstract state, with $t \trianglelefteq \alpha_n(s)$. Then there exists t' an AP-abstract state, with $t' \trianglelefteq \alpha_n(s')$ and $t \xrightarrow{l}_{APn} t', \gamma$ for some graph $\gamma : t \to t'$ safe for (s, s').

It is now possible to prove (see Appendix A.2):

Lemma 21. The depth-1 analysis of an abstract program AP is correct with respect to any program P with $AP \trianglelefteq \alpha(P)$. Also, whenever $P \to^* s$, then $\mathit{Reachable}_{AP}(s)$.

This base case proves correctness of closure analysis, giving:

Lemma 22. Let s be a P-state with $\mathit{Reachable}_{AP}(s)$. Then for any AP-abstract state $t \trianglelefteq \alpha_n(s)$, there exists $w \trianglelefteq \alpha_{n+1}(s)$ with $t \ll_{n+1} w$ and $\mathit{Valid}_{AP}(w)$.

and its trivial corollary:

Corollary 5. Let s be a P-state with $\mathit{Reachable}_{AP}(s)$. Then for any AP-abstract state $t \trianglelefteq \alpha_1(s)$, there exists $w \trianglelefteq \alpha_n(s)$ with $t \ll_n^* w$ and $\mathit{Valid}_{AP}(w)$.

We can now conclude that the depth-n analysis is correct:

Lemma 23. The depth-n analysis of an abstract program AP is correct with respect to any program P with $AP \trianglelefteq \alpha(P)$, for all $n \geq 1$.

Finally, we must adapt the definition of safety (Definition 22). For, this was stated in terms of sets of graphs arising in the execution of the concrete program. However, graphs in the set $S_n(AP)$ are labelled with *abstract states*, and therefore this set cannot strictly be safe for P with the previous definition. We amend this:

Definition 45. A set G of depth-n size-change graphs is *safe for P* if whenever $P_{start} \to^* s \to s'$, then some graph $\gamma \in G$, $\gamma : t \to t'$ where $t \trianglelefteq \alpha_n(s)$, $t' \trianglelefteq \alpha_n(s')$, is safe for (s, s').

Note. There is now a minor issue with using Proposition 1 to prove termination from a safe set of size-change graphs. For, this relies on identifying graphs corresponding to *self-loops* $s \to s$. However, with the amended definition the most we can say is that such a self-loop is covered by a graph $t \to t'$, where $t, t' \trianglelefteq \alpha_n(s)$, but we do not necessarily have $t = t'$.

It suffices, however, to consider equivalence classes with respect to the relation $t \equiv t'$ iff t and t' can represent the same abstract state when identifying self-loops. This is a minor detail left to the reader.

The proof of soundness is completed by giving the proof of Theorem 3:

Proof of Theorem 3. Let P be a concrete program, and $AP \trianglelefteq \alpha(P)$. Suppose that $P_{start} \to^* t \to t'$. We must show that that there exists $\gamma \in S_n(AP)$, $\gamma : s \to s'$, such that $s \trianglelefteq \alpha_n(t)$, $s' \trianglelefteq \alpha_n(t')$ and γ is safe for (s, s'). Now there exists a sequence

$$P_{start} = t_0 \to t_1 \to \cdots \to t_n = t \to t' = t_{n+1}$$

where possibly $n = 0$. We define a sequence of terms s_i by induction on n, such that:

1. $s_i \trianglelefteq \alpha_n(t_i)$ for each i, and
2. $s_i \to_n s_{i+1}, \gamma_i$ for some $\gamma_i : s_i \to s_{i+1}$ safe for (t_i, t_{i+1}).

We can certainly take $s_0 = \alpha_n(t_0)$. Furthermore, given s_i as above, Lemma 23 gives a state t_{i+1} with the required property.

The above guarantees the existence of states $s_n \trianglelefteq \alpha_n(t)$, $s_{n+1} \trianglelefteq \alpha_n(t')$ and a graph γ_n satisfying the required property. Note than γ_n does lie in $S_n(AP)$, as $AP_{start} \to_n^* s_n \to_n s_{n+1}, \gamma_n$. This concludes the proof. ☐

5.3.4 Algorithm

We now give a very brief overview of the algorithm for deciding our termination criterion. Recall that the general pattern for size-change termination is:

- Extract a safe set of size-change graphs for the program.
- Decide whether the set of size-change graphs satisfies the SCT condition.

As previously noted, the second aspect is independent of the program analysis (except for the minor point that abstract states need to be related to concrete states, see above) and we have adopted the algorithm from [18].

The program analysis is conceptually trivial (at least in our current implementation of it): exhaustively apply the inference rules given above until a fixed point is reached.

Thus, we keep a current set of *reachable states*, as well as the current $\xrightarrow{l}$ relations ($l = s$, c or e). At each iteration, each reachable state is inspected to see if it can appear as the left-hand side of any judgement, and any such judgements are added to the current relations (updating the set of reachable states as needed). The process stops when the set of states and the judgement relations do not change; this will happen as the domain of these is finite.

Finally, as each new judgement is added we record the associated graph to use in the second phase, thus generating exactly the set $S_n(AP)$ above. This procedure is certainly inefficient, but serves as an experimental implementation of the above abstract interpretation. This concludes our presentation of the size-change termination analysis.

6 Analysing ML Programs

We have described the size-change termination analysis in terms of a very simple functional language (which we refer to as the *core* language). This language is unrealistically restricted, but this allowed a simpler presentation, unencumbered by irrelevant details of the input language. However, the onus is now on us to show that this is enough to prove termination of programs written in a reasonable language.

The language that we have chosen to implement this translation for is a substantial subset of (the purely functional part of) OCAML [19]. Of course, the same ideas apply to give a translation of many similar strict functional languages. The transformations used in translating OCAML into our core language are well-known (see, for example, [23]). For this reason, the exact details are left to Appendix C.

However, it should be noted that we must ensure that the translation preserves termination exactly, and in particular cannot turn a nonterminating program into a terminating one. This is evidently crucial to the use of this transformation in the frontend of our analysis. As these transformations are usually described in the context of compiler design, where introducing termination can be considered correct, this is not entirely covered by existing presentations. Correctness should be obvious from the description in Appendix C, however.

6.1 The Input Language

The input language that we consider is essentially the purely functional side of OCAML [19]. We do not attempt to describe this precisely here, and assume that the OCAML language is known. Again, Appendix C describes some points of interest.

The features of OCAML that we do *not* include are:

1. Reference cells and imperative features. These would evidently require changes to the analysis itself, and we do not discuss this here.[1] In particular, imperative features include the object-oriented features of OCAML, which we ignore here.

2. Exceptions. We do not consider OCAML's support for exceptions.

3. The module system. The ML module system goes beyond simple namespace control, with functors and module values. We have ignored this aspect entirely here, and leave this as future work (although support for a restricted namespace control feature would be simple).

4. Function values stored inside datatypes. This is a technical restriction that does not appear entirely necessary (in fact, see later in this section for a caveat), but we have kept it for simplicity.

Features of our subset of OCAML that are not present in the core language are essentially λ-abstractions, local **let**(**rec**) definitions and pattern matching. These are the features that must be removed by the transformation.

6.2 The Transformation

The transformation of OCAML into our core language is described in Appendix C. The two main components of it are:

- λ-lifting and block floating [10, 4]. These eliminate λ-abstractions and local function definitions from the program by lifting all such definitions to the top level.

- Compiling pattern matching. Pattern matching expressions are first entirely moved to **match** statements, and these are then compiled into chained **if** statements.

6.3 Limitations

Our treatment of function values in datatypes is quite basic, and could be improved substantially – in particular if complex programs such as parser combinator libraries [20] are to be proved terminating. However, there is an encoding of datatypes as higher-order values that allows precision to be increased, which we describe below.

It is very common to use *tuples* of functions such as (f, h). This is important enough that our translation from OCAML treats it as a special case to improve precision, and encodes a tuple as a closure. A simple encoding of the pair (x, y) would be

$$(x, y) \qquad \Rightarrow \qquad \lambda b.\mathbf{if}\ b\ \mathbf{then}\ x\ \mathbf{else}\ y$$

This is just an ordinary closure, and so in particular x and y can be higher-order values. Increasing the depth of the analysis increases the precision at which x and y are kept. Our implementation treats arbitrary n-tuples in a similar way.

It should also be noted that this encoding gives a covert way of encoding arbitrary datatypes so that arbitrary values can be stored. For example, one could (in our untyped language) represent lists as nested pairs:

$$[x_1, \ldots, x_n] \qquad \Rightarrow \qquad (x_1, (x_2, , \ldots, (x_{n-1}, x_n)))$$

[1] Though reference cells (but not **while** loops) could be dealt with using a similar approach to the closure analysis, by finding an approximation to the set of possible values of a reference cell during a computation

This, through the above encoding of pairs, would allow lists to contain arbitrary elements.

More sophisticated, but similar, encodings could deal with all algebraic datatypes (we further need to ensure that datatypes can be distinguished from one another, this is left to the reader). Furthermore, it appears (see the higher-order fold left example from Section 7) that good size-change information can be obtained on the equivalents of the standard operations on datatypes with these representations.

We have not implemented this translation automatically, mainly because this would make the reporting of potential nonterminating loops messy and unintelligible. We simply note that this could be performed by hand where necessary, to increase precision when analysing programs that use function values stored in datatypes.

7 Expressive Power and Examples

Having described the higher-order size-change termination analysis, we should now justify its usefulness by showing that its *expressive power* - the set of programs that it can prove terminating - is reasonably high. We can distinguish between two notions of expressiveness:

- *Extensional Power*: The set of *functions* that can be represented by a program proved terminating by the SCT analysis.
- *Intensional Power*: The set of *programs* that are proved terminating

Extensional power is mainly of theoretical interest, though it is easier to characterise than intensional power. We have not investigated this aspect for our analysis. Note however that the first-order SCT analysis can express all the *multiple recursive functions* [22, 1], an extension of the primitive recursive functions that includes Ackermann's function. The extensional expressive power is therefore likely to be quite high.

We will attempt to characterise the (intensional) expressive power of the higher-order SCT analysis in several ways:

- By proving that it is at least as powerful as various other criteria for termination. These *partial completeness* proofs serve to reassure us that the class of programs that we have identified is large enough
- By giving an indication of what can and cannot be achieved by looking at representative *examples*
- By trying it out on large bodies of existing programs, to attempt to find out whether it is widely applicable to real programs.

Note. At present, the third aspect above is largely left as further work. We hope to perform experiments on existing test suites, in particular those arising from examples in theorem proving.

7.1 Examples

We first describe some representative examples of what the higher-order SCT analysis can and cannot do. In many cases we do not give details here but refer to previous work.

7.1.1 First-Order Examples

It is useful to consider simple first-order programs to get an idea of the boundaries of the size-change termination method. However, this is not specific to this version of the analysis, and we refer the reader to [18, 5] for a more in-depth discussion.

Note however that the size-change termination method deals easily with all simple functions such as *factorial*, *concat*, *reverse* ... In addition, the Ackermann function is also proved terminating, although it is not primitive recursive. Mutual recursion between several different functions and permutations of parameters are also handled.

It is easy to give an artificial example which causes the SCT analysis to fail:

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

This is in fact terminating, but this happens only because the decrease in x in the call $g \to f$ is larger than the increase in x in the call $f \to g$. This illustrates one way to foil the SCT analysis.

Another, more problematic restriction is shown by the following program for *Quicksort*:

$$\begin{array}{lcl} qsort\ xs & = & \textbf{if}\ \mathsf{null}\ xs\ \textbf{then}\ []\ \textbf{else}\ (qsort\ ys) +\!\!+ [w] +\!\!+ (qsort\ zs) \\ & & \quad \textbf{where}\quad \begin{array}{rcl} w :: ws & = & xs \\ (ys,\ zs) & = & (filter\ (< x)\ ws,\ filter\ (\geq x)\ ws) \end{array} \end{array}$$

This is not found terminating. For, there is no size-change information available for the two self-calls $qsort \to qsort$. Of course, we know that as *filter p ws* is no longer than *ws* for any predicate p, and that *ws* is shorter than *xs* in the above this is terminating. However, there is no inference of the size of the *return value* of a function.

This problem is a major one, as it substantially restricts the range of natural programs to which the SCT analysis can be applied. Note that there are many obvious analyses that can deduce that the return value of *filter* is no larger than its argument. However, these need not concern us here as they are accessory to our main point. We leave it as future work to integrate these within our framework.

7.1.2 Higher-Order Examples

We now give a few example of the SCT analysis in action on some nontrivially higher-order programs. Note that the use of higher-order functions often arises purely out of the use of functionals such as *map* and *fold*. These are quite easily dealt with by the SCT analysis — most natural instances of *map*, *foldl*, *foldr* ... are size-change terminating (these functions are defined by primitive recursion).

We wish to show a few examples that are more challenging than the above. We also refer the reader to [12] — we will later prove that the higher-order SCT is at least as powerful as the λ calculus analysis , whence the examples given there are also applicable to our analysis. These examples show that higher-order functions, including the **Y** combinator, can be handled by the analysis. We merely repeat one of these examples, to show the effect of the depth parameter.

The Effect of Depth We recall the expression for Ackermann's function written using the **Y** combinator, given in [12], section 5.5. The original version had to use two distinct occurrences of the **Y** combinator for the two recursive loops, so as not to confuse the closure analysis.

We show, with an adapted version of this example, that the depth parameter does increase precision, and can solve this kind of problem. Consider the following program (this is presented in named combinator form for ease of understanding, it is equivalent to the obvious λ expression)

$$
\begin{aligned}
Y\ f \quad &= (\lambda q \to f(\lambda s.q\ q\ s))\ (\lambda q \to f(\lambda s.q\ q\ s))\\[1ex]
g\ a\ m \quad &= \textbf{if } m == 0 \textbf{ then } (\lambda v \to v+1)\\
&\quad\ \textbf{else } Y\ h\ (a\ (m-1))\\[1ex]
h\ b\ f\ n &= \textbf{if } n == 0 \textbf{ then } f\ 1\\
&\quad\ \textbf{else } f\ (b\ f\ (n-1))\\[1ex]
Y\ g\ *\ *
\end{aligned}
$$

Analysis of the above program using depth 1 fails, as the two recursive loops are then confused by the control flow analysis, which prevents accurate parameter size information from being found. However, a depth 2 analysis succeeds, thus proving that any program obtained by substituting actual constants for the $*$ abstract values is terminating. This shows that increasing the depth of the analysis can be useful in removing spurious loops in the control flow.

Higher-order fold left It is well known that the standard functional programming function *fold left* (on lists) can be rewritten as a fold right. To achieve this, it is written as a higher-order fold right, returning a function that builds the value computed by fold left, when given the start value. Written in OCAML, this gives:

$$
\begin{aligned}
&\textbf{let } \mathit{fleft}\ (\oplus)\ e\ \mathit{xs} =\\
&\quad \textbf{let } \mathit{step}\ x\ f\ a = f(a \oplus x) \textbf{ in}\\
&\quad \mathit{foldr}\ \mathit{step}\ \mathit{id}\ \mathit{xs}\ e;;
\end{aligned}
$$

where for clarity we have used the *foldr* function, rather than CAML's *fold_right*. We can then write (say) the *reverse* function using this version of fold left:

$$\textbf{let } \mathit{rev} = \mathit{fleft}\ (\mathit{flip}\ (::))\ []$$

This is then proved terminating on any input by the analysis.

It is interesting to analyse this example a bit further. Other than the (easy) recursive call of *foldr* to itself, the analysis identifies a self-loop, from *step* to itself. Now the possible values of the parameters of *step* are found as follows: for x, any constant; for a, a list constant; for f, either $\mathit{id} : 0$ or $\mathit{step} : 3$. Recall that the possible values are depth-1 abstract states, *i.e.* either constants or expressions with the *number* of evaluated arguments. The $f = \mathit{id} : 0$ case is not a problem.

The $f = \mathit{step} : 3$ case leads to the self-loop (as *step* calls f). However, in the loop $\mathit{step} \to \mathit{step}$, the parameter f is found to be decreasing (in place). For, such loops occur only if the *caller* *step* has as f parameter $f_0 = \mathit{step} : 3$. In this case, the value of the f parameter in the *called* *step* is $f_0.f$, which is a proper *substate* of f_0. Hence this function is proved terminating for the nontrivially higher-order reason that closures are getting smaller at each self-call.

Lazy Functional Programs It is natural to ask whether the SCT analysis can be extended to lazy functional programs. We first note that it suffices to consider termination of programs under call-by-name evaluation, rather than full lazy evaluation. For, a lazy functional program terminates iff it terminates under call-by-name (though the call-by-name equivalent may be inefficient). Furthermore, there is an embedding of call-by-name inside a call-by-value language, using *suspensions*. Specifically, a *lazy value* is represented as a function taking a dummy argument, and returning the value. The primitive operations are:

$$
\begin{array}{l}
\textbf{type } \alpha \textit{ lazy} = \textit{unit} \rightarrow \alpha \\
\\
\textbf{let } \textit{force } x = x \ () \\
\\
\textbf{let } \textit{fix } (f : \alpha \textit{ lazy} \rightarrow \alpha \textit{ lazy}) = \\
\quad \textbf{let rec } x \ () = f \ x \ () \textbf{ in} \\
\quad x;;
\end{array}
$$

The *force* function forces evaluation of a lazy value, while the *fix* functions "ties the knot" to compute the fixpoint of a function of a lazy argument.

We can therefore ask whether the *existing* analysis is sufficient to deal with programs that rely on laziness, by encoding the programs using these primitives (which can be performed automatically) and analysing the result.

As an example, consider the program *repmin* [2]. This takes a binary tree as input, and returns a tree of the same shape, with all values at the leaves replaced with the minimum value of the original tree. This can be written as a *circular program* to traverse the tree only once. In Haskell say, we could write this as:

$$
\begin{array}{l}
\textbf{data } \textit{Tree } \alpha = \textit{Tip } \alpha \mid \textit{Fork } (\textit{Tree } \alpha) \ (\textit{Tree } \alpha) \\
\\
\textit{repmin} :: \textit{Ord } \alpha \Rightarrow \textit{Tree } \alpha \rightarrow \textit{Tree } \alpha \\
\textit{repmin } t = t' \\
\qquad\qquad \textbf{where } (t', m) = \textit{rpm } t \ m \\
\\
\textit{rpm } (\textit{Tip } x) \ m = (\textit{Tip } m, x) \\
\textit{rpm } (\textit{Fork } t_1 \ t_2) \ m = (\textit{Fork } t_1' \ t_2', m_1 \textbf{ min } m_2) \\
\qquad\qquad\qquad\qquad \textbf{where } (t_1, m_1) = \textit{rpm } t_1 \ m \\
\qquad\qquad\qquad\qquad\qquad\quad\ (t_2, m_2) = \textit{rpm } t_2 \ m
\end{array}
$$

We can translate this into an OCAML program by using explicit suspensions (and of course *fix* to implement the circularity). The function *rpm* will then take a tree and a lazy α, and will return a lazy pair of a lazy tree and a lazy α. This gives:

$$
\begin{array}{l}
\textbf{let rec } \textit{rpm } t \ m = \\
\quad \textbf{match } t \textbf{ with} \\
\qquad \textit{Tip } x \rightarrow \lambda().(\lambda().\textit{Tip } (\textit{force } m), \lambda().x) \\
\quad \mid \textit{Fork } (t_1, t_2) \rightarrow \textbf{let } (t_1', m_1) = \textit{force}(\textit{rpm } t_1 \ m) \\
\qquad\qquad\qquad\qquad\ \textbf{and } (t_2', m_2) = \textit{force}(\textit{rpm } t_2 \ m) \textbf{ in} \\
\qquad\qquad\qquad\qquad\ \lambda().(\lambda().\textit{Fork } (\textit{force } t_1', \textit{force } t_2'), \\
\qquad\qquad\qquad\qquad\qquad \lambda().(\textit{force } m_1) \textbf{ min } (\textit{force } m_2));; \\
\\
\textbf{let } \textit{repmin } t = \\
\quad \textbf{let } f \ p = \textit{rpm } t \ (\lambda().\textit{force } (\textit{snd } (\textit{force } p))) \textbf{ in} \\
\quad \textbf{let } (t', m) = \textit{force } (\textit{fix } f) \textbf{ in} \\
\quad \textit{force } t';;
\end{array}
$$

This can now be translated directly into our core language (by the implementation). The resulting program is proved terminating by the depth-2 analysis.

7.2 Partial Completeness

This is very much in progress. Detailed proofs can be found in Appendix D, but we state the results here. We make a distinction between two kinds of partial completeness results. The first kind compares this analysis to other versions of the SCT analysis, it should be regarded as a "sanity check" of the validity of our analysis. The other, more interesting kind of partial completeness result compares this analysis to other, unrelated termination criteria. These results are all conjectures at the present time, but we hope to settle some soon.

7.2.1 Completeness Relative to Other SCT Analyses

We now wish to compare our analysis to two other analyses based on the size-change termination principle. The first is the analysis of the pure untyped λ-calculus [12] on which ours is based. The second is the original first-order size-change termination analysis [18].

The λ-Calculus SCT Analysis The λ-calculus analysis described by Neil Jones and Nina Bohr [12] proves termination of a single λ-expression (Λ say) under call-by-value evaluation. Now our analysis works on a different language, namely the named combinator core language described in Section 2. However, λ-lifting provides a faithful translation from one to the other.

The result is therefore as expected:

Theorem 4. Let Λ be a λ-expression, and suppose that P_Λ is the program obtained by λ-lifting Λ. Then whenever Λ is found terminating in the λ-calculus analysis, then P_Λ is terminating in the higher-order SCT analysis.

The proof can be found in Appendix D.2. Note that to avoid unnecessary duplication, we do not describe the λ-calculus analysis, see [12] for details.

The First-Order SCT Analysis The first-order SCT analysis is not uniquely defined in [18]. Instead, this paper describes the basic graph analysis, relying on some program analysis to produce a safe set of size-change graphs. Such an analysis is described in [5], and we aim to prove partial completeness of our analysis with respect to this formulation. This seems straightforward, but is left as future work.

7.2.2 Other Termination Criteria

Simple Types Any simply typeable λ-expression is *strongly normalisable*, so that it is guaranteed to be terminating under any evaluation strategy, in particular call-by-value (for a proof, see [7]). There is an open conjecture that any simply typeable λ-expression is also size-change terminating in the sense of [12].

We wish to examine this conjecture in the context of our analysis. The named combinator language setting may make it easier to reason about. The other advantage is that the precision of our analysis can be increased by increasing the depth parameter. This is left as future work.

Attribute Grammars Recall our discussion in the examples above of lazy functional programs, and their embedding into a call-by-value language. We are particularly interested in *circular* programs, making nontrivial use of call-by-name to sort out evaluation order.

Attribute grammars [14] were originally introduced as a way of describing the semantics of programming language, but are now more commonly used to describe syntax-directed translation in compilers. They are of particular interest to us because of a correspondence between attribute grammars and circular lazy functional programs [11]. This can be viewed either as an implementation of attribute grammars or as a description of programs such as *repmin*.

Attribute grammars allow very general data dependencies, due to the interplay between inherited and synthesised attributes. Accordingly, not every attribute grammar is well-formed. An attribute grammar can fail to be well-formed by exhibiting a *circular attribute dependency* — for some string in the language, some attribute actually depends on itself. There is a well-known criterion due to Knuth [14] that tests for these. Any grammar that passes the test is guaranteed to not to possess any circular dependencies on any input.

We are interested in this because of the result that the lazy functional program derived from an attribute grammar always terminates, provided that the grammar is Knuth acyclic. The converse does not hold, however. Using the encoding given above, we can apply the SCT analysis to the functional program obtained from an attribute grammar. This has been attempted on two representative examples (the block structure of Algol and the binary notation example from [14]). Furthermore, there are similarities between Knuth's criterion and size-change termination.

We are therefore moved to formulate the following conjecture:

Conjecture. Whenever an attribute grammar is Knuth acyclic, the functional program obtained from it is size-change terminating.

We will examine this conjecture in future work.

8 Conclusion and Future Work

To conclude our discussion of the higher-order size-change termination analysis, we now give a few possible ideas for future work along the same lines as this work. These range from simple extensions of the method as it stands to very different contexts in which the same principles might be applicable.

8.1 Related Work

Strongly Terminating Functional Languages Many languages have been proposed that are *strongly terminating*, in the sense that any valid program in such a language is guaranteed to terminate under any evaluation order (in particular, call-by-value). Typed subsets of the λ-calculus provide canonical examples (simple types and system F, among others) [7]. Examples of functional languages using similar ideas, together with restrictions on recursion, include Elementary Strong Functional Programming [26], Hagino's Categorical Programming Language [9] and Charity [3]. We are interested in comparing the expressive power of these languages with size-change termination. Languages that enforce termination through typing have the advantage of allowing a clearly defined class of programs, although the programming patterns they impose can be restrictive.

Termination Analyses of Functional Programs Theorem provers such as HOL [8] typically include termination analyses for functional programs, in order to prove soundness when defining a recursive function in the logic. These analyses are usually not fully automatic in our sense, however, as they rely on the user to provide some information (such as an explicit lexicographic order) to identify descent in many cases. An important example of a toolkit for assisting with termination proofs (that works with several theorem provers) is TFL, by Konrad Slind [25].

Other Work on the SCT Principle We give a brief overview of other work on the SCT principle, many of the following have already been mentioned. The original SCT analysis was described by Chin Soon Lee, Neil Jones and Amir Ben-Amram in [18]. A simple implementation is given in [5]. The SCT analysis has been applied to termination of partial evaluation in [13]. The extensional power of the SCT analysis was characterised in [1], and a polynomial-time approximation to the SCT analysis was defined in [16]. Chin Soon Lee's PhD thesis [15] gives a very complete development of SCT and proposes an extension to higher-order functions. However, it appears that the approach taken is less general than the present approach. Finally, the λ-calculus SCT analysis was described by Neil Jones and Nina Bohr in [12].

8.2 Future Work

8.2.1 Examples and Applications

As mentioned in Section 7, it would be desirable to test out our analysis on a substantial body of real programs, to assess its usefulness on real examples. In addition, one should choose the set of programs to contain the kind of programs that one might want to use the higher-order SCT analysis on. One such application is to *theorem proving*, and we will focus on this.

The theorem prover HOL [8] allows functions to be defined, so that they can be used inside formulae as function symbols. For example, the following definition can be found in the standard HOL theory for arithmetic:

$$(FACT\, 0 = 1) \wedge (\forall n. FACT\, (\mathsf{succ}\ n) = (\mathsf{succ}\ n) \cdot (FACT\, n))$$

This defines the factorial function by primitive recursion. However, to guarantee the soundness of the logic, such definitions must be checked to verify that they define *total* functions, that is to say that the equivalent functional program terminates. This is at present accomplished by simple syntactic checks in HOL, while complicated cases have to be dealt with by hand (for example, by specifying an explicit lexicographic order or variant).

There are two interesting consequences of this:

- HOL examples and theories provide an interesting source of test programs for evaluating the applicability of the SCT analysis
- It would be interesting to integrate the SCT analysis into HOL, to be able to use this method to prove termination automatically, at least in many cases. To achieve this, it is necessary to extend the analysis to produce an explicit *proof* of termination, rather than just a statement that the program terminates.

8.2.2 Arbitrary Higher-Order Parameters

A simple extension of the higher-order SCT analysis is to prove termination of programs that are not completely specified. This has already been done in the simplest case — in order to prove termination of functions such as *fac* on any natural number input, we have generalised the analysis to allow *unspecified constants* to be used in place of actual constants at any place in the program (see Section 5).

The use of unspecified constants is evidently useful, as we would usually like to proved termination of a function on any input. We would like to extend this to unspecified higher-order (functional) parameters. For example, consider the higher-order fold left example (Section 7). We can prove termination of the expression

$$\mathit{fleft}\ \mathit{op}^2_*\ *\ *$$

Now $*$ stands for an arbitrary atom (primitive constant of arity 0), while op^2_* stands for any primitive operator of arity 2. This guarantees that any expression such as *fleft* $(+)\ 0\ xs$, *fleft* $(*)\ 1\ xs$ terminate for any list xs. However, we cannot say anything about

$$\mathit{reverse}\ xs = \mathit{fleft}\ (\mathit{flip}\ (::))\ []\ xs$$

as *flip* (:) is not a primitive operator. Intuitively however, it is clear that the above terminates.

We consider that the only satisfactory proof of termination of a higher-order functional such as *fleft* is to show that *fleft* $f\ e\ xs$ terminates for *all* functions f. However, some restrictions evidently have to be made — at the very least, we must assume that f itself is terminating.

We leave this simple-seeming extension as future work. This includes working out a general enough condition on f making this statement true. We conjecture, however, that it is possible to prove a program terminating for an arbitrary choice of function f such that:

> f is a function of arity n such that for any values $vs = \langle v_1, \ldots, v_n \rangle$, $f : vs$ terminates with result either an atomic constant or *Err*

8.2.3 Improving Efficiency

The Complexity of SCT The size-change termination method is extremely expensive computationally. This is largely unavoidable, as the SCT condition for sets of size-change graphs is PSPACE-complete [18]. That is, the problem of determining whether a set of graphs satisfies the SCT criterion is complete for PSPACE, where the input size is the size of the set of graphs. Note however that some approximations to the SCT criterion on graphs (which is otherwise exact) have been devised [16, 17] that have small-order polynomial (quadratic and cubic) time complexities. We would like to investigate this further to see whether good results can still be obtained for the kind of graph that our higher-order analysis typically produces.

Furthermore, combinatorial explosion can occur in the generation of the set of graphs. For, the size of the state space is exponential in the depth of the analysis. It is easy to give a rough estimation of the size of the state space. Note that abstract states of depth n are trees of depth n, whose branching factor is bounded above by the largest number of parameters of any function in the program, B say — this can be expected to be small, however. The number of internal nodes in such a tree is therefore $\Theta(B^{n-1})$. Now each internal node is labelled by a subexpression of the program. The number of subexpressions of the program corresponds to its size, say N. There are hence at least $N^{B^{n-1}}$ abstract states. Note that this does not take into account states including abstract constants, although the number of these is small in our implementation, as there are few abstract constants.

The size of state space is therefore extremely large, making the SCT analysis prohibitively expensive on large examples. An important question, however, is whether this is a theoretical maximum, attained only in pathological examples, or whether real programs behave badly in the analysis. We lack data to settle this, but note that circular programs and attribute grammars seem to cause large state spaces to arise, as almost all states are connected under the call relation.

Improving Efficiency In light of these issues it is natural to ask whether any special measures can be taken to control the computational complexity of the analysis. We leave this as future work at present. There are many approaches that could be attempted to achieve this, and we detail one here. All ideas involve varying the depth of the analysis to use higher depths only where needed.

We first observe that the depth n analysis relies on the depth 1 analysis, for any $n > 1$. The closure analysis is only performed at depth 1, and this relies heavily on knowledge of the abstract interpretation of the whole program. It is hence undoubtedly unavoidable to perform a complete depth-1 analysis of the program. However, all further analyses are in fact local analyses, relying only on the depth 1 analysis. It is therefore possible to perform an analysis at depth $n > 1$ starting at any state, provided that the analysis at depth 1 has already been computed, so that the $\multimap$ relation is known.

The idea is therefore to analyse the program at depth 1, and to compute the closure of the set of graphs generated in this way. Now it may be the case (if this is not terminating) that certain loops in the program cannot be proved terminating at this depth. One could then do a depth n (for a fixed n) analysis of the specific part of the program that caused a problem. The result that one hopes for then is that said loop will *either* be proved size-change terminating, *or* simply disappear as the higher precision allows for a better approximation to the call graph.

Details of the above method need to be worked out in order to guarantee soundness, and hopefully completeness relative to the depth-n analysis, so that the variable-depth analysis above proves exactly the same programs terminating as the depth n analysis. This should then be implemented and tried out on real examples to ensure that it does help.

8.2.4 Improving the Expressive Power

There are many ways in which the expressive power of the SCT analysis could be improved. We detail two below.

Increasing Parameters A simple problem with the SCT analysis is that any *increase* in a parameter throws it completely, even if followed by a bigger decrease. It is easy to showcase this in an artificial example (which we recall from Section 7):

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

It would be possible to deal with this kind of case by extending the type of *arrows* occurring in size-change graphs. For, at present we only allow the labels $=$ and $\downarrow$. However, it would be possible to fix any finite set of these without affecting soundness, given an appropriate notion of composition of labels. One possibility is to take a set of labels

$$\{\downarrow_k \mid 1 \leq k \leq N\} \cup \{\uparrow_k \mid 1 \leq k \leq M\} \cup \{=\}$$

where the meaning of $\downarrow_k$ is that the value decreases by at least k units, and $\uparrow_k$ indicates that the value increases by at most k units. We can then deduce that $\uparrow_k \cdot \downarrow_{k+1} = \downarrow_1$, effectively dealing with the above problem. This is a simple extension, although its usefulness in real programs is not obvious.

Return Values The main source of inaccuracy in the SCT analysis is the lack of information about the *return value* of functions. Again, we refer back to the *Quicksort* example from Section 7. It is a simple inductive proof to show that the size of the return value of *filter* is no bigger than its input, and that is all that is needed in order for the *Quicksort* program to be size-change terminating. However, this is beyond the scope of our analysis at present.

We are therefore interested in attempting to combine the higher-order size-change termination analysis with an inference for the size of the return values of function. Such an inference is given by *sized types* [21]. In this work, one could give *filter* the type $\mathit{List}\ k\ \alpha \to \mathit{List}\ k\ \alpha$. This type is an assertion that *filter* terminates on any input, and that given a list of size at most k it returns a list of size at most k. This guarantees that *filter* always returns a list that is no longer than its input.

We would therefore be interested in incorporating this or other inferences into the size-change termination analysis, to increase its expressive power.

8.2.5 Comparison with other Analyses

As mentioned in Section 7, *partial completeness* proofs yield an interesting indication of the power of the analysis. Apart from the "technical" results comparing this version of the SCT analysis to previous incarnations of the idea, there are many interesting comparisons that could be made. These can be categorised into three types:

- Comparing the SCT analysis with other, related termination analyses.
- Comparing the SCT analysis with typing disciplines ensuring strong normalisation, such as simple types in the λ calculus
- Comparing the SCT analysis with seemingly unrelated criteria, such as the Knuth acyclicity test for attribute grammars

All of the above are open questions at present.

8.2.6 Extension to other Languages

It would be interesting in future work to investigate whether the ideas presented here can be extended to other languages. The obvious candidates for this are general *term rewriting systems*. These subsume both strict and lazy functional programming languages, and termination issues in term rewriting systems form an important area of study. This would be a possible extension of this work.

Another, more modest language extension would be to extend the idea to the imperative features of ML. As we have hinted in a footnote in Section 6, it should be possible to deal with reference cells by computing an approximation to the set of values that it might take during a computation. This would extend the set of programs that can be accepted substantially. However, it does not allow **while** loops to be dealt with, as these rely on values *changing* during the computation, whereas the obvious approximation to the set of values of a reference cell is fixed throughout the computation. It remains to be seen how easy it would be to extend the SCT analysis to this imperative construct.

Acknowledgements

I would like to thank my supervisor Oege de Moor for his constant support and help on this project and other work. Also, I would like to thank Neil Jones for encouraging me to work on this in the first place, and taking a very close interest during his visit to Oxford in 2004 and since then. Finally, many thanks for detailed proofreading go to Fiona Woollard.

A Proofs

A.1 Semantics (Section 2)

Lemma 2. **If**. Suppose that there is no infinite call sequence $P := s_0 \to s_1 \to \cdots$. We show that $P \Downarrow$ by induction on the length of the longest call sequence $P := s_0 \to s_1 \to \cdots \to s_n$. In fact we show that this holds for *any* state s_0. Note that the case s_0 a value or $s_0 = Err$ is trivial. We can hence assume that $s_0 = e : vs$, where $|vs| = \sharp e$.

Base Case: $n = 0$. Then e is not a function name, an application or an **if** statement (otherwise $s_0 \to s_1$ for some s_1 by rules Function, CApp-1, CIf-1). Hence e is either a constant or a variable. By rules Constant and Variable, $s_0 \Downarrow$, as required.

Inductive Step. By cases on the state s_0. If e is a constant or a variable then $P \Downarrow$ as before. If e_P is a function name, f say, then $s_0 \Rightarrow e^{(f)} : \epsilon := s_1$ But then any call chain from s_1 has length at most $n-1$ (as any call chain from s_0 has length at most n), whence [IH] $s_1 \Downarrow$. Hence by rule Step, $P = s_0 \Downarrow$. The other cases are similar, and are sketched below:

If $e = e_1 e_2$, then $s_0 \to e_1 : vs$, whence [IH] $e_1 : vs \Downarrow$. If $e_1 : vs \Downarrow e_b : ws$ where $|ws| < \sharp e_b$, then $s_0 \to e_2 : vs$, whence [IH] $e_2 : vs \Downarrow v$ say. If $v \neq Err$, then $s_0 \Rightarrow e_B : ws \mathbin{+\!\!+} \langle v \rangle$ by CApp-Call, whence we are done by Step. If $e_1 : vs \Downarrow s$ for s not a closure, or $e_2 : vs \Downarrow Err$, then $s_0 \Downarrow Err$, as required.

If $e = (b \to e_1 \mid e_2)$, then $s_0 \to b : vs$, so [IH] $b : vs \Downarrow v$ for some v. If $v \notin \{T, F\}$ then $s_0 \Downarrow Err$. Otherwise, suppose wlog that $v = T$. Then by CIf-T, $s_0 \Rightarrow e_1 : vs$, as required by Step as before.

Only If. Suppose that $s \Downarrow$, where wlog $s = e : vs$. We show that there is no infinite call chain starting at s by induction on the proof that $s \Downarrow$ and cases on s. We assume for a contradiction that there exists some infinite chain $s \to t \to \cdots$.

If s is a value or if e is a variable, a constant expression or a constant, then $s \not\to s'$ for any state s', so the result is trivial.

Suppose that e is a function name, f say. Then necessarily $t = e^{(f)} : \epsilon$ by inspection. We must show that any call chain from t is finite for a contradiction. Now the last rule used to show $s \Downarrow$ was necessarily Step. Now $s \Rightarrow s'$ implies $s' = t$, whence this shows that $t \Downarrow$, with a shorter proof than $s \Downarrow$. Hence [IH] any call chain from t is finite. The other cases are again similar, we sketch the proofs below:

Suppose that $e = e_1 e_2$. Then as $s \Downarrow$, this is by one of the rules AppErr1, AppErr2 and Step. We consider the latter first. Then the only state such that $s \Rightarrow s'$ is $s' := e_b : ws \mathbin{+\!\!+} \langle v \rangle$ where $s_1 := e_1 : vs \Downarrow e_b : ws$ and $s_2 := e_2 : vs \Downarrow v$ (necessarily unique by determinism). Hence rule CApp-Call applies, whence $s_1 \Downarrow$ and $s_2 \Downarrow$, with shorter proofs. Furthermore $s' \Downarrow$ as $s \Downarrow$ by rule Step. Hence [IH] there are no infinite call chains starting at s_1, s_2 or s'. But by inspection these are the only states that s calls, as required. We now consider one of the error cases, AppErr1 say. Suppose that $s_1 \Downarrow v$ for some value v not a closure. Then [IH] there is no infinite call chain starting at s_1. Also, s calls no other state than s_1, as by determinism, $s_1 \not\Downarrow c$ for any closure c. Hence there can be no infinite call sequence starting at s.

Finally, suppose that $e = (b \to e_1 \mid e_2)$. This is again similar: $s \Downarrow$ by Step necessarily (the error case is similar to the previous case and hence omitted). Now if $s \Rightarrow s'$, then s' is one of $s_1 := e_1 : vs$ and $s_2 := e_2 : vs$ (wlog s_1). Then $b : vs \Downarrow T$ with a shorter proof, whence there is no call chain starting at $b : vs$. Also, $s_1 \Downarrow$ with a shorter proof also (by the antecedent for Step). Hence [IH] there is no call chain starting at s_1. Finally, by determinism $b : vs \not\Downarrow F$, whence $s \not\to s_2$. Hence s calls no other states, completing the proof. $\square$

Lemma 3. By induction on the structure of the proof that $s \to s'$ or $s \Downarrow s'$. Suppose that $SUB(s)$ holds. The cases are:

- The Value rule is trivial. The Constant rule and all the error rules are vacuous.
- The PrimOp rule: suppose $SUB(op : vs)$ holds, and let $w = op(vs)$. We must show that $SUB(w)$ holds. If $w = Err$ or $w \in Atom$ then this is vacuous. Otherwise, $w \in AlgConst$. Now $op \in PrimOp = AtomOp \cup AlgOp$, and as the result of op is not in $Atom$, by assumption $op \notin AtomOp$. Hence $op \in AlgOp$ and op is not a test (as the result of a test is a boolean

value). If op is a constructor C_i, then $w = C_i(v_1, \ldots, v_n)$, so $expsup(w) = \bigcup_{i=1}^{n} expsup(v_i) = expsup(op : \langle v_1, \ldots, v_n \rangle)$. Hence if $SUB(op : vs)$, then $SUB(w)$.

Finally, suppose that op is a destructor. As $c \neq Err$, $vs = \langle v \rangle = \langle C_i(x_1, \ldots, x_n) \rangle$ for some x_i, and $c = x_j$ for some j. Then $expsup(op : \langle v \rangle) = expsup(v) = \bigcup_{i=1}^{n} expsup(x_i) \supseteq expsup(x_j) = expsup(c : \epsilon)$, so again $SUB(op : vs) \Rightarrow SUB(c : \epsilon)$.

- The Variable rule: suppose $SUB(x_i : vs)$ holds. Then $expsup(vs_i) \subseteq \bigcup_{i=1}^{n} expsup(vs_i) \subseteq expsup(x_i : vs) \subseteq subexps(P)$
- The Function rule: $s \Rightarrow e^{(f)} : \epsilon$. Certainly the RHS state is a subexpression of P
- CApp-1, CApp-2, CIf-1, CIf-T, CIf-F: $e : vs \rightarrow e' : vs$ where e' is a subexpression of e. Then

$$\begin{array}{rcl} expsup(e' : vs) & = & subexps(e') \cup \bigcup_{v \in vs} expsup(v) \\ & \subseteq & subexps(e) \cup \bigcup_{v \in vs} expsup(v) \subseteq subexps(P) \end{array}$$

- CApp-Call. Suppose that $e_1 : vs \Downarrow e_b : ws$ and $e_r : vs \Downarrow v$. Also, suppose that $SUB(e_1 e_2 : vs)$ holds. Then (as in the previous case) $SUB(e_1 : vs)$ and $SUB(e_2 : vs)$ hold. Hence by the inductive hypothesis $SUB(e_b : ws)$ and $SUB(v)$. Hence

$$\begin{array}{rcl} expsup(e_b : ws +\!\!+ \langle v \rangle) & = & expsup_{lhs}(e_b) \cup \bigcup_{w \in ws +\!\!+ \langle v \rangle} expsup(w) \\ & = & expsup_{lhs}(e_b) \cup \bigcup_{w \in ws} expsup(w) \cup expsup(v) \\ & = & expsup(e_b : ws) \cup expsup(v) \subseteq subexps(P) \end{array}$$

- The Step rule: trivial [inductive hypothesis]

□

A.2 The Abstract Interpretation (Section 5)

Lemma 20. Let s be a state, and suppose that $t \trianglelefteq \alpha_n(s)$. If $s = Err$ this is trivial. Otherwise we show that $Restrict_n(t) \trianglelefteq \alpha_{n-1}(s)$ for all n by induction on n.

If $n = 1$, then $Restrict_n(s) = \bullet = \alpha_0(s)$, as required.

For the inductive step, assume $n > 1$. The state s is of the form $e : \langle v_1, \ldots, v_k \rangle$ necessarily. Then $t = e' : \langle v'_1, \ldots, v'_k \rangle$ where $e' \trianglelefteq \alpha_{lhs}(e)$ and $v'_i \trianglelefteq \alpha_{n-1}(v_i)$ for each i. We have:

$$\begin{array}{rll} & Restrict_n(t) & \\ = & Restrict_n(e' : \langle v'_1, \ldots, v'_k \rangle) & \\ = & e' : \langle Restrict_{n-1}(v'_1), \ldots, Restrict_{n-1}(v'_k) \rangle & \\ \trianglelefteq & e' : \langle \alpha_{n-2}(v_1), \ldots, \alpha_{n-2}(v_2) \rangle & \text{[IH]} \\ \trianglelefteq & \alpha_{lhs}(e) : \langle \alpha_{n-2}(v_1), \ldots, \alpha_{n-2}(v_2) \rangle & \\ = & \alpha_{n-1}(e : \langle v_1, \ldots, v_k \rangle) & \end{array}$$

as required □

Soundness of the abstract interpretation

Measure of a computation We now give the proof of soundness of the abstract interpretation, starting with the base case (depth-1 interpretation). The proof is by induction on a measure of the size of a derivation, defined as follows:

$$\left\| \frac{\nabla_1 \quad \nabla_2 \quad \cdots \quad \nabla_n}{\cdots} \right\| = 1 + n + \sum_{i=1}^{n} \|\nabla_i\|$$

where we write ∇ for an arbitrary proof tree. Certainly $\|\nabla\|$ is well-defined and finite for any proof ∇. Given a sequence of calls (each justified by a proof tree)

$$s_o \xrightarrow{\nabla_0} s_1 \xrightarrow{\nabla_1} \cdots \xrightarrow{\nabla_{n-1}} s_n$$

we define its measure to be:

$$\left\| s_0 \xrightarrow{l_1} \cdots \xrightarrow{l_n} s_n \right\| = \sum_{i=0}^{n-1} \|\nabla_i\|$$

We will drop the judgement type labels l_i to avoid clutter whenever possible.

We require the following result, which gives a stronger induction principle than mere induction on the structure of proofs:

Lemma 24. Suppose that $\nabla : s_n \to s_{n+1}$ has subproofs $\nabla_1, \ldots, \nabla_k$ $(k > 0)$ and that ∇_i is a proof of $s \to s'$. Then

$$\left\| s_0 \to s_1 \to \cdots \to s_n \xrightarrow{\nabla} s_{n+1} \right\| > \left\| s_0 \to s_1 \to \cdots \to s_n \to s \xrightarrow{\nabla_i} s' \right\|$$

provided $s_n \to s$ either has an empty proof tree or has as proof tree one of the ∇_j $(j \neq i)$.

Proof. It suffices to show that

$$N_1 := \left\| s_n \xrightarrow{\nabla} s_{n+1} \right\| > \left\| s_n \to s \xrightarrow{\nabla_i} s' \right\| =: N_2$$

Now

$$N_1 = 1 + k + \sum_{j=1}^{k} \|\nabla_j\|$$

Also,

$$N_2 = \|s_n \to s\| + \left\| s \xrightarrow{\nabla_i} s' \right\| = \|\nabla_i\| + \|s_n \to s\|$$

Hence if $s_n \to s$ has an empty proof tree, then $N_2 = \|\nabla_i\| + 1 \leq N_1 - k < N_1$ as $k > 0$. Otherwise, $s_n \to s$ has as proof tree one of the ∇_j, whence $N_2 = \|\nabla_i\| + \|\nabla_j\|$. Then again $N_2 \leq N_1 - k < N_1$, as required. □

We can now proceed with the proof of soundness. Note that we do not elide the expression labels (distinguishing between two syntactic occurrences of the same expression) in the following.

Lemma 21. We fix a proof of $P_{start} \to^* s \xrightarrow{l} s'$, say

$$\Pi : P = s_0 \xrightarrow{\nabla_0} s_1 \xrightarrow{\nabla_1} s_2 \to \cdots \xrightarrow{\nabla_{n-1}} s_n = s \xrightarrow[l]{\nabla_n} s_{n+1} = s'$$

The proof is by induction on $\|\Pi\|$, and by cases on the last rule used in the proof of $s \xrightarrow{l} s'$.

We first record a few observations. Note that $\mathit{Reachable}_{AP}(s_0)$ holds trivially. Also, to avoid cluttering the proof in what follows, we only prove that the invariant *Reachable* is maintained when it is nontrivial.

The only rules that affect the environment in a nontrivial way are Variable and CApp-Call, and so the invariant that substates of reachable states are reachable must be checked for these rules. In addition, the primitive operator case for $\mathit{Reachable}_{AP}$ must be checked for the CApp-Call rule. Finally, the constant case for $\mathit{Reachable}_{AP}$ must be checked for the PrimOp rule. We will only need to consider preservation of the invariant for these rules.

- Value. Then $s' = s \in \mathit{Value}_{Err}$.

  If $s = \mathit{Err}$ the result is vacuous. Otherwise, there are three cases for s: a closure, a partially applied primitive operator or an atomic constant. Let $t \trianglelefteq \alpha_1(s)$, with $\mathit{Valid}_{AP}(t)$.

  If s is an atomic constant, say $s = c : \epsilon$, then $t = c' : 0$ where $c' \in \mathit{AbstConst}$, $c' \trianglelefteq \alpha_C(c)$. Hence $s \in \mathit{AbstValue}_{AP}$. If s is a partially applied primitive operator, say $s = op : vs$, then $t = op^\alpha : k$, where $k = |vs| < \sharp op = \sharp op^\alpha$, so that $s \in \mathit{AbstValue}_{AP}$ again.

  Finally, if s is a closure, then $s = e^l : vs$ for some expression e with label l, with $|vs| < \sharp e^l := \sharp l$. Then $t = e'^l : k$, where $k = |vs|$. Now as $\mathit{Valid}_{AP}(t)$ holds, and $k = |vs| < \sharp l = \sharp e'^l$, $t \in \mathit{AbstValue}_{AP}$.

  Hence in all cases $t \in \mathit{AbstValue}_{AP}$, so that by the abstract Value rule, $t \Downarrow_1 t, 1_t$. This is as required, as the graph 1_t is certainly safe for (s, s).

- Variable. Then $s = x_i^l : vs \Downarrow v_i = s'$.

  Suppose that $s \Downarrow s'$ by Variable. Then $\|s_0 \to \cdots s\| < \|s_0 \to \cdots s \to s'\|$, so that by [IH], $\mathit{Reachable}_{AP}(s)$ holds. Now $s = x_i : vs$, so that $f, i \multimap_{AP} w$ for some $w \trianglelefteq \alpha_1(v_i)$ and $\mathit{Valid}_{AP}(w)$ by definition of $\mathit{Reachable}_{AP}$, where f is the combinator in which the expression x_i^l appears. Also, $\mathit{Reachable}_{AP}(v_i)$ holds, again by definition, as required for maintaining the invariant (as $s' = v_i$).

  Let $t \trianglelefteq \alpha_1(s)$, with $\mathit{Valid}_{AP}(t)$. Then necessarily $t = x_i^l : k$, where $k = |vs| = \sharp x_i^l$. By the abstract Variable rule, $t \Downarrow_1 w_i$ for any w_i such that $t \ll_2 x_i : ws$ for some ws. By the $\mathsf{Promote}_{Var}$ rule, this occurs whenever $f, i \multimap_{AP} w_i$.

  In particular, by the above, as $f, i \multimap_{AP} w$, $t \Downarrow_1 w, 1^{i \mapsto \epsilon}$. Now $w \trianglelefteq \alpha_1(v_i)$ and $\mathit{Valid}_{AP}(w)$ by the above, as required.

  Finally, note that the graph $1^{i \mapsto \epsilon}$ is certainly safe for (s, s'). This completes the proof of this case.

- Function. Then $s = f^l : vs \Rightarrow e_P^{(f)} : \epsilon = s'$.

  Let $t \trianglelefteq \alpha_1(s)$ be such that $\mathit{Valid}_{AP}(t)$ holds. Then necessarily $t = f^l : k$, where $k = |vs| = \sharp l$. Hence by the abstract Function rule, $t \Rightarrow_1 e_{AP}^{(f)} : 0, \emptyset$. Certainly $t' = e_{AP}^{(f)} : 0$ satisfies $\mathit{Valid}_{AP}(t')$. Also, $e_{AP}^{(f)} \trianglelefteq e_P^{(f)}$ as $AP \trianglelefteq \alpha(P)$, whence $t' \trianglelefteq \alpha_1(s')$. Finally, the graph $\emptyset$ is safe for any reduction, as required.

- Constant. Then $s = \mathit{Cst}\ c : vs \Downarrow c : \epsilon$.

  Let $t \trianglelefteq s$. Then $t = \mathit{Cst}\ d : k$, where $d \in \mathit{AllConsts}$, $d \trianglelefteq c$. Hence by the abstract Constant rule, $t \Downarrow_1 \alpha_C(d) : 0, 1_{d:0}$.

  The graph $1_{d:0}$ is certainly safe for $(s, c : \epsilon)$. Also, $\alpha_C(d) : 0 \trianglelefteq \alpha_C(c) : 0 = \alpha_1(c : \epsilon)$ as $d \trianglelefteq c$, as required. Finally, $\mathit{Valid}_{AP}(\alpha_C(d) : 0)$ is vacuous.

- PrimOp. Then $s = op : vs \Downarrow op(vs) = s'$.

  Assume wlog that $s' \neq \mathit{Err}$. As $\mathit{Reachable}_{AP}(s)$, we have that $\mathit{Reachable}_{AP}(v)$ for each $v \in vs$. In addition, if $v \in vs$ is not a constant, then $\mathit{ESCAPES}(w)$ for some $w \trianglelefteq \alpha_1(v)$.

  If s' is an atomic constant, then reachability is trivial. Otherwise, op is either a constructor or a destructor.

  If op is a constructor, then $s' = C(vs) : \epsilon$ necessarily. Fix $v \in vs$. Then $\mathit{Reachable}_{AP}(v)$ and $\mathit{EscapeReach}_{AP}(v)$ as $\mathit{Reachable}_{AP}(s)$. This gives that $\mathit{Reachable}_{AP}(s')$, as required. If op is a destructor, then $vs = \langle v \rangle = \langle C(x_1, \ldots, x_k) : \epsilon \rangle$ necessarily and $s' = x_i$ for some i. As $\mathit{Reachable}_{AP}(s)$, $\mathit{Reachable}_{AP}(v)$. In particular, $\mathit{Reachable}_{AP}(x_i)$, as required.

  We now show that there is a corresponding reduction in the abstract semantics. Let $t \trianglelefteq \alpha_1(s)$. , then $t = op' : k$, where $op' \trianglelefteq \alpha_C(op)$ and $k = |vs| = \sharp op = \sharp op'$ necessarily. By the abstract PrimOp rule, $t \Downarrow_1 d : 0, \gamma$ where $d = \mathit{OpRes}_1(op')$ and $\gamma = \mathit{AbstOp}_1^{op}$.

  Suppose first that $c = op(vs)$ is a constant. Then as $* \trianglelefteq \alpha_C(v)$ for all $v \in vs$, $d := op'(*, \ldots, *) \trianglelefteq \alpha_C(d)$ (by AbstConst-Op). Hence $d : 0 \trianglelefteq \alpha_1(s')$, as required. The graph AbstOp_1^{op} is similarly safe for the reduction.

  Now suppose that $op(vs)$ is not a constant. Then as before $vs = \langle v \rangle = \langle C(x_1, \ldots, x_k) \rangle$ necessarily and $s' = x_i$ for some i. But then as $\mathit{Reachable}_{AP}(s)$, $\mathit{ESCAPES}(w)$ for some $w \trianglelefteq \alpha_1(x_i)$ and so by PrimOpEscape, $s \Downarrow_1 w, \mathit{AbstOp}_1^{op}$. This is as required. In particular, safety of the graph is ensured as op is a destructor and so $x_i \sqsubset s$.

  Now recall that $c = \mathit{OpRes}_1(op') := op'(*, \ldots, *)$ (with $\sharp op' = k$ copies of $*$. As $op' \trianglelefteq \alpha_C(op)$ and $* \trianglelefteq v_i$ for each i (trivially), by AbstOp-Consistency, we have $c \trianglelefteq op(vs)$, so that $c : 0 \trianglelefteq \alpha_1(op(vs) : \epsilon)$, as required.

  Also, $\gamma = \mathit{AbstOp}_1^{op'} := \mathit{AbstOp}^{op}(*, \ldots, *)$. Again, $op' \trianglelefteq \alpha_C(op)$ and $* \trianglelefteq v_i$ for each i, so by choice of $\mathit{AbstOp}^{op'}$, γ is safe for (s, s').

  Finally, $\mathit{Valid}_{AP}(c : 0)$ is vacuous, completing this case.

- PrimOpErr, AppErr1, AppErr2, IfErr. These cases are vacuous as $s' = \mathit{Err}$.

- CApp-1. Then $s = (e_1 e_2)^l : vs \to e_1^{l_1} : vs = s'$.

  Let t be such that $t \trianglelefteq \alpha_1(s)$ and $\mathit{Valid}_{AP}(t)$. Then $t = (e_1' e_2') : l$, where $k = |vs| = \sharp l$, and $e_1' \trianglelefteq e_1$, $e_2' \trianglelefteq e_2$. Hence by the abstract CApp-1 rule, $t \to_1 e_1' : k, 1_\downarrow$. Let $t' = e_1' : k$. As $\mathit{Valid}_{AP}(t)$ holds, certainly $\mathit{Valid}_{AP}(t')$ holds. Also, $t' \trianglelefteq \alpha_1(e_1 : vs) = \alpha_1(s')$. Also, the graph $1_\downarrow$ is certainly safe for (s, s'), as required.

- CApp-2. Then $s = (e_1^{l_1} e_2^{l_2})^l : vs \to e_2^{l_2} : vs = s'$. Let $s^1 = e_1^{l_1} : vs$. Then $s^1 \Downarrow e_b^m : ws$, where $|ws| < \sharp e_b^m$.

  As $s_1 \Downarrow e_b : ws^m$ appears as a subproof of $s \to s'$, we have

$$\|s_0 \to \cdots \to s \to s^1 \Downarrow e_b : ws\| < \|s_0 \to \cdots \to s \to s'\|$$

Let $t \trianglelefteq \alpha_1(s)$ with $Valid_{AP}(t)$. Then $t = (e_1'^{l_1} e_2'^{l_2})^l : k$, where $k = |vs| = \sharp l$ and $e_1' \trianglelefteq e_1$, $e_2' \trianglelefteq e_2$.

Let $t_1 = e_1'^{l_1} : k$. Then $t^1 \trianglelefteq \alpha_1(s^1)$. Also, $s^1 \neq Err$ (otherwise $s' = Err$). Finally, certainly $Valid_{AP}(t^1)$ holds. By the above, [IH] applies and so $t^1 \Downarrow v$ for some $v \trianglelefteq \alpha_1(e_b : ws)$, and $Valid_{AP}(v)$.

Hence $v = e_b'^m : k$, where $k < \sharp e_b'^m$. Hence by the abstract CApp-2 rule, $t \rightarrow_1 e_2'^{l_2} : k, 1_\downarrow$. This is as required, as $e_2' : k \trianglelefteq \alpha_1(e_2 : vs)$ certainly, and $1_\downarrow$ is safe for $(s, e_2 : vs)$. Finally, certainly $Valid_{AP}(e_2' : k)$ holds.

- CApp-Call. Then $s = (e_1^{l_1} e_2^{l_2})^l : vs \rightarrow e_b^m : ws +\!\!+ \langle v \rangle = s'$ where $e_1^{l_1} : vs \Downarrow e_b^m : ws$, $e_2^{l_2} : vs \Downarrow v$ and $|ws| < \sharp e_b^m$. Let $s^1 = e_1^{l_1} : vs$, $s^2 = e_2^{l_2} : vs$ and $s^b = e_b^m : ws$.

  Now $s^1 \Downarrow s^b$ and $s^2 \Downarrow v$ appear as subproofs of $s \Rightarrow s'$. Hence we have:

  $$\left\| s_0 \rightarrow \cdots s \rightarrow s^1 \Downarrow s^b \right\|, \left\| s_0 \rightarrow \cdots s \rightarrow s^2 \Downarrow v \right\| < \| s_0 \rightarrow \cdots s \Rightarrow s' \|$$

  Let $t \trianglelefteq \alpha_1(s)$, $Valid_{AP}(t)$. Then $t = (e_1'^{l_1} e_2'^{l_2})^l : k$, where $e_i' \trianglelefteq e_i$ $(i = 1, 2)$ and $k = |vs| < \sharp l$.

  Let $t^i = e_i'^{l_i} : k$ $(i = 1, 2)$. Then $t^i \trianglelefteq \alpha_1(s^i)$, certainly $Valid_{AP}(t^i)$ and $s^i \neq Err$. Hence [IH] $t^1 \Downarrow_1 t^b, \gamma_1$ and $t^2 \Downarrow_1 w, \gamma_2$ for some $t^b \trianglelefteq \alpha_1(s^b)$, $w \trianglelefteq \alpha_1(v)$, with $Valid_{AP}(t^b)$, $Valid_{AP}(w)$ and γ_1, γ_2 safe for (s^1, s^b), (s^2, v) (resp.).

  Now $t^b = e_b'^m : j$ where $e_b' \trianglelefteq e_b$, and $j = |ws|$. Hence $j < \sharp e_b'^m$. The abstract CApp-Call rule therefore applies. We thus have $t \Rightarrow_1 e_b' : (l+1), \gamma$, where $\gamma = [AppG_l(\gamma_1, \gamma_2)]_1$. Now $e_b' : (l+1) \trianglelefteq \alpha_1(e_b : ws +\!\!+ \langle v \rangle)$. Certainly $Valid_{AP}(e_b' : (l+1))$ holds. Also, $AppG_i(\gamma_1, \gamma_2)$ is safe for (s, s') as γ_1 is safe for (s^1, s^b) and γ_2 for (s^2, v). As restriction of graphs preserves safety, so is γ, as required.

  Finally, we must show that the invariant is maintained. Now note that $\left\| s_0 \rightarrow \cdots s \rightarrow s^2 \Downarrow v \right\| < \| s_0 \rightarrow \cdots s \Rightarrow s' \|$. Hence by [IH] the invariant holds for v, so that $Reachable_{AP}(v)$ holds. Also, as $\left\| s_0 \rightarrow \cdots s \rightarrow s^1 \Downarrow s^b \right\| < \| s_0 \rightarrow \cdots s \Rightarrow s' \|$, by [IH] we have $Reachable_{AP}(s^b)$.

  Now $s' = e_b : ws +\!\!+ \langle v \rangle$. As $Reachable_{AP}(s^b)$, in particular $Reachable_{AP}(w)$ for all $u \in ws$. Also, $Reachable_{AP}(v)$. Hence all values bound in the environment are reachable.

  There are two cases to consider. Suppose first that s^b is a primitive operator state $op : ws$. Then as $Reachable_{AP}(s^b)$, $EscapeReach_{AP}(u)$ for all $u \in ws$. If $v \in Constant$, then trivially $EscapeReach_{AP}(v)$ and we are done. Otherwise, $e_1 e_2 \in subexps(P)$ (since $Valid(s)$ holds), $e_1 : \sharp e_1 \Downarrow_1 op' : k$ for some $k < \sharp op'$, and $e_2 : \sharp e_2 \Downarrow_1 w$ for some $w \trianglelefteq \alpha_1(v)$ ([IH], see above). This shows that $ESCAPES(w)$, whence $EscapeReach_{AP}(v)$, as required.

  Finally, suppose that s^b is an expression state, and let $f = CombDef(e_b)$. We must show that $ClosureReach_{AP}(f, i, u_i)$ for all $u \in ws +\!\!+ \langle v \rangle$, where i is the position of u. For $u \in ws$ this is given by [IH]. Hence it suffices to show that $f, j+1 \multimap_{AP} w$ for some $w \trianglelefteq \alpha_1(v)$, and $Valid_{AP}(w)$ to complete the proof.

  Now we recall that $t = e_1' e_2' : k$, where $e_1' e_2' \in subexps(AP)$ as $Valid(t)$ holds. Also, $e_1' : k \Downarrow_1 e_b^m : j$, $j < \sharp e_b^m$. Hence as $e_2' : k \Downarrow_1 w$, this gives that $f, j+1 \multimap_{AP} w$. Now $w \trianglelefteq \alpha_1(v)$ and $Valid_{AP}(w)$ has been established above, completing the proof.

- CIf-1. Then $s = (e \rightarrow e_1 \mid e_2) : vs \rightarrow e : vs = s'$.

  This case is formally identical to CApp-1.

- CIf-T, CIf-F. We consider wlog the case CIf-T. Then $s = (e^m \to e_1^{m_1} \mid e_2^{m_2})^l : vs \Rightarrow e_1^{m_1} : vs = s'$. Let $s^1 = e^m : vs$, then $s^1 \Downarrow T : \epsilon$.

  Let $t \trianglelefteq \alpha_1(s)$ with $\mathit{Valid}_{AP}(t)$. Then $t = (e' \to e_1' \mid e_2')^l : k$, where $k = |vs| = \sharp l$. Also, as $s^1 \Downarrow T : \epsilon$ appears as a subproof of $s \Rightarrow e_1 : vs$, we have

  $$\left\|s_0 \to \cdots \to s \to s^1 \Downarrow T : \epsilon\right\| < \|s_0 \to \cdots \to s \Rightarrow s'\|$$

  Furthermore, $t^1 := e'^m : k \trianglelefteq \alpha_1(s^1)$ as $e' \trianglelefteq e$ necessarily and $s^1 \neq \mathit{Err}$. Also, certainly $\mathit{Valid}_{AP}(t^1)$ holds.

  Hence [IH] we have $t^1 \Downarrow_1 v$ for some $v \trianglelefteq \alpha_1(T : \epsilon) \trianglelefteq T : 0$. Hence by the abstract CIf-T rule we find $t \Rightarrow_1 e_1' : k, 1_\downarrow$. This is as required, as $e_1' : k \trianglelefteq \alpha_1(s')$, $\mathit{Valid}_{AP}(e_1' : k)$ certainly and $1_\downarrow$ is certainly safe for (s, s').

- Step. Then $s \Downarrow v$, where $s \Rightarrow s'$ and $s' \Downarrow v$, and as $v \neq \mathit{Err}$, $s' \neq \mathit{Err}$.

  Now $\|s_0 \to \cdots s \Rightarrow s'\| < \|s_0 \to \cdots s \Downarrow v\|$ (as the proof of $s \to s'$ appears as a subproof of $s \Downarrow v$).

  Let $t \trianglelefteq \alpha_1(s)$ be such that $\mathit{Valid}_{AP}(t)$ holds. Then by [IH] we find $t \Rightarrow_1 u, \gamma_1$ for some $u \trianglelefteq \alpha_1(s')$, $\mathit{Valid}_{AP}(u)$ and γ_1 safe for (s, s').

  Also, $\|s_0 \to \cdots s' \Downarrow v\| < \|s_0 \to \cdots s \Downarrow v\|$ as $s' \Downarrow v$ appears as a subproof of $s \Downarrow v$. As $u \trianglelefteq \alpha_1(s')$, by [IH] we find $u \Downarrow_1 w, \gamma_2$ for some $w \trianglelefteq \alpha_1(v)$, $\mathit{Valid}_{AP}(w)$ and γ_2 safe for (s', v).

  Hence by the abstract Step rule, $t \Downarrow_1 w, (\gamma_1; \gamma_2)$. This is as required, as $w \trianglelefteq \alpha_1(v)$, $\mathit{Valid}_{AP}(w)$ and $\gamma_1; \gamma_2$ is safe for (s, v) as γ_1 is safe for (s, s') and γ_2 is safe for (s', v).

□

Lemma 22. Let s be a P-state, and suppose that $\mathit{Reachable}_{AP}(s)$ holds. Let $t \trianglelefteq \alpha_n(s)$. Then t is necessarily a AP-abstract state. We show that $t \ll_{n+1} w$ for some $w \trianglelefteq \alpha_{n+1}(s)$, w an AP-abstract state, by induction on n.

Base Case $n = 1$. Let $t \trianglelefteq \alpha_1(s)$. We can write $s = \mathit{lhs} : vs$ for some $vs = \langle v_1, \ldots, v_k\rangle$. Then $t = \mathit{lhs}' : k$, where $\mathit{lhs}' \trianglelefteq \alpha_{\mathit{lhs}}(\mathit{lhs})$ and lhs' is either a constant or a subexpression of AP. Now $\mathit{Reachable}_{AP}(s)$ holds, so that for each i, $f, i \multimap_{AP} w_i$ for some $w_i \trianglelefteq \alpha_1(v_i)$ where $f = \mathit{CombDef}(\mathit{lhs})$ if lhs is an expression and $f = \mathit{lhs}$ if lhs is a constant. Also, each w_i is an AP-abstract state. Hence by rule Promote$_{base}$,

$$\mathit{lhs}' : k \ll_2 \mathit{lhs}' : \langle w_1, \ldots, w_k\rangle$$

Now

$$\begin{aligned} \mathit{lhs}' : \langle w_1, \ldots, w_k\rangle &\trianglelefteq \alpha_{\mathit{lhs}}(\mathit{lhs}) : \langle w_1, \ldots, w_k\rangle \\ &\trianglelefteq \alpha_{\mathit{lhs}}(\mathit{lhs}) : \langle \alpha_1(v_1), \ldots, \alpha_k(v_k)\rangle \\ &= \alpha_2(\mathit{lhs} : \langle v_1, \ldots, v_k\rangle) \end{aligned}$$

as required. Also, certainly $\mathit{lhs}' : ws$ is an AP-abstract state as each w_i is an AP-abstract state, completing this case.

Inductive Step Suppose that this holds for $n-1$. We can write $s = \mathit{lhs} : \mathit{vs}$ as above, and $t = \mathit{lhs}' : \mathit{ws}$ where $\mathit{lhs}' \trianglelefteq \alpha_{\mathit{lhs}}(\mathit{lhs})$. Furthermore, $w_i \trianglelefteq \alpha_{n-1}(v_i)$ for each i.

Furthermore, as $\mathit{Reachable}(s)$, $\mathit{Reachable}(v_i)$ for each i. As $w_i \trianglelefteq \alpha_{n-1}(v_i)$, by the inductive hypothesis,

$$(\forall i) \quad w_i \ll_n u_i \text{ for some } u_i \trianglelefteq \alpha_n(v_i)$$

By rule $\mathsf{Promote}_{n+1}$,

$$\mathit{lhs}' : \langle w_1, \ldots, w_k \rangle \ll_{n+1} \mathit{lhs}' : \langle u_1, \ldots, u_k \rangle$$

Now

$$\begin{array}{rcl} \mathit{lhs}' : \langle u_1, \ldots, u_k \rangle & \trianglelefteq & \alpha_{\mathit{lhs}}(\mathit{lhs}) : \langle u_1, \ldots, u_k \rangle \\ & \trianglelefteq & \alpha_{\mathit{lhs}}(\mathit{lhs}) : \langle \alpha_n(v_1), \ldots, \alpha_n(v_k) \rangle \\ & = & \alpha_{n+1}(\mathit{lhs}) : \langle v_1, \ldots, v_k \rangle \end{array}$$

as required. Again, $\mathit{lhs}' : \mathit{us}$ is an AP-abstract state, completing the proof. □

Lemma 23. Suppose that $P_{start} {\to_P}^* s \xrightarrow{l}_P s'$, $s' \neq \mathit{Err}$. We must show that whenever $t \trianglelefteq \alpha_n(s)$ and t is an AP-abstract state, then $t \xrightarrow{l}_{APn} t', \gamma$ for some $t' \trianglelefteq \alpha_n(s')$, with t' an abstract AP-state and γ safe for (s, s').

The proof is as before by induction on the measure of the proof of $P_{start} \to^* s \to s'$, and by cases on the last rule used to show $s \xrightarrow{l} s'$. Most of the cases are entirely similar to the depth-1 case treated previously. The only significant differences occur in the Variable, PrimOp and CApp-Call rules. We may assume $n > 1$ in the following.

- Variable. Then $s = x_i^l : \mathit{vs} \Downarrow v_i = s'$.

  Let $t \trianglelefteq \alpha_n(s)$, with $\mathit{Valid}_{AP}(t)$. As $\mathit{Reachable}_{AP}(s)$ holds, since $P_{start} \to^* s$) (by Lemma 21), we can apply Lemma 22. Hence there exists $w \trianglelefteq \alpha_{n+1}(s)$ with $t \ll_{n+1} w$, and $\mathit{Valid}_{AP}(w)$. Now $w = x_i^l : \mathit{us}$ for some us necessarily, where $u_j \trianglelefteq \alpha_n(v_j)$ for each j. Also, as $\mathit{Valid}_{AP}(w)$ holds, $\mathit{Valid}_{AP}(u_j)$ for each j. In particular, $u_i \trianglelefteq \alpha_n(v_i)$ and $\mathit{Valid}_{AP}(u_i)$. By the abstract Constant rule, $t \Downarrow_n u_i, 1^{i \mapsto \epsilon}$. The graph is trivially safe, and so this is as required.

- PrimOp. Then $s = \mathit{op} : \mathit{vs} \Downarrow \mathit{op}(\mathit{vs}) = s'$. Let $t \trianglelefteq \alpha_n(s)$, then $t = \mathit{op}' : \mathit{avs}$ where $\mathit{av}_i \trianglelefteq \alpha_n(v_i)$ for each i, and $\mathit{op}' \trianglelefteq \alpha(\mathit{op})$.

  Suppose first that s' is not a constant. Then op is necessarily a destructor, and $\mathit{vs} = \langle v \rangle = \langle C(x_1, \ldots, x_k) : \epsilon \rangle$. Furthermore, $s' = x_i$ for some i. As $\mathit{Reachable}_{AP}(s)$, in particular $\mathit{Reachable}_{AP}(v)$ and so $\mathit{EscapeReach}_{AP}(x_i)$. Hence $\mathit{ESCAPES}_{AP}(x)$ for some $x \trianglelefteq \alpha_1(x_i)$. Thus by the PrimOpEscape rule, $t \Downarrow_n y, \mathit{AbstOp}_1^{\mathit{op}'}$ for any y such that $x \ll_n^* y$. By the corollary to Lemma 22, there exists such a y with $y \trianglelefteq \alpha_n(x_i)$ and $\mathit{Valid}_{AP}(y)$, as required. Safety of the graph is as before.

  Now assume that s' is a constant state. We may assume that $s' \neq \mathit{Err}$. Recall that $t = \mathit{op}' : \mathit{avs}$ where $\mathit{op}' \trianglelefteq \alpha_C(\mathit{op})$ and $\mathit{av}_i \trianglelefteq \alpha_n(v_i)$ for each i.

  By the abstract PrimOp rule, $t \Downarrow_n r : \epsilon, \gamma$, where $r = \mathit{op}'(\mathit{avs})$ and $\gamma = \mathit{AbstOp}^{\mathit{op}}(\mathit{avs})$. As before, as $\mathit{op}' \trianglelefteq \alpha_C(\mathit{op})$ and $\mathit{av}_i \trianglelefteq \alpha_n(v_i)$ for each i, by AbstOp-Consistency we have $r \trianglelefteq \alpha_C(\mathit{op}(\mathit{vs}))$, as required. Also, γ is safe for (s, s') by choice of $\mathit{AbstOp}^{\mathit{op}'}$. Finally, $\mathit{Valid}_{AP}(r : \epsilon)$ is trivial, completing this case.

- CApp-Call. Then $s = e_1 e_2 : vs \Rightarrow e_b : ws +\!\!+ \langle v \rangle = s'$, where $s^1 := e_1 : vs \Downarrow e_b : ws$ and $s^2 := e_2 : vs \Downarrow v$, $|{:}|\ ws < \sharp e_b$.

  Let $t \trianglelefteq \alpha_n(s)$, $Valid_{AP}(t)$. Then $t = e_1' e_2' : us$, where $e_1', e_2' \trianglelefteq e_1, e_2$ (resp.) and $u_i \trianglelefteq \alpha_{n-1}(v_i)$ for each i.

  Let $t^1 = e_1' : us$, $t^2 = e_2' : us$. Then $Valid_{AP}(t^1)$ and $Valid_{AP}(t^2)$ and as before, the [IH] applies to show that $t^1 \Downarrow_n t^b, \gamma_1$, $t^2 \Downarrow_n z, \gamma_2$ where $t^b \trianglelefteq e_b : ws$, $z \trianglelefteq \alpha_n(v)$, $Valid_{AP}(t^b)$, $Valid_{AP}(z)$ and γ_1, γ_2 are safe for the respective reductions.

  Now $t^b = e_b' : ys$, where $e_b' \trianglelefteq e_b$, and $y_i \trianglelefteq \alpha_{n-1}(w_i)$ for each i. Also, $|ys| = |ws| < \sharp e_b = \sharp e_b'$. Hence by the abstract CApp-Call rule,

  $$e_1' e_2' : us \Rightarrow_n e_b' : ys +\!\!+ \langle Restrict_n(z) \rangle, [AppG_i(\gamma_1, \gamma_2)]_n$$

  Safety of the graph is as before, it therefore suffices to show that

  $$t' := e_b' : ys +\!\!+ \langle Restrict_n(z) \rangle \trianglelefteq s' = e_b : ws +\!\!+ \langle v \rangle$$

  Now $e_b' \trianglelefteq e_b$ and $y_i \trianglelefteq \alpha_{n-1}(w_i)$ for each i by the above. It therefore suffices to show that $Restrict_n(z) \trianglelefteq \alpha_{n-1}(v)$ to complete the proof. But we know that $z \trianglelefteq \alpha_n(v)$, and therefore Lemma 20 does the trick.

  Finally, $Valid_{AP}(t')$ holds as e_b' is a subexpression of AP, $Valid_{AP}(y_i)$ holds for each i and $Valid_{AP}(z)$ holds by induction. Trivially this implies that $Valid_{AP}(Restrict_n(z))$ holds, as required.

□

B Precise Definitions and Algorithms

The purpose of this appendix is to contain material (precise definitions, semantics and/or algorithms) that do not fit in the body of the text but are useful as a reference for future work.

B.1 Abstract Interpretation

Figures 6 (basic rules) and 7 (closure and escape analysis) give the precise definition of the abstract interpretation.

B.2 Abstract Constants

We have specified our analysis in terms of arbitrary sets of constants and abstract constants, provided only that they satisfy certain assumptions. We now proceed to describe a possible choice of these, corresponding to the current implementation of the SCT analysis. This is intended both to show that the properties we have assumed are satisfied by some sensible choice of set of constants, and to give a precise description of the implementation.

B.2.1 The Set of Constants

Atoms An input program to the analysis can contain a number of (possibly mutually recursive) algebraic datatype definitions. Of course, we are working in an untyped setting, and so the purpose of those is merely to introduce constructor names and arities. Together, these define a

Value

$$\frac{}{s \Downarrow_n s, 1_s}\ {\scriptstyle s \in AbstValue}$$

Step

$$\frac{s \Rightarrow s', \gamma_1 \qquad s' \Downarrow v, \gamma_2}{s \Downarrow v, (\gamma_1 ; \gamma_2)}$$

Constant

$$\frac{}{Cst\ c : vs \Downarrow_n c^{\alpha} : \epsilon, 1_{c:\epsilon}}\ {\scriptstyle c \in Constant}$$

Function

$$\frac{}{f : vs \Rightarrow_n e^{(f)} : \epsilon, \emptyset}$$

Variable

$$\frac{x_i : vs \lll_{n+1} x_i : ws}{x_i : vs \Downarrow_n ws_i, 1^{i \mapsto \epsilon}}\ {\scriptstyle n>1}$$

PrimOp

$$\frac{}{op : vs \Downarrow_n op(vs), AbstOp^{op}(vs)}\ {\scriptstyle n=\sharp op}$$

PrimOpEscape

$$\frac{ESCAPES(v) \qquad v \lll_n^* w}{op : vs \Downarrow_1 w, AbstOp_1^{op}(n)}\ {\scriptstyle \begin{array}{l}|vs|=\sharp op\\ op \text{ is a destructor}\end{array}}$$

CApp-1

$$\frac{}{e_1 e_2 : vs \rightarrowtail_n e_1 : vs, 1_{\downarrow}}$$

CApp-2

$$\frac{e_1 : vs \Downarrow_n e_b : ws, \gamma_1}{e_1 e_2 : vs \rightarrowtail_n e_2 : vs, 1_{\downarrow}}\ {\scriptstyle |ws|<\sharp e_b}$$

CApp-Call

$$\frac{e_1 : vs \Downarrow_n e_b : ws, \gamma_1 \qquad e_2 : vs \Downarrow_n v, \gamma_2}{e_1 e_2 : vs \Rightarrow_n e_b : ws +\!\!+ \langle Restrict_n\ v \rangle, [AppG_i(\gamma_1, \gamma_2)]_n}\ {\scriptstyle i=|ws|<\sharp e_b}$$

CIf-1

$$\frac{}{(e \to e_1 \mid e_2) : vs \rightarrowtail_n e : vs, 1_{\downarrow}}$$

CIf-T

$$\frac{e : vs \Downarrow_n v : \epsilon, \gamma \qquad v \trianglelefteq T^{\alpha}}{(e \to e_1 \mid e_2) : vs \Rightarrow_n e_1 : vs, 1_{\downarrow}}$$

CIf-F

$$\frac{e : vs \Downarrow_n v : \epsilon, \gamma \qquad v \trianglelefteq F^{\alpha}}{(e \to e_1 \mid e_2) : vs \Rightarrow e_2 : vs, 1_{\downarrow}}$$

Figure 6: Abstract Interpretation, General Case (1)

$$\textsf{Promote}_n \quad \frac{(\forall i) \quad v_i \ll_{n-1} w_i}{e : \langle v_1, \ldots, v_k \rangle \ll_n e : \langle w_i, \ldots, w_k \rangle}\; n>2 \qquad \textsf{Promote}_\epsilon \quad \frac{}{\mathit{lhs} : \epsilon \ll_n \mathit{lhs} : \epsilon}$$

$$\textsf{Promote}_{Op} \quad \frac{(\forall i) \quad op, i \multimap c_i}{op : k \ll_2 op : cs} \qquad \textsf{Promote}_{Var} \quad \frac{(\forall i) \quad f, i \multimap v_i}{e : k \ll_2 e : vs}\; f = \mathit{CombDef}(e)$$

$$\textsf{Variable-Approx} \quad \frac{e_1 e_2 \in \mathit{subexps}(P) \qquad e_1 : \sharp e_1 \Downarrow_1 f : i-1 \qquad e_2 : \sharp e_2 \Downarrow_1 v}{f, i \multimap v}$$

$$\textsf{Escape-Analysis} \quad \frac{e_1 e_2 \in \mathit{subexps}(P) \quad e_1 : \sharp e_1 \Downarrow_1 op' : k \quad e_2 : \sharp e_2 \Downarrow_1 v}{\mathit{ESCAPES}(v)}\; \begin{array}{l} op' \text{ is a constructor} \\ k < \sharp op' \end{array}$$

Figure 7: Abstract Interpretation, General Case (2)

set *AlgConst* of values that can be constructed using these constructors. These values, and the operations on them, are discussed in Section 2.

The set of atoms is then defined to be:

$$Atom = \mathbb{B} \cup \mathbb{N}$$

where again the union is assumed disjoint. The set $\mathbb{B}$ is the set of boolean values, while the set $\mathbb{N}$ is the set of natural numbers.

We must now define a well-founded order on *Atom*. This is defined as the disjoint union of the following orders:

- The trivial order on $\mathbb{B}$
- The usual order on $\mathbb{N}$

Each of these is trivially well-founded, whence so is their union.

Primitive Operators Finally, we define the primitive operators on atoms. We include the following operators on natural numbers and booleans, with the usual meaning:

$$\mathit{BoolOps} = \{\wedge, \vee, \neg\} \text{ and } \mathit{NatOps} = \{+, *, -, \mathbf{div}, \mathbf{mod}, =, \neq, >, <, \geq, \leq\}$$

This yields the set of primitive operators:

$$\mathit{PrimOp} = \mathit{BoolOps} \cup \mathit{NatOps}$$

It is clear that these satisfy the requirements of Section 2.

B.2.2 Abstract Constants

Abstract Atomic Constants We now need to define the set *AbstBasic* say of abstract basic constants (*i.e.* not primitive operators). This should be finite. There are of course many ways to do this, and we describe just one possibility here. The most interesting aspect of it is the set of abstract constants corresponding to natural numbers. We define this to be:

$$AbstNat = \{nat, nat_{>0}, nat_{>1}\} \cup \{nat_{=i} \mid 0 \leq i \leq N\}$$

for some N. The informal meaning of each of these is:

- *nat* represents any natural number
- $nat_{>0}$, $nat_{>1}$ represent any natural number $n > 0$ (resp. $n > 1$). This will be useful for finding size changes as shown below
- $nat_{=i}$ represents the natural number i. This is useful for the translation of tuples, as the i^{th} coordinate of a tuple v is selected by a call get i p. In this case, for precision we would like to know precisely what the value of i is. This will be the only use of these values, and the upper bound N should therefore be chosen to be the size of the largest tuple appearing in the program (known statically).

The information order on *AbstNat* can be chosen so that:

- $nat \trianglelefteq x$ for all $x \in AbstNat$
- $nat_{>0} \trianglelefteq nat_{>1}$

This is not the most precise order possible (for example, we could add $nat_{>0} \trianglelefteq nat_{=1}$), but it will suffice for our purposes.

The set of abstract boolean constants is easier to define, as the original set is finite:

$$AbstBool = \{bool, T, F\}$$

with order $bool \trianglelefteq T, F$. The value *bool* represents an unknown boolean.

Note that these orders make *AbstNat* and *AbstBool* into join semilattices.

Finally, we choose the set of abstract algebraic constants to contain precisely one member for each type defined in the program:

$$AbstAlg = \{t \mid t \text{ is an algebraic datatype}\}$$

For example, this would contain one constant *list* to represent all lists. This is finite, and we define the order on it to be the trivial order (all values are incomparable).

The complete set of abstract atoms can then be defined to be:

$$AbstBasic = AbstNat \cup AbstBool \cup AbstAlg \cup \{*\}$$

where the union is disjoint, and $*$ represents *any* constant. The order on this set is the disjoint union of the individual orders, with in addition $* \trianglelefteq x$ for all x. This is trivially a finite join semilattice, as required.

The Abstraction Function The abstraction function is a map $\alpha : \mathit{BasicConst} \to \mathit{AbstBasic}$. This is easily defined so that as much precision as possible is obtained. It is just the identity on booleans (as $\mathbb{B} \subseteq \mathit{AbstBool}$), and can be defined as follows for algebraic datatypes:

$$\alpha(c) = t \text{ where } t \text{ is the type of } c$$

Finally, for natural numbers, we define:

$$\alpha(n) = \begin{cases} \mathit{nat}_{>1} & \text{if } n > 1 \\ \mathit{nat}_{>0} & \text{if } n = 1 \\ \mathit{nat} & \text{if } n = 0 \end{cases}$$

This is slightly counter-intuitive, as it never produces the values $\mathit{nat}_{=i}$. However, recall that our analysis allows us to prove termination of a program with abstract constants occurring inside expressions (such a program is not executable, however). This is how the abstract constants $\mathit{nat}_{=i}$ are introduced by the implementation, only in the context of tuple selection functions (recall that we do not wish to use these abstract constants in other contexts). The precise operations takes the expression $\mathsf{get}\ i$ to the expression $\mathsf{get}\ \mathit{nat}_{=i}$. There are of course other ways in which these values could be used, but this is the only case in which the increase in the size of the state space seemed justified.

Abstract Operators We must now define, for each primitive operator op, an abstract primitive operator op^{α}. The requirement on this is that:

$$(\forall i.d_i \trianglelefteq \alpha(c_i)) \Rightarrow op^{\alpha}(d_1, \ldots, d_n) \trianglelefteq \alpha(op(c_1, \ldots, c_n)) \vee op(c_1, \ldots, c_n) = \mathit{Err}$$

where $n = \sharp op$.

- Algebraic Datatypes

We deal with algebraic datatypes first, as these are easier. There are three operations that we must consider: a constructor $\mathsf{C}_{\mathsf{i}}^{\mathsf{t}}$ constructing a value of type t, a destructor $\mathsf{getC}_{\mathsf{i}}^{\mathsf{t}}.\mathsf{j}$ selecting the j^{th} component of a value constructed using C_i^t, and finally a test $\mathsf{isC}_{\mathsf{i}}^{\mathsf{t}}$ to test whether a value of type t has C_i as its top-level constructor. The best we can do, given the approximation that we have chosen is to define:

- ${\mathsf{C}_{\mathsf{i}}^{\mathsf{t}}}^{\alpha}(v_1, \ldots, v_n) = t$ for all $(v_1, \ldots, v_n)$ $(n = \sharp \mathsf{C}_{\mathsf{i}}^{\mathsf{t}})$
- ${\mathsf{getC}_{\mathsf{i}}^{\mathsf{t}}.\mathsf{j}}^{\alpha}(v) = *$ for all v
- ${\mathsf{isC}_{\mathsf{i}}^{\mathsf{t}}}^{\alpha}(v) = \mathit{bool}$ for all v

Note that this does not deal with error cases, but this is not required for correctness. These clearly satisfy the above axiom.

- Natural Numbers and Booleans

We do not give exact definitions for all the operators on natural numbers and booleans, but merely highlight the interesting points. First note that the values $\mathit{nat}_{=i}$ are only ever meant to be tested for equality, and hence we do not need to use the added precision in any of the operators apart from $=$. We could define $=$ to be:

$$(v_1 =^{\alpha} v_2) = \begin{cases} i = j & \text{if } v_1 = \mathit{nat}_{=i} \text{ and } v_2 = \mathit{nat}_{=j} \\ \mathit{bool} & \text{otherwise} \end{cases}$$

This is precise enough for our purposes. The boolean operators are defined to make maximal use of the information available. For example, conjunction can be defined thus:

$$(v_1 \wedge^\alpha v_2) = \begin{cases} F & \text{if } v_1 = F \vee v_2 = F \\ T & \text{if } v_1 = T \wedge v_2 = T \\ bool & \text{otherwise} \end{cases}$$

Finally, we should make sure that the information $nat_{>0}$ (resp. $nat_{>1}$) is propagated and generated as much as possible, as this will turn out to be useful in generating size change graphs. For example, addition could be defined as:

$$(v_1 +^\alpha v_2) = \begin{cases} nat_{>1} & \text{if} \quad v_1 = nat_{>1} \vee v_2 = nat_{>1} \\ & \qquad \vee (v_1 = nat_{>0} \wedge v_2 = nat_{>0}) \\ nat_{>0} & \text{if } v_1 = nat_{>0} \vee v_2 = nat_{>0} \\ nat & \text{otherwise} \end{cases}$$

Again, it is trivial to check that these satisfy the axiom above. The other primitive operators are left to the reader's imagination.

B.2.3 Size-Change Graphs

The last aspect of the abstract interpretation of constants that we need to describe is the generation of size-change graphs. Recall that these are introduced *via* the construction

$$AbstOp^{op}(v_1, \ldots, v_n)$$

where the v_i are abstract values, and $n = \sharp op$. The required property of this is:

If $v_i \trianglelefteq \alpha(w_i)$ for each i, then $AbstOp^{op}(v_1, \ldots, v_n)$ is safe for the reduction

$$op : \langle w_1, \ldots, w_n \rangle \Downarrow op(w_1, \ldots, w_n) : \epsilon$$

Now the graph basis on the left-hand side is $\{\epsilon\} \cup \{i \mid 1 \leq i \leq n\}$, while the graph basis on the right-hand side is $\{\epsilon\}$. All arcs in the produced graph will go from a node i on the left-hand side to ϵ on the right-hand side (there is no notion of size change for an arc from ϵ to ϵ). We write $i > res$ to mean that there is an arc labelled $\downarrow$ from i and $i \geq res$ to indicate that there is an arc labelled $=$ from i. These statements are enough to describe the graphs.

The easiest operators to deal with are destructors — these *always* have arity 1 and decrease the size of their argument (except in an error case, which we do not need to deal with). We therefore have

$$AbstOp^{\mathsf{getC_i.j}}(v) : \qquad v > res$$

Constructors and tests are trivial, as no size-change information can be deduced in either case. This is also the case of most of the operators on natural numbers and booleans. The only exceptions are $-$, **div** and **mod**. The only problem is that these are not uniform in their size change. In particular,

- $a - b$ is less than a, *provided* $b > 0$
- a **div** b is less than b, *provided* $b > 1$.

We can finally reveal this as the reason for introducing the values $nat_{>0}$ and $nat_{>1}$. These allow us to produce reasonably precise size-change information for these operators, as shown below. Subtraction can be defined as:

$$AbstOp^{-}(v_1, v_2): \qquad \begin{cases} v_1 > res & \text{if } nat_{>0} \trianglelefteq v_2 \\ v_1 \geq res & \text{otherwise} \end{cases}$$

Division is similar:

$$AbstOp^{\mathbf{div}}(v_1, v_2): \qquad \begin{cases} v_1 > res & \text{if } nat_{>1} \trianglelefteq v_2 \\ v_1 \geq res & \text{otherwise} \end{cases}$$

Finally, **mod** is simpler (although one might find it hard to think of an example in which this size change information could ever prove termination)

$$AbstOp^{\mathbf{mod}}(v_1, v_2): \qquad v_2 > res$$

Note that in both the **div** and **mod** cases, the fact that we do not have to deal with error cases here makes our life much simpler!

This concludes our discussion of the abstract interpretation for constants as currently implemented. This should give the reader a good idea of the issues that arise in using these ideas in practice, as well as the many ways in which this could be extended to increase precision if needed.

B.3 An Example Program Evaluation

We give an example of the semantics of our language in action, giving the proof tree for the evaluation of a small example program in full detail.

The program

$$\begin{array}{l} f\ x = x + x \\ f\ \mathsf{2} \end{array}$$

To evaluate this program, we must evaluate the expression $f\ \mathsf{2}$ in the empty environment, in the context of the definition of the function f.

A few points to note:

- We have written $x + x$ above as a shorthand for the expression $\mathsf{plus}\ x\ x$. In the proof trees below we use the prefix notation for consistency with the semantics
- Recall that states are defined in terms of actual *occurrences* of expressions (each subexpression of the program is assumed to be uniquely identified by some label). However, this example is small enough that it is not necessary to make the distinction between the form of an expression and an occurrence of it. For example, when we write $\mathsf{plus}\ x\ x$ in a state, we really mean the right-hand side of the definition of f. The only ambiguity is with the expression x (two occurrences), and we label these x^1 and x^2
- The program includes the subexpressions plus and $\mathsf{2}$. These are *syntactic constants.* The *semantic values* of these are the mathematical objects + (a function) and 2 (a natural number).

We now give the proof tree for the reduction of $f\ \mathsf{2} : \epsilon$, shown in several parts for space reasons.

Proof tree ∇_1 for $f\ 2 : \epsilon \Rightarrow e^{(f)} : \langle 2 : \epsilon \rangle$

$$\dfrac{\dfrac{\dfrac{\text{function}}{f : \epsilon \Rightarrow \mathsf{plus}\ x^1\ x^2 : \epsilon} \quad \dfrac{\text{value}}{\mathsf{plus}\ x^1\ x^2 : \epsilon \Downarrow \mathsf{plus}\ x^1\ x^2 : \epsilon}}{f : \epsilon \Downarrow \mathsf{plus}\ x^1\ x^2 : \epsilon}\ \mathsf{step} \qquad \dfrac{\mathsf{constant}}{2 : \epsilon \Downarrow 2 : \epsilon}}{f\ 2 : \epsilon \Rightarrow \mathsf{plus}\ x^1\ x^1 : \langle 2 : \epsilon \rangle}\ \mathsf{call}$$

Proof tree ∇_2 for $\mathsf{plus}\ x^1\ x^2 : \langle 2 : \epsilon \rangle \Rightarrow + : \langle 2 : \epsilon, 2 : \epsilon \rangle$

$$\dfrac{\dfrac{\dfrac{\mathsf{constant}}{\mathsf{plus} : \langle 2 : \epsilon \rangle \Downarrow + : \epsilon} \quad \dfrac{\mathsf{variable}}{x^1 : \langle 2 : \epsilon \rangle \Downarrow 2 : \epsilon}}{\mathsf{plus}\ x^1 : \langle 2 : \epsilon \rangle \Downarrow + : \langle 2 : \epsilon \rangle}\ \mathsf{call} \qquad \dfrac{\mathsf{variable}}{x^2 : \langle 2 : \epsilon \rangle \Downarrow 2 : \epsilon}}{\mathsf{plus}\ x^1\ x^2 : \langle 2 : \epsilon \rangle \Rightarrow + : \langle 2 : \epsilon, 2 : \epsilon \rangle}$$

Proof tree ∇_3 for $\mathsf{plus}\ x^1\ x^2 : \langle 2 : \epsilon \rangle \Downarrow 4 : \epsilon$

$$\dfrac{\dfrac{\nabla_2}{\mathsf{plus}\ x^1\ x^2 : \langle 2 : \epsilon \rangle \Rightarrow + : \langle 2 : \epsilon, 2 : \epsilon \rangle} \quad \dfrac{}{+ : \langle 2 : \epsilon, 2 : \epsilon \rangle \Downarrow 4 : \epsilon}\ \mathsf{primop}}{\mathsf{plus}\ x^1\ x^2 : \langle 2 : \epsilon \rangle \Downarrow 4 : \epsilon}\ \mathsf{step}$$

Putting it all together

$$\dfrac{\dfrac{\nabla_1}{f\ 2 : \epsilon \Rightarrow \mathsf{plus}\ x^1\ x^2 : \langle 2 : \epsilon \rangle} \quad \dfrac{\nabla_3}{\mathsf{plus}\ x^1\ x^2 : \langle 2 : \epsilon \rangle \Downarrow 4 : \epsilon}}{f\ 2 : \epsilon \Downarrow 4 : \epsilon}\ \mathsf{step}$$

Notes A few notes about the preceding proof trees:

- The rule value states that $v \Downarrow v$ whenever v is a value. In the above, it is used with v a *closure* (recall that all closures are values), in the first derivation
- In the second derivation, in the subtree leading to the conclusion $\mathsf{plus}\ x^1 : \langle 2 : \epsilon \rangle \Downarrow + : \langle 2 : \epsilon \rangle$, we are actually using both the call and step rules, but the step rule has been elided for space reasons. Note also that the environments on the left-hand and right-hand sides of the $\Downarrow$ are not related, but are just accidentally identical
- The function rule is used to conclude $f : \langle 2 : \epsilon \rangle \Rightarrow e^{(f)} : \epsilon$, where $e^{(f)} = \mathsf{plus}\ x\ x$ is the body of x. Now $e^{(f)} : \epsilon$ is a closure ($e^{(f)}$ expects one argument) and thus reduces to itself. Hence using step, $f : \langle 2 : \epsilon \rangle \Downarrow e^{(f)} : \epsilon$. A function name, evaluated in any environment, thus reduces to the closure corresponding to that function with no arguments. The only case in which this might not be a final state (and thus the reason why function is a *call* rule, rather than a *reduce* rule) is the case of a "function" with no arguments (known as a constant applicative form). In this case, the state $e^{(f)} : \epsilon$ is not a closure, and thus needs to be evaluated further.
- The constant rule applied to plus (in the second derivation) is similar to the function rule.

- The primop rule evaluates an operator when all its arguments are known. Thus $+ : \langle 2 : \epsilon, 2 : \epsilon \rangle \Downarrow 4$ by primop. Note that this could yield *Err*, say when evaluating $+ : \langle \mathit{True} : \epsilon, 2 : \epsilon \rangle$ or $hd : \langle [] : \epsilon \rangle$. This is taken as part of the definition of $+$ as a partial function from pairs of constants to constants.

C Translating OCAML into the Core Language

In this appendix, we give the precise details of the translation from our subset of the OCAML language into the core language. This is intended to be an illustrative example, and a similar translation could be given from any similar ML-like purely functional language.

C.1 The OCAML Language

We first detail some salient features of the OCAML language that we must deal with.

C.1.1 Patterns

An OCAML pattern is one of:

1. _, pronounced "any". This matches any value
2. c where c is a constant. This matches exactly c.
3. x for a variable x. This matches any value and binds x to the value matched
4. $C\ p$ for a constructor C and a pattern p. This matches a value of the form $C\ x$, where x matches p, and binds any variables as in p
5. $(p_1, \dots, p_n)$ for patterns p_i. This matches any n-tuple $(x_1, \dots, x_n)$ where x_i matches p_i, and binds variables as appropriate
6. $p_1 \mid p_2$. This matches whenever at least of p_1 and p_2 matches. Both patterns must bind exactly the same variables (with the same types). The bindings are those of p_1 if it matches, otherwise those of p_2.

An OCAML pattern-matching expression has the form:

$$\begin{array}{ll} \multicolumn{2}{l}{\textbf{match } e \textbf{ with}} \\ & p_1\ [\textbf{when } g_1] \to e_1 \\ \mid & p_2\ [\textbf{when } g_2] \to e_2 \\ \vdots & \\ \mid & p_n\ [\textbf{when } g_n] \to e_n \end{array}$$

where each **when** guard g_i is optional. The (informal) semantics of this expression is: to evaluate the above,

- Evaluate the expression e, yielding a value v (or nontermination)
- Match v against each of the p_i, starting at p_1, until some pattern matches (this is guaranteed to terminate) or none match. If none match, return an error value
- If p_i is the first matching pattern, then there are two possibilities:

- If p_i has no guard g_i, then evaluate e_i in an environment including the bindings of p_i. The result of this is the value of the pattern-matching expression (or nontermination)
- If p_i has a guard g_i, evaluate g_i in the environment enriched with the bindings of p_i. If this does not terminate the expression does not terminate. If g_i evaluates to *true*, proceed as above. If g_i evaluates to *false*, resume matching at p_{i+1}. Otherwise, return an error value.

This is the primitive pattern-matching construct in OCAML. However, it is not the only way to introduce pattern matching in a program. For, local and top-level function definitions can also introduce pattern matching. We discuss this below.

C.1.2 λ-Abstractions

OCAML programs can include λ-abstractions, introduced using two constructs, the **fun** and **function** constructs.

A **function** expressions has the form **function** *patmatch*, where *patmatch* is the body of a **match** expression as defined above. The intended semantics of this is:

$$\textbf{function}\ \mathit{patmatch} \qquad \equiv \qquad \lambda x.\textbf{match}\ x\ \textbf{with}\ \mathit{patmatch}$$

where x is a fresh variable name. There is a subtle point here, which we return to after discussing the **fun** constructed.

A **fun** expression has the form

$$\textbf{fun}\ p_1\ p_2\ \cdots\ p_n \to e$$

where each p_i is a pattern and e is an expression. This is defined to be equivalent to the expression

$$\textbf{function}\ p_1 \to \textbf{function}\ p_2 \to \cdots \to \textbf{function}\ p_n \to e$$

as expected. We therefore do not need to deal explicitly with the **fun** construct.

However, the OCAML manual specifies that in a function application

$$(\textbf{fun}\ p_1\ \cdots\ p_n \to b)\ e_1\ \cdots\ e_n$$

the values v_i of the arguments e_i are matched *in parallel*. That is, all the arguments are evaluated first, and the values are then matched. This is an important point, as the patterns p_i could be refutable, and the e_i nonterminating. In particular, e_2 will be evaluated even if the value v_1 of e_1 does not match the pattern p_1. If e_2 is nonterminating, this will cause the expression above to fail to terminate. However, by the translation we have given above this should be syntactic sugar for

$$(\textbf{function}\ x \to \textbf{match}\ x\ \textbf{with}\ p_1 \to (\textbf{fun}\ p_2\ \cdots\ p_n \to b))\ e_1\ \cdots\ e_n$$

This clearly has the wrong behaviour, in particular it fails immediately if v_1 does not match p_1. We are therefore forced to assume that expressions of the form

$$\textbf{function}\ p_1 \to \textbf{function}\ p_2 \to \cdots \to \textbf{function}\ p_n \to e$$

are a special case, which should be translated differently. It is unclear whether this is intended as an actual feature of OCAML, and whether it is in agreement with the semantics of standard

ML. Experiments with both the OCAML interpreter and the Moscow ML interpreter with the function definition

$$\mathbf{let}\ f\ (x :: xs)\ ys = xs\ @\ ys$$

yield the following intriguing result (where Ω stands for any nonterminating expression, we chose $\mathbf{let\ rec}\ g\ () = g\ ()\ \mathbf{in}\ g\ ()$):

	OCAML	Moscow ML
$f\ []$	Function value	*Match Failed* exception
$f\ []\ \Omega$	Nontermination	Nontermination

We did not attempt to examine this issue in any more detail, as it is unrelated to the problem at hand, but thought it worth noting.

C.1.3 Local Definitions

There are two forms of local declarations in OCAML, namely **let** and **rec** expressions.

The syntax of **let** and **let rec** expression is identical (except for the **let** keyword) and has the form:

$$\begin{array}{lllll}
\mathbf{let}\ [\mathbf{rec}] & p_1\ [q^1_1\ \cdots\ q^1_{n_1}] & = & e_1 \\
\mathbf{and} & p_2\ [q^2_1\ \cdots\ q^2_{n_2}] & = & e_2 \\
\vdots & & & \\
\mathbf{and} & p_k\ [q^k_1\ \cdots\ q^k_{n_k}] & = & e_k \\
\mathbf{in}\ e & & &
\end{array}$$

where $n_i \geq 0$ for each i. First note that

$$\mathbf{let}\ p_i\ q_1\ q_2\ \cdots\ q_n = e \qquad \equiv \qquad \mathbf{let}\ p_i = \mathbf{fun}\ q_1\ \cdots\ q_n \to e$$

whenever $n > 0$ (this is just syntactic sugar) and so we may assume $n_i = 0$ for all i.

The difference between **let** and **let rec** is that in a **let rec** expression, the right-hand side expressions are evaluated in a context that includes bindings for all the newly defined identifiers (on the left-hand side). However, in order for this to make sense under call-by-value, this restricts the form of definitions in **let rec** definitions. In particular, it must be possible to evaluate the defined values *before* they are defined. We therefore restrict **let rec** definitions to *function* definitions of the form

$$\mathbf{let\ rec}\ f_1 = \mathbf{function}\ p_1 \to e_1\ \mathbf{and}\ \cdots\ \mathbf{and}\ f_k = \mathbf{function}\ p_k \to e_k$$

This is equivalent to assuming $n_i > 0$ for all i in the above. This restriction is consistent with the OCAML semantics — although more general uses of **let rec** are allowed, their behaviour is assumed to be implementation-dependent.

C.2 The Transformation

We will now describe how the features defined above can be translated into the core language. We can separate this into several phases, which we give below. The last two phases (compiling pattern matching and λ-lifting) are the only nontrivial operations. We have endeavoured to ensure that the translations, as well as being functionally correct and termination preserving, are complexity-preserving and do not increase code size beyond reason.

C.2.1 Currying Constructors

There is a technical issue that we must deal with before describing the translation. Constructors in OCAML are *uncurried* functions, so that one might write $(::)\ (x, xs)$ for the list with head x and tail xs. However, it is more natural in our core language to express constructors as curried functions. We therefore need to translate all applications of a constructor to an n-tuple into curried application of said constructor to n variables.

Now note that each constructor C can only appear in one datatype, and has a fixed *arity* $n \geq 0$. Then if $n > 0$, any occurrence of C in a valid OCAML program must be applied to some expression e. Now if e is *not* an n-tuple, we apply the rewrite

$$C\ e \qquad \mapsto \qquad \textbf{let}\ (x_1, \ldots, x_n) = e\ \textbf{in}\ C\ (x_1, \ldots, x_n)$$

This is clearly behaviour-preserving, even in ill-typed programs (resulting in a runtime type error if e is not an n-tuple). The result of this is that each constructor C either has arity 0 or is applied to an n-tuple, where C has arity n. We can then curry each constructor application using the following rewrite rule:

$$C\ (x_1, \ldots, x_n) \qquad \mapsto C\ x_1\ \cdots\ x_n$$

The only other occurrences of constructors are inside pattern matching expressions. We must uncurry those as well — although this is not valid OCAML syntax, we need to do this to distinguish "proper" tuple patterns from tuple patterns. Suppose that the pattern $C\ p$ occurs inside a pattern. There are several cases for p:

- $p = C'\ q$ for some constructor C'. Then the arity of C is necessarily 1, so that the pattern need not be changed
- $p = (p_1, \ldots, p_n)$ for some patterns p_i. Then the arity of C is n, and we must curry this to $C\ p_1\ \cdots\ p_n$.
- $p = p_1 \mid p_2$. We need not deal with this case explicitly, as $C\ p \equiv (C\ p_1) \mid (C\ p_2)$ in this case
- $p = x$ for some identifier x. Then the arity of C is nonzero. If the arity of C is 1, then we can leave this pattern as is. However, if the arity of C is $n > 1$, then this binds x to the tuple of subcomponents of C. To curry all constructor applications, this must be changed to $C\ x_1\ \cdots\ x_n$ for some fresh variables x_i. Any occurrence of x must be replaced by the tuple $(x_1, \ldots, x_n)$.
- $p = \_$. Then this can be replaced with $C\ \_\ \cdots \_$, where the number of occurrences of $\_$ is the arity of C.

Finally, at this stage the only tuples left in the program do not arise from constructor applications. We replace each tuple expression with an explicit function call, which we will define later:

$$(e_1, \ldots, e_n) \qquad \mapsto \qquad \mathsf{tuple}_n\ e_1\ \cdots\ e_n$$

C.2.2 Syntactic Sugar

The next step abstracts away some of the subtleties of the OCAML syntax, to reduce the number of cases that may have to deal with. First, eliminate all **fun** expressions using the rule

$$\textbf{fun}\ p_1\ \cdots\ p_n \to e \qquad \mapsto \qquad \textbf{function}\ p_1 \to \cdots \to \textbf{function}\ p_n \to e$$

Now, to deal with the issue mentioned above of nested **function** definitions and curried functions, we introduce "proper" λ-abstractions using the following rule:

$$\textbf{function}\ p_1 \to \cdots \to \textbf{function}\ p_n \to e \qquad \mapsto \qquad \lambda p_1\ \cdots\ p_n.e$$

$$\textit{provided}\ e \neq \textbf{function}\ q \to e'$$

The next step is that we wish to move pattern matching entirely inside explicit **match** statements. The translation of patterns found inside λ abstractions is easy:

$$\lambda p_1\ p_2\ \cdots\ p_n.e \qquad \mapsto \qquad \begin{array}{l} \lambda x_1\ x_2\ \cdots\ x_n. \\ \quad \textbf{match}\ x_1\ \textbf{with}\ p_1 \to \\ \quad \textbf{match}\ x_2\ \textbf{with}\ p_2 \to \\ \quad \cdots \\ \quad \textbf{match}\ x_n\ \textbf{with}\ p_n \to \\ \quad\quad e \end{array}$$

where the x_i are fresh variables.

To eliminate pattern matching from **let** expressions, we must add definitions for all the free variables of the pattern. For example

$$\textbf{let}\ (x, y) = e$$

becomes

$$\textbf{let}\ z = e\ \textbf{and}\ x = \textit{fst}\ z\ \textbf{and}\ y = \textit{snd}\ z$$

where z is a fresh variable. We thus add one declaration for each free variable, defined to be the appropriate selectors applied to the original value. We will describe these in details when we come to compiling pattern matching. Note that **let rec** declarations cannot contain patterns, as we have stipulated that only function values can be defined.

Finally, at this stage we apply the following rewrite rule:

$$\textbf{match}\ e\ \textbf{with}\ p \qquad \mapsto \qquad (\lambda x.\textbf{match}\ x\ \textbf{with}\ p)e$$

where x is a fresh variable, whenever the expression e is not a variable or a constant. This ensures that even if the expression to be matched is duplicated in the compiled code, this does not lead to unbounded code bloating, and does not affect complexity.

C.2.3 Eliminating let Expressions

We now apply two simple transformations to eliminate nonrecursive **let** declarations entirely. First, turn multiple **let** declarations into nested declarations:

$$\begin{array}{llll} \textbf{let} & x_1 & = & e_1 \\ \textbf{and} & x_2 & = & e_2 \\ \cdots & & & \\ \textbf{and} & x_n & = & e_n \\ \textbf{in} & e & & \end{array} \qquad \mapsto \qquad \begin{array}{llll} \textbf{let} & x_1 & = & e_1\ \textbf{in} \\ \textbf{let} & x_2 & = & e_2\ \textbf{in} \\ \cdots & & & \\ \textbf{let} & x_n & = & e_n\ \textbf{in} \\ e & & & \end{array}$$

Second, apply the rewrite rule:

$$\textbf{let}\ x = e_x\ \textbf{in}\ \textit{body} \qquad \mapsto \qquad (\lambda x.\textit{body})e_x$$

This is clearly correct and termination preserving under a strict semantics. We only need to do it when the defined value x is *not* defined as a λ-abstraction. For, if e_x is not an abstraction then evaluation of it may not terminate, and thus we must force evaluation of e_x even if x is not used inside *body*. It is however pointless to do this if e_x *is* an abstraction. In this case, we may wlog replace **let** by **let rec**, which is dealt with later (in λ-lifting).

C.2.4 Compiling Pattern Matching

At this stage, the transformed program contains only three features that still need to be translated: simple λ-abstractions of the form $\lambda x_1 \cdots x_n.e$ for some variables x_i, **let rec** declarations with no patterns, and pattern matching expressions of the form **match** x **with** p, where x is either a variable or a constant. We now describe how pattern matching can be compiled into more primitive notions. Recall that we have already performed currying of constructors.

We first define the *free variables* of a pattern:

$$\begin{array}{rcl} FV(\_) & = & \emptyset \\ FV(c) & = & \emptyset \\ FV(x) & = & \{x\} \\ FV(C\ p_1\ \cdots\ p_n) & = & \bigcup_{i=1}^{n} FV(p_i) \\ FV(p_1, \ldots, p_n) & = & \bigcup_{i=1}^{n} FV(p_i) \\ FV(p_1 \mid p_2) & = & FV(p_1) \cup FV(p_2) \end{array}$$

Our intention in what follows is to compile pattern matching by:

1. Replacing the test for matching by a boolean guard
2. Replacing variable bindings by expressions to select the appropriate component from the original value.

Recall again that we are matching either variables or constants, and thus we are free to duplicate the matched expression without affecting complexity.

The first step is to eliminate the disjunction operator $p_1 \mid p_2$ from patterns. We use the following *distributivity* rules:

$$C\ (p_1 \mid p_2) \equiv (C\ p_1) \mid (C\ p_2)$$

and

$$(p_1, \ldots, p_i \mid q_i, \ldots, p_n) \equiv (p_1, \ldots, p_i, \ldots, p_n) \mid (p_1, \ldots, q_i, \ldots, p_n)$$

Also, note that $\mid$ is *associative*, so that there is no ambiguity in writing a pattern with all uses of disjunctions lifted to the top level as $q_1 \mid q_2 \mid \cdots \mid q_n$, where none of the q_i contain $\mid$. We rather abusively call this disjunctive normal form (DNF). Note that this includes the case $n = 1$ (no disjunction).

Using the above, the pattern matching expression becomes a list of cases in DNF, of the form

$$(q_1 \mid q_2 \mid \cdots \mid q_n)\ [\textbf{when}\ g] \to e$$

It is trivial that this is equivalent to the list of cases:

$$\begin{array}{l} \quad q_1 \ [\textbf{when } g] \to e \\ \mid q_2 \ [\textbf{when } g] \to e \\ \vdots \\ \mid q_n \ [\textbf{when } g] \to e \end{array}$$

when we have succeeded in eliminating disjunction from individual cases. As patterns in OCAML are restricted to being *linear* (no more than one use of any variable within a conjunction), this guarantees that any $x \in FV(q_i)$ occurs *exactly once* in q_i.

Variable Bindings This ensures that the *selection path* of a free variable, as defined below, is well-defined:

$$\begin{array}{rcl} \mathit{SelPath}\ x\ x & = & \epsilon \\ \mathit{SelPath}\ x\ (C\ p_1\ \cdots\ p_n) & = & C_i :: \mathit{SelPath}\ x\ p_i \\ & & \qquad\qquad \text{if } x \text{ occurs in } p_i \\ \mathit{SelPath}\ x\ (p_1, \ldots, p_n) & = & T_i^n :: \mathit{SelPath}\ x\ p_i \\ & & \qquad\qquad \text{if } x \text{ occurs in } p_i \end{array}$$

Note that provided $x \in FV(p)$, no other cases can arise in computing $\mathit{SelPath}\ x\ p$.

We can now use this selector path to define expressions that extract the required values out of the matched value. For any selector path q and core language expression e, we define an expression $\mathit{Sel}\ e\ q$ that evaluates to the component of the value of e selected by q. We use the basic functions on algebraic datatypes defined in Section 2 (to which we add a function for selecting a component of a tuple):

1. $\mathsf{getC}.i$ selects the i^{th} component of a value built using constructor C
2. isC determines whether a value has C as its top-level constructor, returning a boolean
3. get_i^n selects the i^{th} component of an n-tuple.

We can now define the selection function:

$$\begin{array}{rcl} \mathit{Sel}\ e\ \epsilon & = & e \\ \mathit{Sel}\ e\ (C_i :: p) & = & \mathit{Sel}\ (\mathsf{getC}.i\ e)\ p \\ \mathit{Sel}\ e\ (T_i^n :: p) & = & \mathit{Sel}\ (\mathsf{get}_i^n\ e)\ p \end{array}$$

It is trivial that if matching of e succeeds against the pattern that gave rise to q, the $\mathit{Sel}\ e\ q$ evaluates to the value bound to the variable that q points to in normal matching. We can hence eliminate all matching variables from the right-hand side expressions of a pattern matching case:

$$p \textbf{ when } g \to e \quad \mapsto \quad \begin{array}{c} p \textbf{ when } g[\mathit{Sel}\ m\ p(x)/x \mid x \in FV(p)] \to \\ e[\mathit{Sel}\ m\ p(x)/x \mid x \in FV(p)] \end{array}$$

where m is the expression matched, and $p(x)$ is the path to x in pattern p. Substitution is defined as usual to deal with the case of redefined variables properly. Finally, the **when** part is as usual optional

Matching We have now succeeded in eliminating variables bound in the pattern from the right-hand side parts of pattern matching expressions. We now show how each pattern p can be turned into an expression $\mathit{Test}\ e\ p$ that should evaluate to *true* if the value of e matches p, and *false* otherwise. We define:

$$
\begin{array}{rcl}
\mathit{Test}\ e\ \_ & = & \mathit{true} \\
\mathit{Test}\ e\ x & = & \mathit{true} \\
\mathit{Test}\ e\ c & = & e = c \\
\mathit{Test}\ e\ (C\ p_1\ \cdots\ p_n) & = & \mathsf{isC}\ e \wedge \bigwedge_{i=1}^{n} \mathit{Test}\ (\mathsf{getC}.i\ e)\ p_i \\
\mathit{Test}\ e\ (p_1, \ldots, p_n) & = & \bigwedge_{i=1}^{n} \mathit{Test}\ (\mathsf{get}_i^n\ e)\ p_i
\end{array}
$$

Note that in the tuple case, there is no need to check whether e is an n-tuple, as matching a value that is not a tuple against a tuple pattern will result in an error value, as in the above. However, for the constructor case, for this to be valid we must take the main conjunction to use short-circuit evaluation (to avoid creating spurious error values). This can be implemented as an **if** construct.

The guard for a branch of a pattern matching expression is then (recall that we have already performed the substitution to eliminate matching variables):

$$\mathit{Guard}\ e\ (p\ \textbf{when}\ g \to b) = \mathit{Test}\ e\ p \wedge g$$

where again we must use short-circuit evaluation for $\wedge$ (as g could be nonterminating and so must only be evaluated if p matches). If g is absent we can take simply $g = \mathit{true}$.

The translation of a pattern-matching expression can now be completed:

$$
\begin{array}{l}
\textbf{match}\ e\ \textbf{with}\ \langle p_i\ \textbf{when}\ g_i \to e_i \rangle_{i=1}^{n} \\
\qquad \mapsto \\
\textbf{if}\ \mathit{Guard}\ e\ (p_1\ \textbf{when}\ g_1 \to e_1)\ \textbf{then}\ e_1\ \textbf{else} \\
\textbf{if}\ \mathit{Guard}\ e\ (p_2\ \textbf{when}\ g_2 \to e_2)\ \textbf{then}\ e_2\ \textbf{else} \\
\cdots \\
\textbf{if}\ \mathit{Guard}\ e\ (p_n\ \textbf{when}\ g_n \to e_n)\ \textbf{then}\ e_n\ \textbf{else} \\
\textbf{fail}
\end{array}
$$

where **fail** is an expression that, when evaluated, terminates with an error value. This is of course not a particularly efficient way of doing things, but, as we are only concerned with termination preservation, functional correctness and complexity preservation, this is not a problem.

C.2.5 λ-Lifting

At this stage, the only features that we must eliminate are λ-abstractions and **let rec** definitions of functions. We can deal with both cases at once by the transformation

$$\lambda x_1 \cdots x_n.e \qquad \mapsto \qquad \textbf{let rec}\ f = \lambda x_1 \cdots x_n.e\ \textbf{in}\ f$$

Note that the **let** definition introduced is in fact nonrecursive, but we use **let rec** throughout for simplicity.

The crucial observation at this stage is, *assuming* that all variable and function names in the program are distinct (which we can guarantee by α-conversion), a local definition can be lifted to the top level under certain conditions.

For, consider an expression

$$E = \textbf{let rec } \left\langle f_i = \lambda x_1^i \cdots x_{n_i}^i \to e_i \right\rangle_{i=1}^{n} \textbf{ in } e$$

where

$$FV(e_i) \subseteq \{x_1^i, \ldots, x_{n_i}^i\} \cup \{f_1, \ldots, f_n\} \tag{1}$$

The expression $FV(e_i)$ denotes the free variables of e_i. Note that the names of top-level functions are not considered free variables (they are constants). Then (as each e_i is a λ-abstraction, it is terminating) this is equivalent to the expression e evaluated in a context where the f_i have been added as top-level definitions.

This process allows a **let rec** construct to be removed from the program, therefore iterating this we can remove all **let rec** expressions from the program, *provided* Equation 1 can be made to hold for all such definitions.

It therefore suffices to introduce a sound method for transforming a **let rec** expression into one that satisfies 1. This is known as *parameter lifting* — we will add parameters to each function definition until all free variables are accounted for.

Let Σ be the set of variables in scope at evaluation of the expression E (the entire **let rec** construct above). Then we have certainly

$$FV(e_i) \subseteq \Sigma \cup \{x_1^i, \ldots, x_{n_i}^i\} \cup \{f_1, \ldots, f_n\} := \Pi$$

We must eliminate free variables in $\Pi \setminus \Sigma$. This is achieved by β-expansion: to eliminate a variable z from f_i,

1. Replace the definition $f_i = \lambda x_1^i \cdots x_{n_i}^i \to e_i$ with $f_i = \lambda z x_1^i \cdots x_{n_i}^i \to e_i$
2. Replace each occurrence of f_i with $f_i\ z$.

We must show that this is valid, *i.e.* behaviour and termination-preserving. It is certainly the case that after performing this transformation z is not free in e_i.

Note that the expression f_i can only occur within one of the e_j, or within e. In both cases, the variable z is in scope at the occurrence of f_i (whenever $z \in \Pi$) and has the same value as its value within the body of f_i. Also, note that the expressions f_i and $f_i\ z$ are both terminating, and by the above evaluate to the expression

$$\lambda x_1^i \cdots x_{n_i}^i \to e_i$$

in the same environment, as z is bound to the same value in both cases. This implies that the operation of adding one parameter is behaviour and termination-preserving.

By induction, we may add parameters in the above fashion until no variables are free. Note that this must terminate, as adding all variables in Π as parameters guarantees that each e_i is closed.

This completes our description of λ-lifting.

C.2.6 Tuples

Finally, to complete our description of the translation of OCAML into our core language we must define the pairing functions explicitly. Recall that we have translated all tuples and selections of values of tuples into uses of the two functions tuple_n (to construct an n-tuple) and get_i^n (to select the i^{th} value from an n-tuple). Of course, tuples could be treated as an algebraic datatype, but we use a different encoding to increase precision in the common case of tuples of functions. This translation could also be applied to any algebraic datatype (at least in our untyped setting) and thus also serves as an illustrative example.

The idea is to represent the tuple $(x_1, \ldots, x_n)$ by a map selecting any of the components:

$$(x_1, \ldots, x_n) \text{ is represented by } \lambda i. x_i$$

The selection function is easy to define:

$$\mathsf{get}_i^n \; p = p \; i$$

In fact, for simplicity it is preferable to use a general function (to avoid having to define multiple copies):

$$\mathsf{get} \; i \; p = p \; i$$

Then $\mathsf{get}_i^n = \mathsf{get} \; i$.

It now remains to define the tupling function:

$$\begin{array}{rcl} \mathsf{tuple}_n \; x_1 \; x_2 \; \cdots \; x_n \; i & = & \textbf{if } i = 1 \textbf{ then } x_1 \textbf{ else} \\ & & \textbf{if } i = 2 \textbf{ then } x_2 \textbf{ else} \\ & & \vdots \\ & & \textbf{if } i = n-1 \textbf{ then } x_{n-1} \textbf{ else } x_n \end{array}$$

Note that this is correct provided i is never given a value outside $\{1, \ldots, n\}$, which is guaranteed by the implementation (the tuple_n function is not accessible to user programs and is only used internally).

Note however that this can introduce nontermination in ill-typed programs. For, consider the program

$$(1, 2) \; \Omega$$

where Ω is any nonterminating expression. Then this is ill-typed in ML, but under (a simple extension of) our semantics, in which the operator is evaluated before the argument, this should evaluate to *Err*, as a tuple cannot be applied. However, after the above translation the tuple $(1, 2)$ is represented by the value $\mathsf{tuple}_2 : \langle 1, 2 \rangle$. Thus the above becomes a valid application, which forces evaluation of Ω. However, this is never invalid as it only introduces *non*termination. Furthermore this only occurs in ill-typed programs, so that the slight loss of precision is acceptable.

There are a few points to be made here:

- The same idea could be extended to any algebraic datatype (a tuple is just a special case of an algebraic datatype) quite trivially. It is thus possible to increase precision in programs that use function values in datatypes.

- For precision, it is desirable that the expression get i p only reduce to the values to which the i^{th} component of i reduces. We therefore use a special kind of constant to represent the i in the above that is *not* abstracted away by the analysis. This is safe (the state space is still finite) as there are only finitely many of these appearing in the program, and no operations are ever possible on such values (only tests for equality), whence no new values can be generated.
- In a depth-n analysis, a function stored in a tuple will be stored as a depth-$(n-1)$ value. There is therefore a certain loss of precision when function values are stored in tuples.

C.3 The Implementation

An experimental implementation of the analysis described in these notes has been developed in OCAML. This is intended to serve in determining the practical boundaries of the method by testing it out on simple examples, as well as to serve as a first proof of concept. However, it should be noted that we have aimed for simplicity rather than efficiency. The current implementation suffices to handle simple programs, analysed to depth 2, but depth 3 analyses of nontrivial programs are intractable with it. This may be intrinsic to the problem (or at least to the algorithm), but it may also be a consequence of the slow implementation.

The language accepted by the current implementation is essentially the purely functional subset of OCAML, as described above. The translation given in this section is performed automatically, and the resulting program analysed. The output of the program includes all the self-loops found in the control flow of the program, in particular those self-loops (if any) that have not been proved terminating (potential counterexamples).

D Partial Completeness Results

In this section, we give some proofs of *partial completeness* of the higher-order SCT analysis. Recall that *soundness* of the SCT is just the statement that whenever a program is size-change terminating, then it does actually terminate. Thus soundness guarantees that when a positive answer is given, it is correct.

However, we have not established so far that the analysis ever does give a positive answer. This is the purpose of this section. We shall attempt to prove partial converses to soundness. By the undecidability of the halting problem, we can of course only aim for partial converses. The general form of the results that we seek to obtain is:

> Whenever a program P is proved terminating by analysis X, P is also size-change terminating.

We first give a few general results about the SCT analysis, then proceed with the partial completeness results.

D.1 General Results

In order to make the subsequent proofs easier, we need to obtain a few results about the higher-order SCT analysis.

D.1.1 Cycles and Termination

Lemma 25. Suppose that $s_0 \to s_1 \to \cdots \to s_{k-1} \to s_k = s_0$, where all the s_i are states in P. Then each s_i is of the form $e_i : n_i$, where $n_i = \sharp e_i$ and $e_i \in \mathit{subexps}(P)$. Also, $s_j = e^{(f)} : n_j$ for some j and some function f.

Proof. Suppose for a contradiction that for some s_i, $s_i \neq e_i : n_i$. Let $s_i = \mathit{lhs} : k$. Then either $k < \sharp\mathit{lhs}$, or lhs is a constant. In the first case, the only rule that applies is Value, whence $s_i \not\to s_{i+1}$, a contradiction. In the second case, the only possible rules are Value and PrimOp, whence again $s_i \not\to s_{i+1}$ and we have a contradiction.

Now suppose for a contradiction again that no s_i is a function body state. Then the only possible rules used to prove $s_i \to s_{i+1}$ for each i are CApp-1, CApp-2, CIf-1, CIf-T and CIf-F. For, the Function and CApp-Call rule only apply with a function body state as right-hand side.

However, whenever $s_i \to s_{i+1}$ by one of the above rules, e_{i+1} is a proper subexpression of e_i. By transitivity this therefore yields that $e_0 = e_n$ is a proper subexpression of e_0, a contradiction as required. $\square$

D.1.2 An Order on Size-Change Graphs

We define an order relation on size-change graphs. This is written $\subseteq$ for want of a better symbol, but is slightly different to the subset order — for, we want to include the fact that an arc $x \xrightarrow{=} y$ gives less information than an arc $x \xrightarrow{\downarrow} y$.

The idea of the $\subseteq$ order on graphs is that whenever $\gamma \subseteq \delta$, then γ is safe whenever δ is. We define this precisely below.

Definition 46. We define an order $<$ on $\{=, \downarrow\}$ by $= < \downarrow$.

Two trivial results:

Lemma 26. If $l_1 \leq l_1'$ and $l_2 \leq l_2'$, then $l_1 l_2 \leq l_1' l_2'$.

Proof. If one of l_1', l_2' equals $\downarrow$, then $l' = l_1' l_2' = \downarrow$ and so this is immediate. Otherwise, l_1' and l_2' both equal $=$, whence so do l_1 and l_2. Therefore $l = l_1 l_2$ is equal to $=$, as is l', so $l = l'$. $\square$

Lemma 27. If $x \xrightarrow{l} y \in S$, then $x \xrightarrow{l'} y \in \mathit{Simplify}(S)$ for some $l' \geq l$.

Proof. Recall that $\mathit{Simplify}(S)$ is obtained from S by removing any edge $a \xrightarrow{=} b$ whenever there exists an edge $a \xrightarrow{\downarrow} b$ in S.

Hence if $l = \downarrow$, then the arrow cannot have been removed, so take $l' = l$. Otherwise, the only problematic case occurs if $x \xrightarrow{=} y$ has been removed. This can only occur if $x \xrightarrow{\downarrow} y \in S$, in which case $x \xrightarrow{\downarrow} y \in \mathit{Simplify}(S)$. Hence take $l' = \downarrow$. $\square$

We can now define the order on size-change graphs:

Definition 47. Let γ, γ' be size-change graphs. Then $\gamma \subseteq \gamma'$ iff the source (resp. target) node sets of γ and γ' are equal, and:

$$x \xrightarrow[l]{\gamma} y \qquad \Rightarrow \qquad x \xrightarrow[l']{\gamma'} y \qquad \text{for some } l' \geq l$$

We wish to show that the two main operations on graphs are monotonic under this operation:

Lemma 28. Suppose that $\gamma_1 \subseteq \delta_1$ and $\gamma_2 \subseteq \delta_2$. Then $\gamma_1 ; \gamma_2 \subseteq \delta_1 ; \delta_2$.

Proof. Certainly the source and target node sets match as expected. Let $\gamma = \gamma_1 ; \gamma_2$ and $\delta = \delta_1 ; \delta_2$. Let S_1 and S_2 be the respective sets of nodes of γ_1, γ_2 and T_1, T_2 be those of δ_1, δ_2.

Suppose that $x \xrightarrow[l]{\gamma} z$. Then $x \xrightarrow{l} z \in \mathit{Simplify}(S_1 S_2) \subseteq S_1 S_2$. Thus $x \xrightarrow[l_1]{\gamma_1} y \xrightarrow[l_2]{\gamma_2} z$ for some y and $l_1 l_2 = l$. As $\gamma_i \subseteq \delta_i$, $x \xrightarrow[l'_1]{\delta_1} y \xrightarrow[l'_2]{\delta_2} z$, where $l'_i \geq l_i$. Hence $l' = l'_1 l'_2 \geq l_1 l_2 = l$.

Now $x \xrightarrow{l'} z \in T_1 T_2$. Hence $x \xrightarrow{l''} z \in \mathit{Simplify}(T_1 T_2)$ for some $l'' \geq l' \geq l$, as required since this implies that $x \xrightarrow[l'']{\delta} z$. □

Lemma 29. Suppose that $\gamma_1 \subseteq \delta_1$ and $\gamma_2 \subseteq \delta_2$. Then $\gamma_1 \cup \gamma_2 \subseteq \delta_1 \cup \delta_2$.

Proof. Let $\gamma = \gamma_1 \cup \gamma_2$ and $\delta = \delta_1 \cup \delta_2$ and S_i, T_i as above. Suppose that $x \xrightarrow[l]{\gamma} y$. Then $x \xrightarrow{l} y \in \mathit{Simplify}(S_1 \cup S_2) \subseteq S_1 \cup S_2$. Hence wlog $x \xrightarrow{l} y \in S_1$. Thus $x \xrightarrow{l'} y \in T_1 \subseteq T_1 \cup T_2$ for some $l' \geq l$. Hence $x \xrightarrow{l''} y \in \mathit{Simplify}(T_1 \cup T_2)$ for some $l'' \geq l'$. Hence $x \xrightarrow[l'']{\delta_1 \cup \delta_2} y$, as required as $l'' \geq l' \geq l$. □

D.2 The λ-Calculus SCT Analysis

We can now prove the partial completeness result relative to the λ calculus analysis. We first give a characterisation of the λ-lifted program corresponding to a λ expression, and then proceed to the main proof.

λ-Lifting

We briefly describe (the relevant aspects of) the λ-lifting procedure, to prove termination of the resulting program. Recall that the aim is to obtain a program P that is in named combinator form and equivalent to Λ. Note here that we assume wlog that all bound variables in λ-abstractions are distinct (use α-conversion). In particular a λ-abstraction is identified by its variable, we write this λx when appropriate. Recall also that each subexpression of Λ is labelled uniquely, and we make this explicit by writing e^l when confusion could occur.

We first define the *context* of an expression. This is a *sequence* of variables, those abstracted in the context in which the expression appears in Λ, in the outermost to innermost order. We write this $C\ e$, note that this only makes sense for labelled e. For example, if

$$\Lambda = \lambda x.((\lambda y.e_1)\ e_2)$$

then $C\ e_1 = \langle x, y \rangle$ and $C\ e_2 = \langle x \rangle$. Note that trivially (provided Λ is closed)

- $C\ e \supseteq FV(e)$ (the free variables of e)
- If e_1 is a subexpression of e_2, then $C\ e_2$ is a prefix of $C\ e_1$.

The λ-lifting algorithm can be described as follows:

1. A new combinator definition f_x is created for each abstraction λx in Λ. There are no other combinators.
2. A map $[\![\bullet]\!]$ that takes subexpressions of Λ to subexpressions of P.

This is easy to define. The definition of f_x is, if the abstraction is $\lambda x.e$:

$$f_x\ c_1\ c_2\ \cdots\ c_k\ x = [\![e]\!] \text{ where } \langle c_1, \ldots, c_k \rangle = C\ (\lambda x.e)$$

We let $n_x = \sharp f_x = |C\ (\lambda x)| + 1$.

The map $[\![\bullet]\!]$ is also easy to define:

Definition 48. We define the λ-lifted expression corresponding to e to be $[\![e]\!]$, where:

$$\begin{array}{rcl} [\![x]\!] & = & x \\ [\![e_1 e_2]\!] & = & [\![e_1]\!]\ [\![e_2]\!] \\ [\![\lambda x.e]\!] & = & f_x\ c_1\ \cdots\ c_k \text{ where } \langle c_1, \ldots, c_k \rangle = C\ (\lambda x.e) \end{array}$$

whenever $e \in \mathit{subexps}(\Lambda)$.

We note a few facts about this procedure:

- $FV([\![e]\!]) \subseteq C\ e$ for any $e \in \mathit{subexps}(\Lambda)$.
- As a result, each combinator definition in P is well-formed (all free variables in the RHS occur as parameters)
- The translation $[\![\bullet]\!]$ really takes a labelled expression to a labelled expression. The effect on labels is defined to be the identity, so that $[\![a^l]\!] = b^l$ for some b whenever $a \in \mathit{subexps}(\Lambda)$.

Note that a subexpression of Λ defines a (depth-1) abstract state. We define a map that takes such an expression to a state. Note that there is a subtle point here, which is that a λ-abstraction $\lambda x.e$ is translated into *two* expressions: the call site $[\![\lambda x.e]\!] = f_x$ *context* and the combinator definition f_x *context* $x = [\![e]\!]$. We will need both translations and therefore introduce two maps: $\lfloor e \rfloor$ and $\lceil e \rceil$, such that $\lfloor \lambda x.e \rfloor$ is the call site and $\lceil \lambda x.e \rceil$ the combinator body:

$$\begin{array}{rcl} \lfloor e \rfloor & = & [\![e]\!] : (\sharp\, [\![e]\!]) \\ \lceil e \rceil & = & \begin{cases} e^{(f_x)} : (\sharp f_x - 1) & \text{if } e = \lambda x.e_b \\ \lfloor e \rfloor & \text{otherwise} \end{cases} \end{array}$$

Note that $\lfloor e \rfloor$ is always a state to be evaluated, whereas $\lceil e \rceil$ is a value whenever e is a value.

The λ-calculus Partial Completeness Proof Throughout, let Λ be a λ-program and P be the program obtained by λ-lifting Λ.

The function definition associated with λx is denoted by f_x, the body of f_x is e_x, the arity of f_x is n_x. We use $c_1, \ldots, c_k$ for arbitrary variable names when these variables appear as *context* variables.

Lemma 30. Suppose that $e_1 e_2 \in \mathit{subexps}(P)$. Then

- *either* $e_1 = [\![d_1]\!]$, $e_2 = [\![d_2]\!]$, $d_1 d_2 \in \mathit{subexps}(\Lambda)$ and $[\![d_1 d_2]\!] = e_1 e_2$
- *or* $e_1 e_2 = f_x\ c_1\ \cdots\ c_i$ for some i, $1 \leq i < n_x$.

Note that if it is further known that $e_1 e_2 = [\![d]\!]$ for some d, the second case is restricted to $i = n_x - 1$.

Proof. Suppose first that $e_1e_2 = [\![d]\!]$ for some $d \in \mathit{subexps}(\Lambda)$. Then there are two cases, as d cannot be a variable ($[\![x]\!] = x \neq e_1e_2$):

If $d = d_1d_2$, then $[\![d]\!] = [\![d_1]\!]\,[\![d_2]\!]$, so necessarily $e_1 = [\![d_1]\!]$, $e_2 = [\![d_2]\!]$ and $e_1e_2 = [\![d_1d_2]\!]$.

If $d = \lambda x.d'$, then $[\![d]\!] = f_x\ c_1\ \cdots\ c_{n_x-1} = e_1e_2$. Note that $n_x > 1$ necessarily as $[\![d]\!]$ is an application, as required.

Now suppose that $e_1e_2 \neq [\![d]\!]$ for any $d \in \mathit{subexps}(\Lambda)$. Then note that e_1e_2 occurs either in the main expression, in which case $e_1e_2 \in \mathit{subexps}([\![\Lambda]\!])$, or in some combinator body, in which case $e_1e_2 \in \mathit{subexps}([\![e]\!])$ for some $\lambda x.e \in \mathit{subexps}(\Lambda)$.

Hence e_1e_2 is a subexpression of some $[\![d]\!]$. Now $d \neq x$ certainly. Also, if $d = d_1d_2$, then as $e_1e_2 \neq [\![d]\!] = [\![d_1]\!]\,[\![d_2]\!]$, e_1e_2 is necessarily a subexpression of $[\![d_1]\!]$ or $[\![d_2]\!]$. Proceeding this way, we find that e_1e_2 is a proper subexpression of $[\![\lambda x]\!]$ for some x. Now

$$[\![\lambda x]\!] = f_x\ c_1\ c_2\ \cdots\ c_{n_x-1} = (\cdots((f_x\ c_1)\ c_2)\cdots c_{n_x-1}) \text{ for some } c_1,\ldots,c_{n_x-1}$$

whence any proper subexpression that is an application is of the form $f_x\ c_1\cdots c_i$, where $1 \leq i < n_x - 1$, as required. $\square$

Corollary 6. Suppose that $e_1e_2 \in \mathit{subexps}(P)$, with $e_1 : n \Downarrow^P f_x : n_x - 1$, where $n = \sharp e_1$. Then $e_1e_2 = [\![d_1d_2]\!]$ where $d_1d_2 \in \mathit{subexps}(P)$.

Proof. The only other case is $e_1e_2 = f_x\ c_1\cdots c_i$, where $i < n_x$. However, then $e_1 : n = f_x\ c_1\cdots c_{i-1}$. Now $i-1 < n_x - 1$, and trivially $e_1 : n$ only reduces to $e_x : (i-1) \neq f_x : (n_x - 1)$. This is a contradiction, as required. $\square$

Corollary 7. Suppose that $e_1e_2 \in \mathit{subexps}(P)$, with $e_1 = [\![d_1]\!]$ or $e_2 = [\![d_2]\!]$ for some $d_i \in \mathit{subexps}(\Lambda)$. Then $e_1e_2 = [\![d_1d_2]\!]$.

Proof. Suppose that $e_1e_2 \neq [\![d_1d_2]\!]$. Then $e_1e_2 = f_x\ c_1\ \cdots\ c_i$ for some $i < n_x$. Then $e_2 = c_i \neq [\![d_2]\!]$ for any $d_2 \in \mathit{subexps}(\Lambda)$. Also, $e_1 = f_X\ c_1\ \cdots\ c_{i-1} \neq [\![d_1]\!]$ for any $d_1 \in \mathit{subexps}(\Lambda)$, as $i-1 < n_x-1$. This is as required. $\square$

Lemma 31. Whenever $s \Rightarrow^P e_x : i$ or $s \Downarrow^P e_x : i$, where $0 \leq i < n_x - 1$, then

- *either* $s = f_x\ c_1\ \cdots\ c_i : n$, where $n = \sharp(f_x\ c_1\cdots c_i)$
- *or* $s = e_x : i$

Note that in the second case $s \Downarrow^P e_x : i$ only, as $s \not\Rightarrow$.

Note also that in all proofs we may safely ignore all rules relating to constants and **if** statements.

Proof. By induction on the proof of $s \Downarrow^P e_x : i$, $s \Rightarrow^P e_x : i$ (resp.) and cases on the last rule used. Let $e^i = f_x\ c_1\ \cdots\ c_i$.

- Value. Then $s \Downarrow^P e_x : i$, and $s = e_x : i$ necessarily.
- Step. Then $s \Rightarrow^P s' \Downarrow^P e_x : i$. By [IH], either $s' = e_x : i$ or $s' = e^i : n$. In the first case, by [IH], either $s = e_x : i$ or $s = e^i : n$, as required.

  In the second case, $s \Rightarrow^P e^i : n$. Now the only rule to conclude $s \Rightarrow^P e^i : n = f_x\ c_1\cdots c_i : n$ is CApp-Call. This can only occur if e^i is the body of a function definition. But as $i < n_x-1$, $e^i \neq [\![d]\!]$ for any $d \in \mathit{subexps}(\Lambda)$. This is a contradiction, as all function bodies in P arise in this way, as required.

- Function. Then $s \Downarrow e_x : i$, and necessarily $i = 0$. Then $s = f_x : n$, as required.
- Variable. Suppose that $s \Downarrow^P e_x : i$ by Variable. Then $s = y : n$ for some variable y appearing in f say. Now $f, y \multimap e_x : i$ appears as a subproof of $s \Downarrow^P e_x : i$, so that there exists $e_1 e_2 \in \mathit{subexps}(P)$ with $e_1 : \sharp e_1 \Downarrow^P f : j$ for some j and $e_2 : \sharp e_2 \Downarrow^P e_x : i$. Now by [IH], as $e_2 : \sharp e_2$ is not an incomplete closure, this forces $e_2 = f_x\ c_1 \cdots c_i$. But there is only one occurrence of f_x in P, in the expression $f_x\ c_1 \cdots c_{n_x - 1}$. Now as $i < n_x - 1$, e_2 cannot be the operand in any application, a contradiction.
- CApp-Call. Suppose that $s \Rightarrow^P e_x : i$ by CApp-Call. Then $s = e_1 e_2 : n$, where $e_1 : n \Downarrow^P e_x : i - 1$. By [IH], $e_1 = f_x\ c_1 \cdots c_{i-1}$. Now there is only one occurrence of f_x, and so this forces $e_2 = c_i$, whence $s = f_x\ c_1 \cdots c_i : n$, as required.

□

Lemma 32. For any variable x appearing as a parameter of f_y,

$$f_y, x \multimap v \iff f_x, x \multimap v$$

In addition, given a proof of $f_y, x \multimap v$, $y \neq x$, $f_x, x \multimap v$ appears as a subproof.

Proof. Note that if x appears as a parameter of f_y, then λy appears a subexpression of λx, so that $C\ (\lambda y)$ is a suffix of $C\ (\lambda x)$, say

$$C\ (\lambda x) = \langle c_1, \ldots, c_k \rangle \text{ and } C\ (\lambda y) = \langle c_1, \ldots, c_k, x, x_1, \ldots, x_{n-1} \rangle$$

where $n \geq 0$. We define $x_0 = x$ and $x_n = y$. We proceed by induction on n.

Base Case Then $n = 0$, whence $x = y$ and so this is trivial

Inductive Step Suppose that $n > 0$ and this holds for $n - 1$.

($\Leftarrow$). Suppose first that $f_x, x \multimap v$. Then [IH], $f_{x_{n-1}}, x \multimap v$. But λy is a subexpression of λx_{n-1}, whence

$$f_y\ c_1\ \cdots\ c_k\ x\ \cdots\ x_n \text{ is a subexpression of } e_{x_{n-1}}$$

Let $l = \sharp e_{x_{n-1}}$. Then $f_y\ c_1\ \cdots\ c_k : l \Downarrow^P e_y : k$ certainly. Also, $x : l \Downarrow^P v$ as $f_{x_{n-1}}, x \multimap v$. Hence we conclude $f_y, (k+1) \multimap v$, whence $f_y, x \multimap v$ as x is the $(k+1)^{th}$ variable of f_y

($\Rightarrow$). Suppose that $f_y, x \multimap v$, and consider a given proof of this. We must show that this contains $f_x, x \multimap v$ as a subproof. Now the antecedents of $f_y, x \multimap v$ are $e_1 e_2 \in \mathit{subexps}(P)$, $e_1 : l \Downarrow^P f_y : k$, $e_2 : l \Downarrow^P v$, where $l = \sharp e_1 e_2$ (recall that x is the $(k+1)^{th}$ variable of f_y).

Now $n_y = k + n + 1$, and $n > 0$, so that $k < n_y - 1$. By Lemma 31, $e_1 = f_y\ c_1\ \cdots\ c_k$. Now there is only one textual occurrence of f_y in P, namely in the body of $f_{x_{n-1}}$. Also, this occurs within $[\![\lambda y]\!] = f_y\ c_1 \cdots x \cdots x_n$, whence necessarily $e_2 = x$.

Now $e_2 : l \Downarrow^P v$ occurs as a subproof of $f_y, x \multimap v$. As $e_2 = x$, the only rule that applies to conclude $x : l \Downarrow^P$ is rule Variable (the Step rule can only be used with an application state by cases chasing). Thus $f_{x_{n-1}}, x \multimap v$ occurs as a subproof of $e_2 : l \Downarrow^P v$. By [IH], either $x_{n-1} = x$ in which case we are done, or a proof of $f_x, x \multimap v$ occurs as a subproof of $f_{x_{n-1}}, x \multimap v$. This is as required. □

Corollary 8. Suppose that $f_y, x \multimap v$. Then there exists $d_1 d_2 \in \mathit{subexps}(\Lambda)$ with $\lfloor d_1 \rfloor \Downarrow^P f_x : n_x - 1$ and $\lfloor d_2 \rfloor \Downarrow^P v$, both subproofs of $f_y, x \multimap v$.

Proof. By Lemma 32, either $x = y$ or $f_x, x \multimap v$ occurs as a subproof. Hence there exists $e_1 e_2 \in \mathit{Subexps}(P)$ with $e_1 : \sharp e_1 \Downarrow^P f_x : n_x - 1$ and $e_2 : \sharp e_2 \Downarrow^P v$, both subproofs of $f_y, x \multimap v$. By Corollary 6, $e_1 e_2 = [\![d_1 d_2]\!]$ where $d_1 d_2 \in \mathit{subexps}(\Lambda)$. Then $e_i = [\![d_i]\!]$, whence $e_i : \sharp e_i = \lfloor d_i \rfloor$ $(i = 1, 2)$, as required. $\square$

Lemma 33. Suppose that $s = \lfloor e \rfloor$ for some $e \in \mathit{subexps}(\Lambda)$. Then if $s \Downarrow^P v$, then $v = \lceil w \rceil$ for some $w \in \mathit{subexps}(\Lambda)$ and $e \Downarrow^\Lambda w$.

Proof. We must strengthen the result to use induction. The additional result that we prove is:

If $s \Rightarrow^P s'$, where $s = \lfloor e \rfloor$, then

- If e is an application, then $s' = \lfloor e' \rfloor$ with $e \Rightarrow^\Lambda e'$
- If e is an abstraction, then $s' = \lceil e \rceil$.

The proof is by induction on the proof of $s \Rightarrow^P s'$, resp. $s \Downarrow^P v$, and cases on e.

- $e = x$

Then $s = x : n$, where $n = \sharp x$. Then the only case is $s \Downarrow^P v$ by Variable. By Corollary 8, there exists $d_1 d_2 \in \mathit{subexps}(\Lambda)$, with $\lfloor d_1 \rfloor \Downarrow^P f_x : n_x - 1$, $\lfloor d_2 \rfloor \Downarrow^P v$, both subproofs. By [IH], $f_x : n_x - 1 = \lceil w \rceil$ for some $w \in \mathit{subexps}(\Lambda)$, and $d_1 \Downarrow^\Lambda w$. The only possibility is $w = \lambda x.b$ for some b. Also, by [IH], $v = \lceil u \rceil$ for some $u \in \mathit{subexps}(\Lambda)$, and $d_2 \Downarrow^\Lambda u$. By the closure analysis rule in the λ analysis, $x \Downarrow^\Lambda u$, as required as $v = \lceil u \rceil$.

- $e = e_1 e_2$

As $s = \lceil e \rceil$, $s = d_1 d_2 : l$, where $d_i = [\![e_i]\!]$ and $l = \sharp d_1 d_2$. The possible reductions are $s \Rightarrow^P$ by CApp-Call, and $s \Downarrow^P$ by Step.

If $s \Rightarrow^P s'$ by CApp-Call, then $d_1 : l \Downarrow^P e_x : i$ for some $i < n_x$, and $d_2 : l \Downarrow^P v$ for some value v. Then $s' = e_x : i + 1$. Now by [IH], as $d_1 : l = \lfloor e_1 \rfloor$, $e_x : i = \lceil b \rceil$ for some $b \in \mathit{subexps}(\Lambda)$. The only possibility is $b = \lambda x$, and thus $i = n_x - 1$. Also by [IH] we have $e_1 \Downarrow^\Lambda \lambda x.e_b$. By [IH] again, $e_2 \Downarrow^\Lambda w$ for some w, with $v = \lceil w \rceil$. Then by the λ call rule, $e_1 e_2 \Rightarrow^\Lambda e_b$. But $\lfloor e_b \rfloor = e_x : n_x = e_x : i + 1$ as $i = n_x - 1$, as required.

If $s \Downarrow^P v$ by Step, then $s \Rightarrow^P s' \Downarrow^P v$ for some s'. By [IH], $s' = \lfloor e' \rfloor$ for some e' with $e \Rightarrow^\Lambda e'$. Again by [IH], $v = \lceil w \rceil$ for some w with $e' \Downarrow^\Lambda w$. By the step rule in the λ analysis, $e \Downarrow^\Lambda w$, as required.

- $e = \lambda x.b$

Finally, suppose that $e = \lambda x.b$. Then $\lfloor e \rfloor = f_x\ c_1\ \cdots\ c_{n_x - 1} : n$.

The only proper calls and reductions of s are then $s \Rightarrow^P e_x : n_x - 1$ and $s \Downarrow^P e_x : n_x - 1$, trivially (note that this is also the case if $n_x = 1$). But this is as required, as $e_x : n_x - 1 = \lceil \lambda x.b \rceil = \lceil e \rceil$, and $\lambda x.b \Downarrow^P \lambda x.b$ by the λ value rule. This concludes the proof. $\square$

Lemma 34. Suppose that $s \to^P \lfloor e \rfloor$ for some $e \in \mathit{subexps}(\Lambda)$. Then $s = \lfloor e' \rfloor$ where $e' \in \mathit{subexps}(\Lambda)$, and $e' \to^\Lambda e$.

Proof. Note that the only rules that can conclude $s \to^P \lfloor e \rfloor$ are CApp-1, CApp-2 and CApp-Call (Function cannot apply as $\lfloor e \rfloor$ is not a closure).

Suppose first that $[\![e]\!]$ is the right-hand side of a function definition, say of f_x. That is, $\lfloor e \rfloor = e_x : n_x$ necessarily. Then only CApp-Call can apply. We then deduce that $s = d_1 d_2 : n$, where $n = \sharp d_1 d_2$, and $d_1 : n \Downarrow^P f_x : n_x - 1$ and $d_2 : n \Downarrow^P v$ for some v. Now by Corollary 6, $d_1 d_2 = [\![e_1 e_2]\!]$ for some $e_i \in \mathit{subexps}(\Lambda)$, whence $s = \lfloor e_1 e_2 \rfloor$. Also, $d_i : n = \lfloor e_i \rfloor$ $(i = 1, 2)$. By Lemma 33, $e_x : n_x - 1 = \lceil v \rceil$ for some $v \in \mathit{subexps}(\Lambda)$ with $e_1 \Downarrow^\Lambda v$. The only such v is $\lambda x.e$ (as $[\![e]\!]$ is the body of f_x), whence $e_1 \Downarrow^\Lambda \lambda x.e$. By Lemma 33 again, $e_2 \Downarrow^\Lambda w$ for some w. Thus $e_1 e_2 \Rightarrow^\Lambda e$ by the λ call rule, as required.

Now we suppose that $[\![e]\!]$ is not the right-hand side of a function definition. Then the only rules that can apply are CApp-1 and CApp-2.

If the rule used is CApp-1, then $s = [\![e]\!]\, d' : n$ for some d'. Now by Corollary 7 this forces $[\![e]\!]\, d' = [\![ee']\!]$ for some $e' \in \mathit{subexps}(\Lambda)$, whence $s = \lfloor ee' \rfloor$. By the λ operator rule, $ee' \rightarrow^\Lambda e$, as required.

If the rule used is CApp-2, then $s = d'\, [\![e]\!] : n$ for some d'. By Corollary 7 again this forces $d'\, [\![e]\!] = [\![e'e]\!]$ for some $e' \in \mathit{subexps}(\Lambda)$, whence $s = \lfloor e'e \rfloor$. By the λ operand rule, $e'e \rightarrow^\Lambda e$, as required again. ☐

Corollary 9. Suppose that $s_0 \rightarrow s_1 \rightarrow \cdots \rightarrow s_N \rightarrow \cdots \rightarrow s_{N+k-1} \rightarrow s_N$ for some states s_i, all in P. Then $s_i = \lfloor t_i \rfloor$ for some $t_i \in \mathit{subexps}(\Lambda)$ for each i, and $t_0 \rightarrow \cdots \rightarrow t_N \rightarrow \cdots \rightarrow t_{N+k-1} \rightarrow t_N$.

Proof. By the previous result this holds immediately provided we can show that some s_{N+i}, $i \geq 0$ satisfies $s_{N+i} = \lfloor t_{N+i} \rfloor$ — then as $s_N \rightarrow^* s_{N+i}$ for all i, this shows that $s_N = \lfloor t_N \rfloor$ for some t_N, and we can work backwards along the chain from there.

By Lemma 25, some s_{N+i} is a function body state of the form $e_x : n_x$ for some x. But then $s_{N+i} = \lfloor e_b \rfloor$, where $\lambda x.e_b \in \mathit{subexps}(\Lambda)$, whence the result follows. ☐

We must now strengthen these results to prove that the size-change graphs that arise in both abstract interpretations are related. As each rule in the P abstract interpretation has a unique counterpart in the λ abstract interpretation, it suffices to show that the graphs produced by each rule are related.

In particular, we wish to show that whenever γ is produced in the P analysis and δ is the corresponding graph produced in the Λ analysis, then $\delta \subseteq \gamma$. We first observe:

Lemma 35. Let $e, v \in \mathit{subexps}(\Lambda)$, with v a value, say $v = \lambda x.b$. Then:

- $\mathit{GB}(\lfloor e \rfloor) \supseteq \{\langle x \rangle \mid x \in \mathit{fv}(e)\} \cup \{\epsilon\}$
- $\mathit{GB}(\lceil v \rceil) \supseteq \{\langle x \rangle \mid x \in \mathit{fv}(v)\} \cup \{\epsilon\}$

Proof. Recall that

$$\mathit{GB}(l : n) = \{\epsilon\} \cup \{x_i \mid 1 \leq i \leq n\}$$

for any depth-1 abstract state $l : n$. Now $\lfloor e \rfloor = l : n$ where $n = \sharp l$. Now the parameters of the combinator in which $\lfloor e \rfloor$ appears are precisely $C\ e$, the context of e. Hence the first result is immediate as $C\ e \supseteq \mathit{fv}(e)$.

For the second case, $\lceil v \rceil = e_x : n_x - 1$. Now the parameters of f_x are $C\ v +\!\!+ \langle x \rangle$. Hence the first $n_x - 1$ parameters of f_x are precisely $C\ v$, as required. ☐

We will in the sequel identify $\langle x \rangle$ with x and ϵ with $\bullet$, as this is a straightforward isomorphism between the set of graph nodes in both the P analysis and the Λ analysis. No confusion can arise.

We can now prove that the graph constructions are as required:

- The Value and (ValueAG) Rules

The graph constructed in Value is

$$1_v = \{p \xrightarrow{\equiv} p \mid p \in GB(v)\}$$

The graph constructed in (ValueAG) is

$$id^{\equiv}_{\lambda x.b} = \{\bullet \xrightarrow{\equiv} \bullet\} \cup \{x \xrightarrow{\equiv} x \mid x \in fv(\lambda x.b)\}$$

Lemma 36. Whenever $v \in subexps(\Lambda)$ is a value, $id^{\equiv}_v \subseteq 1_{\lceil v \rceil}$.

Proof. Immediate by Lemma 35. ☐

- The $\mathsf{CApp\text{-}1}$, $\mathsf{CApp\text{-}2}$ and (OperatorAG), (OperandAG) Rules

The graph constructed in both $\mathsf{CApp\text{-}1}$ and $\mathsf{CApp\text{-}2}$ is

$$1^{\downarrow}_e = \{\epsilon \xrightarrow{\downarrow} \epsilon\} \cup \{p \xrightarrow{\equiv} p \mid p \in GB(e) \wedge |p| > 1\}$$

The graph constructed in both (OperatorAG) and (OperandAG) rules is:

$$id^{\downarrow}_e = \{\bullet \xrightarrow{\downarrow} \bullet\} \cup \{x \xrightarrow{\equiv} x \mid x \in fv(e)\}$$

Lemma 37. For any expression $e \in subexps(\Lambda)$, $id^{\downarrow}_e \subseteq 1^{\downarrow}_{\lfloor e \rfloor}$

Proof. Immediate by Lemma 35. ☐

- The $\mathsf{Variable}$ and (VarAG) Rules

The graph constructed in the $\mathsf{Variable}$ rule is:

$$[1^{x \mapsto \epsilon}_v]_1 = [\{xp \xrightarrow{\equiv} p \mid p \in GB(v)\}]_1$$

It is easy to see that this equals

$$[1^{x \mapsto \epsilon}_v]_1 = \{\langle x \rangle \xrightarrow{\equiv} \epsilon\} \cup \{\langle x \rangle \xrightarrow{\downarrow} \langle y \rangle \mid \langle y \rangle \in GB(v)\}$$

The graph constructed in the (VarAG) rule is:

$$id^{x \mapsto \bullet}_v = \{x \xrightarrow{\equiv} \bullet\} \cup \{x \xrightarrow{\downarrow} y \mid y \in fv(v)\}$$

Lemma 38. For any value $v \in subexps(\Lambda)$, $id^{x \mapsto \bullet}_v \subseteq 1^{x \mapsto \epsilon}_{\lceil v \rceil}$.

Proof. Immediate by Lemma 35. ☐

- The $\mathsf{CApp\text{-}Call}$ and (CallAG) Rules

Both rules involve graphs γ_1 (generated when reducing the operator) and γ_2 (generated when reducing the operand). The graph constructed in $\mathsf{CApp\text{-}Call}$ is $[AppG_x(\gamma_1, \gamma_2)]_1$, where x is the variable to be bound in the application. Expanding the definition and specialising to the depth-1 case, we get

$$\begin{aligned} AppG_x(\gamma_1, \gamma_2) \quad = \quad & \{\epsilon \xrightarrow{\downarrow} y \mid \epsilon \rightarrow y \in \gamma_1\} \\ & \cup \{z \xrightarrow{l} y \mid z \xrightarrow{l} y \in \gamma_1\} \\ & \cup \{\epsilon \xrightarrow{\downarrow} x \mid \epsilon \rightarrow \epsilon \in \gamma_2\} \\ & \cup \{z \xrightarrow{l} x \mid z \xrightarrow{l} \epsilon \in \gamma_2\} \\ & \cup \{z \xrightarrow{l} xy \mid z \xrightarrow{l} y \in \gamma_2\} \end{aligned}$$

Therefore we find (up to removing unnecessary edges):

$$\begin{array}{rcl}[AppG_x(\gamma_1,\gamma_2)]_1 & = & \{\epsilon \xrightarrow{\downarrow} y \mid \epsilon \to y \in \gamma_1\} \\ & & \cup \{z \xrightarrow{l} y \mid z \xrightarrow{l} y \in \gamma_1\} \\ & & \cup \{\epsilon \xrightarrow{\downarrow} x \mid \epsilon \to \epsilon \in \gamma_2\} \\ & & \cup \{z \xrightarrow{l} x \mid z \xrightarrow{l} \epsilon \in \gamma_2\}\end{array}$$

This is exactly identical to the definition of the graph $\gamma_1^{-\bullet} \cup \gamma_2^{\bullet \mapsto x}$ produced in the (CallAG) rule from the same graphs.

Lemma 39. Suppose that $\delta_1 \subseteq \gamma_1$ and $\delta_2 \subseteq \gamma_2$. Then

$$\delta_1^{-\bullet} \cup \delta_2^{\bullet \mapsto x} \subseteq [AppG_x(\gamma_1,\gamma_2)]_1$$

Proof. All the operations used in constructing the graphs are monotonic (in particular, union) and the definitions are identical. □

- The Step and (ApplyAG) Rules

in both cases, the antecedents are $e \Rightarrow e', \gamma_1$ and $e' \Downarrow v, \gamma_2$. The consequent is $e \Downarrow v, (\gamma_1; \gamma_2)$. We recall:

Lemma 40. Suppose that $\gamma_1 \subseteq \delta_1$ and $\gamma_2 \subseteq \delta_2$. Then $\gamma_1; \gamma_2 \subseteq \delta_1; \delta_2$.

which is the required property.

The proofs of the following are now immediate using the properties of graphs to strengthen the inductive proofs given previously:

Lemma 41. Suppose that $s = \lfloor e \rfloor$ for some $e \in \mathit{subexps}(\Lambda)$. Then if $s \Downarrow^P v, \gamma$, then $v = \lceil w \rceil$ for some $w \in \mathit{subexps}(\Lambda)$ and $e \Downarrow^\Lambda w, \delta$ where $\delta \subseteq \gamma$.

Lemma 42. Suppose that $s \to^P \lfloor e \rfloor, \gamma$ for some $e \in \mathit{subexps}(\Lambda)$. Then $s = \lfloor e' \rfloor$ where $e' \in \mathit{subexps}(\Lambda)$, and $e' \to^\Lambda e, \delta$ for some $\delta \subseteq \gamma$.

As a result:

Corollary 10. Suppose that γ is a graph of type $a \to a$ for some a, with $\gamma \in \mathrm{Cl}(S_1(P))$ (the set of graphs produced by the depth 1 analysis on P). Then there exists a graph $\delta \in \mathrm{Cl}(\mathit{abstint}(\Lambda))$ with $\delta \subseteq \gamma$.

Proof. Such a graph can only arise as $\gamma = \gamma_0; \gamma_1; \cdots; \gamma_{k-1}$ where

$$\begin{array}{l}P = s_0 \to^* s_N \to s_{N+1} \to \cdots \to s_{N+k-1} \to s_{N+k} = s_N \\ \text{and } s_{N+i} \to s_{N+i+1}, \gamma_i \text{ for each } i\end{array}$$

Note that possibly $k = 1$ in the above. Now by the previous result and Corollary 9, there is a corresponding reduction sequence in Λ:

$$\begin{array}{l}\Lambda \to^* t_N \to^\Lambda t_{N+1} \to^\Lambda \cdots \to^\Lambda t_{N+k-1} \to^\Lambda t_{N+k} = t_N \\ \text{with for all } i,\ s_i = \lfloor t_i \rfloor \text{ and } t_{N+i} \to^\Lambda t_{N+i+1}, \delta_i \text{ for some } \delta_i \subseteq \gamma_i\end{array}$$

Then $\delta = \delta_0; \delta_1; \cdots; \delta_{k-1} \in \mathrm{Cl}(\mathit{abstint}(\Lambda))$. Also, by Lemma 28, $\delta \subseteq \gamma$, as required. □

We can now deduce:

Theorem 5. Suppose that Λ is size-change terminating. Then P is size-change terminating in the depth-1 analysis.

Proof. Suppose that P is not size-change terminating. Then there exists a graph $\gamma : s \to s$ in $\mathrm{Cl}(S_1(P))$, with $\gamma^2 = \gamma$ and no arc $x \xrightarrow{\downarrow} x$ in γ. By Corollary 10, there exists a graph $\delta \in \mathrm{Cl}(\mathit{abstint}(\Lambda))$ with $\delta \subseteq \gamma$. Then $\delta : t \to t$ for some state t with $s = \lfloor t \rfloor$.

Now some positive power δ^n $(n > 0)$ of δ is idempotent. Furthermore, $\delta^n \subseteq \gamma^n$. Also, $\delta^n \in \mathrm{Cl}(\mathit{abstint}(P))$ as this is closed under composition of graphs.

Suppose for a contradiction that δ^n has an *in-situ* descent, that is $x \xrightarrow{\downarrow} x$ occurs in δ^n. Then as $\delta^n \subseteq \gamma^n$, $x \xrightarrow{l} x$ occurs in γ^n for some $l \geq \downarrow$. This forces $l = \downarrow$. But γ is idempotent and $n \geq 1$, whence $\gamma^n = \gamma$. Thus $x \xrightarrow{\downarrow} x$ occurs in γ, contradicting the choice of γ.

Hence δ^n is idempotent and has no *in-situ* descent. Thus $\mathit{abstint}(\Lambda)$ is not size-change terminating. That is, Λ is not size-change terminating, as required. $\square$

References

[1] Amir M. Ben-Amram. General size-change termination and lexicographic descent. *Lecture Notes in Computer Science*, 2566:3–17, 2002.

[2] Richard Bird. Using circular programs to eliminate multiple traversals of data. *Acta Informatica*, 21:239–250, 1984.

[3] Robin Cockett and Tom Fukushima. About Charity. Yellow Series Report No. 92/480/18, Department of Computer Science, The University of Calgary, June 1992.

[4] Olivier Danvy. Lambda-lifting in quadratic time. In *FLOPS*, volume 2441 of *Lecture Notes in Computer Science*, pages 134–151. Springer, 2002.

[5] Carl Christian Frederiksen. A simple implementation of the size-change termination principle. Technical report, DIKU, Copenhagen University, 2001. TOPPS Reference D-442.

[6] Carl Christian Frederiksen. Automatic runtime analysis for first order functional programs. Master's thesis, DIKU, University of Copenhagen, September 2002.

[7] Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and Types.* Cambridge tracts in Theoretical Computer Science. Cambridge University Press, 1989.

[8] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic.* Cambridge University Press, 1993.

[9] Tatsuya Hagino. *A Categorical Programming Language.* PhD thesis, University of Edinburgh, 1986.

[10] Thomas Johnsson. Lambda lifting: transforming programs to recursive equations. In *Proceedings of a conference on Functional Programming languages and Computer Architecture.* Springer-Verlag, 1985.

[11] Thomas Johnsson. Attribute grammars as a functional programming paradigm. In *Proceedings of a conference on Functional Programming languages and Computer Architecture*, pages 154–173. Springer-Verlag, 1987.

[12] Neil D. Jones and Nina Bohr. Termination analysis of the untyped λ-calculus. In *Rewriting Techniques and Applications: 15th international conference, RTA 2004*, volume 3091 of *Lecture Notes in Computer Science*, pages 1–23, 2004.

[13] Neil D. Jones and Arne John Glenstrup. Program generation, termination, and binding-time analysis. In Walid Taha, editor, *Proceedings of GPCE'02.* Springer-Verlag, 2002.

[14] Donald Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):239–250, 1968. Correction in Math. Syst. Theory 5(1):95-96.

[15] Chin Soon Lee. *Program Termination Analysis and the Termination of Offline Partial Evaluation.* PhD thesis, University of Western Australia, March 2001.

[16] Chin Soon Lee. Finiteness analysis in polynomial time. In M. Hermenegildo and G. Puebla, editors, *Static Analysis: 9th International Symposium, SAS 2002*, volume 2477 of *Lecture Notes in Computer Science*, pages 493–508. Springer, September 2002.

[17] Chin Soon Lee. Program termination analysis in polynomial time. In Don Batory, Charles Consel, and Walid Taha, editors, *Generative Programming and Component Engineering: ACM SIGPLAN/SIGSOFT Conference, GPCE 2002*, volume 2487 of *Lecture Notes in Computer Science*, pages 218–235. ACM, Springer, October 2002.

[18] Chin Soon Lee, Neil D. Jones, and Amir M. Ben-Amram. The size-change principle for program termination. *ACM SIGPLAN Notices*, 36(3):81–92, 2001.

[19] Xavier Leroy. *The Objective Caml System: Documentation and User's Manual*, 2004.

[20] Chris Okasaki. Even higher-order functions for parsing or why would anyone ever want to use a sixth-order function? *Journal of Functional Programming*, 8(2):195–199, March 1998.

[21] Lars Pareto. *Types for Crash Prevention*. PhD thesis, Chalmers University of Technology, 2000.

[22] R. Péter. *Recursive Functions*. Academic Press, third edition, 1967.

[23] Simon Peyton-Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, 1987.

[24] Olin Shivers. Control-flow analysis in scheme. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, June 1988.

[25] Konrad Slind. *Reasoning about Terminating Functional Programs*. PhD thesis, Institut für Informatik der Technischen Universität München, 1999.

[26] Alastair Telford and David Turner. A hierarchy of languages with strong termination properties. Technical Report TR 2-00, Computing Lab, University of Kent at Canterbury, The Computing Laboratory, The University, Canterbury, Kent, CT2 7NF, February 2000. Paper currently under revision.