Termination Analysis of the Untyped λ-Calculus

Neil D. Jones[1] and Nina Bohr[2]

[1]DIKU, University of Copenhagen and [2]IT University of Copenhagen

Abstract. An algorithm is developed that, given an untyped λ-expression, can certify that its call-by-value evaluation will terminate. It works by an extension of the "size-change principle" earlier applied to first-order programs. The algorithm is sound (and proven so in this paper) but not complete: some λ-expressions may in fact terminate under call-by-value evaluation, but not be recognised as terminating.
The *intensional* power of size-change termination is reasonably high: It certifies as terminating all primitive recursive programs, and many interesting and useful general recursive *algorithms* including programs with mutual recursion and parameter exchanges, and Colson's "minimum" algorithm. Further, the approach allows free use of the Y combinator, and so can identify as terminating a substantial subset of PCF.
The *extensional power* of size-change termination is the set of functions computable by size-change terminating programs. This lies somewhere between Péter's multiple recursive functions and the class of ϵ_0-recursive functions.

1 Introduction

The *size-change* analysis of [5] can show termination of first-order functional programs whose parameter values have a well-founded size order. The method is reasonably general, easily automated, and does not require human invention of lexical or other parameter orders. This paper applies similar ideas to establish termination of *higher-order* programs. For simplicity and generality we focus on the simplest such language, the λ-calculus. We expect that the framework can be naturally extended to higher-order functional programs, e.g., functional subsets of Scheme or ML.

1.1 An example of size-change analysis

Example 1. A motivating example is the size-change termination analysis of a first-order program, using the framework of [5]. Consider a program with functions f and g defined by mutual recursion:

```
f(x,y)   = if x=0 then y else 1: g(x,y,y)
g(u,v,w) = if w=0 then 3:f(u-1,w) else 2:g(u,v-1,w+2)
```

The three function calls have been labeled 1, 2 and 3. The "control flow graph" in Figure 1 shows the calling function and called function of each call, e.g., $3 : \texttt{g} \to \texttt{f}$. Each call is associated with a "size-change graph", e.g., G_3 for call 3, that describes the data flow from the calling function's parameters to the called function's parameters.

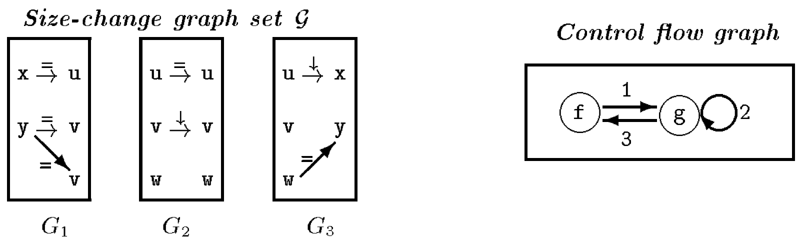

Size-change graph set $\mathcal{G}$
$\texttt{x} \xrightarrow{=} \texttt{u}$
$\texttt{y} \xrightarrow{=} \texttt{v}$
=
$\texttt{u} \xrightarrow{=} \texttt{u}$
$\texttt{v} \xrightarrow{\downarrow} \texttt{v}$
w w
$\texttt{u} \xrightarrow{\downarrow} \texttt{x}$
v y
=
w
G_1
G_2
G_3
Control flow graph
f
g
1
3
2

Figure 1: Call graph and size-change graphs for the example first-order program.

Termination reasoning: Consider (in order to prove it impossible) any *infinite size-change graph sequence* $\mathcal{M} = g_1 g_2 \ldots \in \{G_1, G_2, G_3\}^\omega$ that follows the program's control flow:

> **Case 1:** $\mathcal{M} = \ldots (G_2)^\omega$ ends in infinitely many G_2's: In this case, *variable* `v` *descends infinitely.*
> **Case 2:** $\mathcal{M} = \ldots (G_1 G_2^* G_3)^\omega$. In this case, *variable* `u` *descends infinitely.*

Both cases are impossible (assuming, as we do, that the data value set is well-founded). Therefore a call of *any program function with any data will terminate.* *End of example.*

1.2 Definitions and terminology

We describe the structure of size-change graphs and state the size-change termination condition.

Definition 1.

1. *A* size-change graph $A \xrightarrow{G} B$ *consists of a* source *set* A*; a* target *set* B*; and a set of labeled arcs* $G \subseteq A \times \{=, \downarrow\} \times B$*. If* A, B *are clear from context, we identify* $A \xrightarrow{G} B$ *with its arc set* G*.*
2. *The* identity *size-change graph for* A *is* $A \xrightarrow{id_A} A$ *where* $id_A = \{\mathtt{x} \xrightarrow{=} \mathtt{x} \mid \mathtt{x} \in A\}$*.*
3. *Size-change graphs* $A \xrightarrow{G_1} B$ *and* $C \xrightarrow{G_2} D$ *are* composible *if* $B = C$*.*
4. *The* sequential composition *of size-change graphs* $A \xrightarrow{G_1} B$ *and* $B \xrightarrow{G_2} C$ *is* $A \xrightarrow{G_1;G_2} C$ *where*

$$\begin{aligned} G_1;G_2 = \{\mathtt{x} \xrightarrow{\downarrow} \mathtt{z} \mid\ &\downarrow\ \in \{\, r, s \mid \mathtt{x} \xrightarrow{r} \mathtt{y} \in G_1 \textit{ and } \mathtt{y} \xrightarrow{s} \mathtt{z} \in G_2 \textit{ for some } \mathtt{y} \in B\}\ \} \\ \cup \{\mathtt{x} \xrightarrow{=} \mathtt{z} \mid\ &\{=\} = \{\, r, s \mid \mathtt{x} \xrightarrow{r} \mathtt{y} \in G_1 \textit{ and } \mathtt{y} \xrightarrow{s} \mathtt{z} \in G_2 \textit{ for some } \mathtt{y} \in B\}\ \} \end{aligned}$$

Lemma 1. *Sequential composition is associative.* $A \xrightarrow{G} B$ *implies* $id_A; G = G; id_B = G$*.*

Definition 2. *A* multipath $\mathcal{M}$ *over a set* $\mathcal{G}$ *of size-change graphs is a finite or infinite composible sequence of graphs in* $\mathcal{G}$*. Define*

$$\mathcal{G}^\omega = \{\mathcal{M} = G_0, G_1, \ldots \mid \textit{graphs } G_i, G_{i+1} \textit{ are composible for } i = 0, 1, 2, \ldots\}$$

Definition 3.

1. *A* thread *in a multipath* $\mathcal{M} = G_0, G_1, G_2, \ldots$ *is a sequence* $t = a_j \xrightarrow{r_j} a_{j+1} \xrightarrow{r_{j+1}} \ldots$ *such that* $a_k \xrightarrow{r_k} a_{k+1} \in G_k$ *for every* $k \geq j$ *(and each* r_k *is* $=$ *or* $\downarrow$*.)*
2. *Thread* t *is of* infinite descent *if* $r_k = \downarrow$ *for infinitely many* $k \geq j$*.*
3. *A set* $\mathcal{G}$ *of size-change graphs satisfies the* size-change condition *if every* $\mathcal{M} \in \mathcal{G}^\omega$ *contains at least one thread of infinite descent.*

The size-change condition is decidable; its worst-case complexity, as a function of the size of the program being analysed, is shown complete for PSPACE in [5].

The example revisited: The program of Figure 1 has three size-change graphs, one for each of the calls `1 : f` $\to$ `g`, `2 : g` $\to$ `g`, `3 : g` $\to$ `f`, so $\mathcal{G} = \{A \xrightarrow{G_1} B, B \xrightarrow{G_2} B, B \xrightarrow{G_2} A\}$ where $A = \{\mathtt{x}, \mathtt{y}\}$ and $B = \{\mathtt{u}, \mathtt{v}, \mathtt{w}\}$. (Note: the vertical layout of the size-change graphs in Figure 1 is inessential, though intuitively appealing. One could simply write, for instance, $G_3 = \{\mathtt{u} \xrightarrow{\downarrow} \mathtt{x}, \mathtt{w} \xrightarrow{=} \mathtt{y}\}$.)

The termination reasoning shows that $\mathcal{G}$ satisfies the size-change condition: Every infinite multipath has either a thread that decreases `u` infinitely, or a thread that decreases `v` infinitely.

2 The call-by-value λ-calculus

We first summarise some standard definitions and results.

Definition 4. *Exp is the set of all λ-expressions that can be formed by these syntax rules, where* `@` *is the* application operator *(sometimes omitted). We use the* `teletype` *font for λ-expressions.*

$$\begin{array}{lll} \mathtt{e},\ \mathtt{P} & ::= & \mathtt{x} \mid \mathtt{e}\ \mathtt{@}\ \mathtt{e} \mid \lambda\mathtt{x}.\mathtt{e} \\ \mathtt{x} & ::= & \text{Variable name} \end{array}$$

- *The set of* free variables $fv(\mathtt{e})$ *is defined in the usual way:* $fv(\mathtt{x}) = \{\mathtt{x}\}$, $fv(\mathtt{e@e'}) = fv(\mathtt{e}) \cup fv(\mathtt{e'})$ *and* $fv(\lambda\mathtt{x}.\mathtt{e}) = fv(\mathtt{e}) \setminus \{\mathtt{x}\}$. *A closed λ-expression* `e` *satisfies* $fv(\mathtt{e}) = \emptyset$.
- *A* program, *usually denoted by* `P`, *is any closed λ-expression.*
- *The set of* subxpressions *of a λ-expression* `e` *is denoted by* $subexp(\mathtt{e})$.

The following is standard, e.g., [9]. Notation: $\mathtt{e}[v/\mathtt{x}]$ (β-reduction) is the result of substituting v for all free occurrences of `x` in `e` and renaming λ-bound variables if needed to avoid capture.

Definition 5. (Call-by-value semantics) *The* call-by-value evaluation relation *is defined by the following inference rules, with judgement form* $\mathtt{e} \Downarrow v$ *where* `e` *is a λ-expression and* $v \in ValueS$. *ValueS (for "standard value") is the set of all abstractions* $\lambda\mathtt{x}.\mathtt{e}$.

$$(\text{ValueS})\ \frac{}{v \Downarrow v}\ (\text{If } v \in \mathit{ValueS}) \qquad (\text{ApplyS})\ \frac{\mathtt{e}_1 \Downarrow \lambda\mathtt{x}.\mathtt{e}_0 \quad \mathtt{e}_2 \Downarrow v_2 \quad \mathtt{e}_0[v_2/\mathtt{x}] \Downarrow v}{\mathtt{e}_1\mathtt{@}\mathtt{e}_2 \Downarrow v}$$

Lemma 2. (Determinism) *If* $\mathtt{e} \Downarrow v$ *and* $\mathtt{e} \Downarrow w$ *then* $v = w$.

Lemma 3. *If* `e` *is closed and its λ-variables are all distinct, then a deduction of* $\mathtt{e} \Downarrow v$ *involves no renaming.*

3 Challenges in termination analysis of the λ-calculus

The size-change termination analysis of [5] is based on several concepts including

1. Identifying nontermination as being caused by *infinite sequences of state transitions.*
2. A fixed set of *program control points.*
3. *Observable decreases* in data value sizes.
4. *Construction* of one size-change graph for each function call.
5. Finding the program's entire *control flow graph*, and the call sequences that follow it.

At first sight *all* these concepts seem to be absent from the λ-calculus, except that an application must be a call; and even then, it is not a priori clear *which* function is being called. We will show, one step at a time, that all the concepts do in fact exist in call-by-value λ-calculus evaluation.

3.1 Identifying nontermination (Challenge 1)

We will sometimes write $\mathtt{e} \Downarrow$ to mean $\mathtt{e} \Downarrow v$ for some $v \in ValueS$, and write $\mathtt{e} \not\Downarrow$ to mean there is no $v \in ValueS$ such that $\mathtt{e} \Downarrow v$, i.e., if evaluation of `e` does not terminate.

A proof of $\mathtt{e} \Downarrow v$ is a finite object, and no such proof exists if the computation of `e` fails to terminate. In order to trace an arbitrary computation, terminating or not, we introduce the "calls" relation $\mathtt{e} \to \mathtt{e'}$. The rationale is straightforward: $\mathtt{e} \to \mathtt{e'}$ if in order to deduce $\mathtt{e} \Downarrow v$ for some value v, it is necessary first to deduce $\mathtt{e'} \Downarrow u$ for some u, i.e., some inference rule has form $\frac{\ldots\ \mathtt{e'} \Downarrow ?\ \ldots}{\mathtt{e} \Downarrow ?}$.
Applying this to Definition 5 gives the following.

Definition 6. *The* call relation $\rightarrow \subseteq Exp \times Exp$ *is* $\rightarrow = \underset{r}{\rightarrow} \cup \underset{d}{\rightarrow} \cup \underset{c}{\rightarrow}$ *where* $\underset{r}{\rightarrow}, \underset{d}{\rightarrow}, \underset{c}{\rightarrow}$ *are defined by the following inference rules.*[1]

$$\text{(OperatorS)}\ \frac{}{\mathtt{e_1 @ e_2} \underset{r}{\rightarrow} \mathtt{e_1}} \qquad \text{(OperandS)}\ \frac{\mathtt{e_1} \Downarrow v_1}{\mathtt{e_1 @ e_2} \underset{d}{\rightarrow} \mathtt{e_2}} \qquad \text{(CallS)}\ \frac{\mathtt{e_1} \Downarrow \lambda\mathtt{x.e_0} \quad \mathtt{e_2} \Downarrow v_2}{\mathtt{e_1 @ e_2} \underset{c}{\rightarrow} \mathtt{e_0}[v_2/\mathtt{x}]}$$

As usual, we write $\rightarrow^+$ *for the transitive closure of* $\rightarrow$*, and* $\rightarrow^*$ *for its reflexive transitive closure.*

A familiar example: Call-by-value reduction of the combinator $\Omega = (\lambda\mathtt{x.x@x})\mathtt{@}(\lambda\mathtt{y.y@y})$ yields an infinite call chain:

$$\Omega = (\lambda\mathtt{x.x@x})\mathtt{@}(\lambda\mathtt{y.y@y}) \rightarrow (\lambda\mathtt{y.y@y})\mathtt{@}(\lambda\mathtt{y.y@y}) \rightarrow (\lambda\mathtt{y.y@y})\mathtt{@}(\lambda\mathtt{y.y@y}) \rightarrow \ldots$$

In fact, this linear-call-sequence behaviour is typical of nonterminating computations:

Lemma 4. (*NIS, or* ***N****ontermination* ***I****s* ***S****equential*) *Let* P *be a program. Then* P $\not\Downarrow$ *if and only if there exists an infinite call chain*

$$\mathtt{P} = \mathtt{e_0} \rightarrow \mathtt{e_1} \rightarrow \mathtt{e_2} \rightarrow \ldots$$

Proof See the Appendix. □

Rules (CallS) and (ApplyS) have a certain overlap: $\mathtt{e_2} \Downarrow v_2$ appears in both, as does $\mathtt{e_0}[v_2/\mathtt{x}]$. Thus the (Call) rule can be used as an intermediate step to simplify the (Apply) rule. Variations on the following combined set will be used in the rest of the paper:

Definition 7. *(Combined evaluate and call rules, standard semantics)*

$$\text{(ValueS)}\ \frac{}{v \Downarrow v}\ (\text{If } v \in \mathit{Value})$$

$$\text{(OperatorS)}\ \frac{}{\mathtt{e_1 @ e_2} \underset{r}{\rightarrow} \mathtt{e_1}} \qquad \text{(OperandS)}\ \frac{\mathtt{e_1} \Downarrow v_1}{\mathtt{e_1 @ e_2} \underset{d}{\rightarrow} \mathtt{e_2}}$$

$$\text{(CallS)}\ \frac{\mathtt{e_1} \Downarrow \lambda\mathtt{x.e_0} \quad \mathtt{e_2} \Downarrow v_2}{\mathtt{e_1 @ e_2} \underset{c}{\rightarrow} \mathtt{e_0}[v_2/\mathtt{x}]} \qquad \text{(ApplyS)}\ \frac{\mathtt{e_1 @ e_2} \underset{c}{\rightarrow} \mathtt{e'} \quad \mathtt{e'} \Downarrow v}{\mathtt{e_1 @ e_2} \Downarrow v}$$

3.2 Finitely describing the computation space (Challenge 2)

An equivalent *environment-based* semantics addresses the lack of program control points.

Definition 8. (States, values and environments) *State, Value, Env are the smallest sets such that*

$$\begin{array}{lcll} \mathit{State} & = & \{\ \mathtt{e} : \rho & \mid\ \mathtt{e} \in \mathit{Exp}, \rho \in \mathit{Env}\ \text{and}\ \mathit{fv}(\mathtt{e}) \subseteq \mathit{dom}(\rho)\ \} \\ \mathit{Value} & = & \{\ \lambda\mathtt{x.e} : \rho & \mid\ \lambda\mathtt{x.e} : \rho \in \mathit{State}\ \} \\ \mathit{Env} & = & \{\ \rho : X \rightarrow \mathit{Value} & \mid\ X\ \text{is a finite set of variables}\ \} \end{array}$$

The empty environment with domain $X = \emptyset$ *is written* $[]$*. The evaluation judgement form is* $s \Downarrow v$ *where* $s \in \mathit{State}, v \in \mathit{Value}$*.*

We follow the pattern of Definition 7, except that substitution (β-reduction) $\mathtt{e_0}[v_2/\mathtt{x}]$ of the (ApplyS) rule is replaced by a "lazy substitution" that just updates the environment in the (Call) rule.

[1] Naming: r, d in $\underset{r}{\rightarrow}, \underset{d}{\rightarrow}$ are the last letters of *operato*r and *operan*d, and c in $\underset{c}{\rightarrow}$ stands for "call".

Definition 9. (Environment-based evaluation and call semantics) *The evaluation and call relations* $\Downarrow, \to$ *are defined by the following inference rules, where* $\to \; = \; \underset{r}{\to} \cup \underset{d}{\to} \cup \underset{c}{\to}$.

(Value) $\dfrac{}{v \Downarrow v}$ (If $v \in \mathit{Value}$) (Var) $\dfrac{}{\mathtt{x}\colon \rho \Downarrow \rho(\mathtt{x})}$

(Operator) $\dfrac{}{\mathtt{e}_1 @ \mathtt{e}_2 : \rho \underset{r}{\to} \mathtt{e}_1 : \rho}$ (Operand) $\dfrac{\mathtt{e}_1 : \rho \Downarrow v_1}{\mathtt{e}_1 @ \mathtt{e}_2 : \rho \underset{d}{\to} \mathtt{e}_2 : \rho}$

(Call) $\dfrac{\mathtt{e}_1 : \rho \Downarrow \lambda \mathtt{x}.\mathtt{e}_0 : \rho_0 \qquad \mathtt{e}_2 : \rho \Downarrow v_2}{\mathtt{e}_1 @ \mathtt{e}_2 : \rho \underset{c}{\to} \mathtt{e}_0 : \rho_0[\mathtt{x} \mapsto v_2]}$ (Apply) $\dfrac{\mathtt{e}_1 @ \mathtt{e}_2 : \rho \underset{c}{\to} \mathtt{e}' : \rho' \qquad \mathtt{e}' : \rho' \Downarrow v}{\mathtt{e}_1 @ \mathtt{e}_2 : \rho \Downarrow v}$

Remark: *A tighter version of these rules would "shrink-wrap" the environment* ρ *in a state* $\mathtt{e} : \rho$, *so that* $dom(\rho) = fv(\mathtt{e})$ *(rather than* $\supseteq$*). For instance, the conclusion of* (Operator) *would be*[2] $\mathtt{e}_1 @ \mathtt{e}_2 : \rho \underset{r}{\to} \mathtt{e}_1 : \rho_{|fv(\mathtt{e}_1)}$. *This has no significant effect on computations.*

Following the lines of [9], the environment-based semantics is shown equivalent to the standard semantics in the sense that they have the same termination behaviour, and when evaluation terminates the computed values are related by function $F : \mathit{Exp} \times \mathit{Env} \to \mathit{Exp}$ defined as

$$F(\mathtt{e} : \rho) = \mathtt{e}[F(\rho(\mathtt{x}_1))/\mathtt{x}_1, ..., F(\rho(\mathtt{x}_k))/\mathtt{x}_k] \text{ where } \{\mathtt{x}_1, .., \mathtt{x}_k\} = dom(\rho) \cap fv(e)$$

Lemma 5. $\mathtt{P} : [] \Downarrow v$ *(by Definition 9) if and only if* $\mathtt{P} \Downarrow F(v)$ *(by Definition 5).*

Proof is in the Appendix. The following is proven in the same way as Lemma 4.

Lemma 6. (*NIS, or* **N***ontermination* **I***s* **S***equential*) *Let* $\mathtt{P}$ *be a program. Then* $\mathtt{P} : [] \not\Downarrow$ *if and only if there exists an infinite call chain*

$$\mathtt{P} : [] = \mathtt{e}_0 : \rho_0 \to \mathtt{e}_1 : \rho_1 \to \mathtt{e}_2 : \rho_2 \to \dots$$

3.3 The subexpression property

Definition 10. *Given a state s, we define its* expression support *exp_sup(s) by*

$$\mathit{exp\_sup}(\mathtt{e} : \rho) = \mathit{subexp}(\mathtt{e}) \cup \bigcup_{\mathtt{x} \in fv(\mathtt{e})} \mathit{exp\_sup}(\rho(\mathtt{x}))$$

Lemma 7. (Subexpression property) *If* $s \Downarrow s'$ *or* $s \to s'$ *then* $\mathit{exp\_sup}(s) \supseteq \mathit{exp\_sup}(s')$.

Corollary 1. *If* $\mathtt{P} : [] \Downarrow \lambda \mathtt{x}.\mathtt{e} : \rho$ *then* $\lambda \mathtt{x}.\mathtt{e} \in \mathit{subexp}(\mathtt{P})$.

Proof of Lemma 7 is by induction on the proof of $s \Downarrow v$ or $s \to s'$. Base cases: $s = \mathtt{x} : \rho$ and $s = \lambda \mathtt{x}.\mathtt{e} : \rho$ are immediate. For rule (Call) suppose $\mathtt{e}_1 : \rho \Downarrow \lambda \mathtt{x}.\mathtt{e}_0 : \rho_0$ and $\mathtt{e}_2 : \rho \Downarrow v_2$. By induction

$$\mathit{exp\_sup}(\mathtt{e}_1 : \rho) \supseteq \mathit{exp\_sup}(\lambda \mathtt{x}.\mathtt{e}_0 : \rho_0) \quad \text{and} \quad \mathit{exp\_sup}(\mathtt{e}_2 : \rho) \supseteq \mathit{exp\_sup}(v_2)$$

Thus

$$\begin{array}{rl} \mathit{exp\_sup}(\mathtt{e}_1 @ \mathtt{e}_2 : \rho) & = \mathit{exp\_sup}(\mathtt{e}_1 : \rho) \cup \mathit{exp\_sup}(\mathtt{e}_2 : \rho) \supseteq \\ \mathit{exp\_sup}(\lambda \mathtt{x}.\mathtt{e}_0 : \rho_0) \cup \mathit{exp\_sup}(v_2) & \supseteq \mathit{exp\_sup}(\mathtt{e}_0 : \rho_0[\mathtt{x} \mapsto v_2]) \end{array}$$

For rule (Apply) we have $\mathit{exp\_sup}(\mathtt{e}_1 @ \mathtt{e}_2 : \rho) \supseteq \mathit{exp\_sup}(\mathtt{e}' : \rho') \supseteq \mathit{exp\_sup}(v)$. Cases (Operator), (Operand) are immediate. □

[2] The *restriction* of ρ to a finite set A of variables is the environment $\rho_{|A}$ with domain $A \cap dom(\rho)$ that agrees with ρ on this domain.

3.4 A control point is a subexpression of a λ-program (Challenge 2)

The subexpression property does not hold for the standard rewriting semantics, but it is the starting point for our program analysis: A *control point* will be a subexpression of the program `P` being analysed, and our analyses will bind program flow information to subexpressions of `P`.

By the NIS Lemma 6, if $\mathtt{P} \not\Downarrow$ then there exists an infinite call chain

$$\mathtt{P} : [] = \mathtt{e}_0 : \rho_0 \to \mathtt{e}_1 : \rho_1 \to \mathtt{e}_2 : \rho_2 \to \dots$$

By Lemma 7, $\mathtt{e}_i \in \mathit{subexp}(\mathtt{P})$ for each i. Our termination-detecting algorithm will focus on the *size relations between consecutive environments* ρ_i and ρ_{i+1} in this chain. Since $\mathit{subexp}(\mathtt{P})$ is a finite set, at least one subexpression `e` occurs infinitely often, so "self-loops" will be of particular interest.

Example 2. Figure 2 shows the combinator $\Omega = (\lambda\mathtt{x}.\mathtt{x@x})\mathtt{@}(\lambda\mathtt{y}.\mathtt{y@y})$ as a tree whose subexpressions are labeled by numbers. To its right is the "calls" relation $\to$. It has an infinite call chain:

$$\Omega : [] \to \mathtt{x@x} : \rho_1 \to \mathtt{y@y} : \rho_2 \to \mathtt{y@y} : \rho_2 \to \mathtt{y@y} : \rho_2 \to \dots$$

or $\mathtt{1} : [] \to \mathtt{3} : \rho_1 \to \mathtt{7} : \rho_2 \to \mathtt{7} : \rho_2 \to \dots$ where $\rho_1 = [\mathtt{x} \mapsto \lambda\mathtt{y}.\mathtt{y@y} : []]$ and $\rho_2 = [\mathtt{y} \mapsto \lambda\mathtt{y}.\mathtt{y@y} : []]$

The set of states reachable from $\mathtt{P} : []$ is finite, so this computation enters a "repetitive loop."

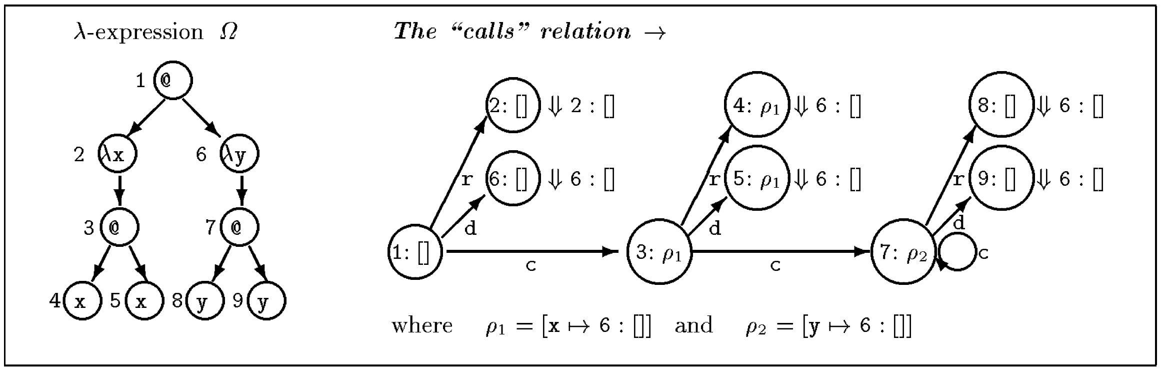


Figure 2: A λ-expression and its call relation

It is also possible that a computation will reach infinitely many states that are all different. Since all states have an expression component lying in a set of fixed size, and each expression in the environment also lies in this finite set, in an infinite state set $\mathcal{S}$ there will be states whose *environment depths* are arbitrarily large.

4 Size-change analysis of the untyped λ-calculus

Our end goal, given program `P`, is correctly to assert the nonexistence of infinite call chains starting at $\mathtt{P} : []$. Our means are size-change graphs G that "safely" describe the relation between states s_1 and s_2 in a call $s_1 \to s_2$ or an evaluation $s_1 \Downarrow s_2$. First, we develop a "value decrease" notion suitable for the untyped λ-calculus. This is essentially the "height" of a closure value $\mathtt{e} : \rho$.

4.1 Size changes in a computation (Challenge 3)

The *support* of a state $s = \mathtt{e} : \rho$ is given by

$$\mathit{support}(\mathtt{e} : \rho) = \{\mathtt{e} : \rho\} \cup \bigcup_{\mathtt{x} \in \mathit{fv}(\mathtt{e})} \mathit{support}(\rho(\mathtt{x}))$$

Definition 11. *Relations $s_1 \succeq s_2$ and $s_1 \succ s_2$.*

- *$s_1 \succeq s_2$ holds if support$(s_1) \ni s_2$;*
- *$s_1 \succeq s_2$ holds if $s_1 = \mathtt{e}_1 : \rho_1$ and $s_2 = \mathtt{e}_2 : \rho_2$, where subexp$(\mathtt{e}_1) \ni \mathtt{e}_2$ and $\rho_1(\mathtt{x}) = \rho_2(\mathtt{x})$ for all $\mathtt{x} \in fv(\mathtt{e}_2)$; and*
- *$s_1 \succ s_2$ holds if $s_1 \succeq s_2$ and $s_1 \neq s_2$.*

Sets *support*$(\mathtt{e} : \rho)$ and *subexp*$(\mathtt{e})$ are clearly finite, yielding:

Lemma 8. *The relation $\succ \subseteq State \times State$ is well-founded.*

Definition 12. *We now relate states to the components of a size-change graph:*

1. *The* graph basis *of a state $s = \mathtt{e} : \rho$ is $gb(s) = fv(\mathtt{e}) \cup \{\bullet\}$.*
2. *Suppose $s_1 = \mathtt{e}_1 : \rho_1$ and $s_2 = \mathtt{e}_2 : \rho_2$. A size-change graph G relating the pair (s_1, s_2) has* source *$gb(s_1)$ and* target *$gb(s_2)$.*
3. *We abbreviate $(fv(\mathtt{e}_1) \cup \{\bullet\}) \xrightarrow{G} (fv(\mathtt{e}_2) \cup \{\bullet\})$ to $\mathtt{e}_1 \xrightarrow{G} \mathtt{e}_2$.*
4. *The* valuation function *$\overline{s} : gb(s) \to Value$ of a state s is defined by:*

$$\overline{s}(\bullet) = s \quad \text{and} \quad \overline{\mathtt{e} : \rho}(\mathtt{x}) = \rho(\mathtt{x})$$

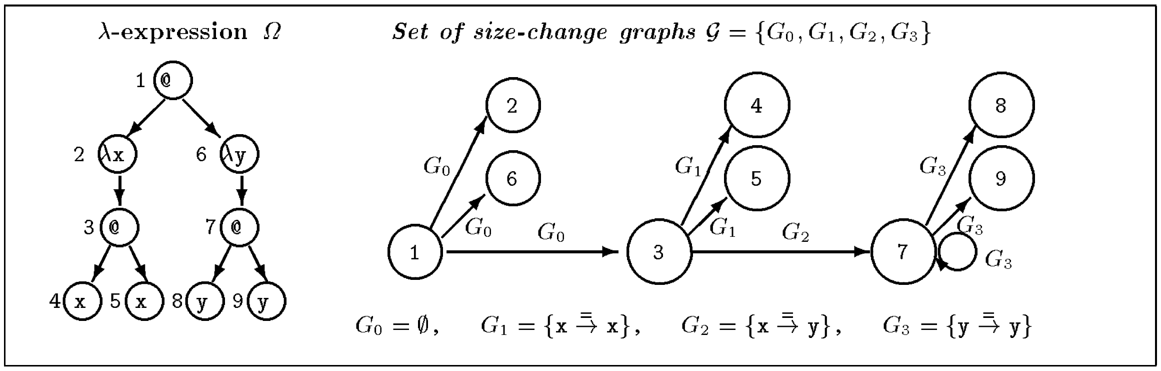
Definition 13. *Let $s_1 = \mathtt{e}_1 : \rho_1$ and $s_2 = \mathtt{e}_2 : \rho_2$. Size-change graph $\mathtt{e}_1 \xrightarrow{G} \mathtt{e}_2$ is* safe[3] *for (s_1, s_2) if*

$$a_1 \xrightarrow{=} a_2 \in G \;\text{ implies }\; \overline{s_1}(a_1) = \overline{s_2}(a_2) \;\text{ and }\; a_1 \xrightarrow{\downarrow} a_2 \in G \;\text{ implies }\; \overline{s_1}(a_1) \succ \overline{s_2}(a_2)$$

Explanation: an arc $\bullet \xrightarrow{r} \bullet$ in G r-relates the two states $s_1 = \mathtt{e}_1 : \rho_1$ and $s_2 = \mathtt{e}_2 : \rho_2$. Relation $r \in \{=, \downarrow\}$ corresponds to $s_1 = s_2$ or $s_1 \succ s_2$ as in Definition 11. Further, an arc $\mathtt{x} \xrightarrow{r} \mathtt{y}$ similarly relates the value of $\mathtt{x}$ in environment ρ_1 to the value of $\mathtt{y}$ in environment ρ_2. An arc $\bullet \xrightarrow{r} \mathtt{y}$ in G r-relates the state $s_1 = \mathtt{e}_1 : \rho_1$ to the value of $\mathtt{y}$ in environment ρ_2, and an arc $\mathtt{x} \xrightarrow{r} \bullet$ similarly relates the value of $\mathtt{x}$ in environment ρ_1 to the state $s_2 = \mathtt{e}_2 : \rho_2$.

Definition 14. *A set $\mathcal{G}$ of size-change graphs is* safe for program $\mathtt{P}$ *if $\mathtt{P} : [] \to^* s_1 \to s_2$ implies some $G \in \mathcal{G}$ is safe for the pair (s_1, s_2).*

Example 3. Figure 3 shows a graph set $\mathcal{G}$ that is safe for program $\Omega = (\lambda \mathtt{x}.\mathtt{x@x})(\lambda \mathtt{y}.\mathtt{y@y})$. For brevity, each subexpression of Ω is referred to by number in the diagram of $\mathcal{G}$. Subexpression $\mathtt{1} = \Omega$ has no free variables, so arcs from node $\mathtt{1}$ are labeled with size-change graphs $G_0 = \emptyset$.



λ-expression Ω
Set of size-change graphs 𝒢 = {G0, G1, G2, G3}
1 @
2 λx
6 λy
3 @
7 @
4 x
5 x
8 y
9 y
G0
G1
G2
G3
G0 = ∅, G1 = {x =→ x}, G2 = {x =→ y}, G3 = {y =→ y}

Figure 3: A set of size-change graphs that safely describes Ω's nonterminating computation.

[3] The term "safe" comes from abstract interpretation [3]. An alternative would be "sound."

Theorem 1. *If $\mathcal{G}$ is safe for program* $\mathtt{P}$ *and satisfies the size-change condition, then call-by-value evaluation of* $\mathtt{P}$ *terminates.*

Proof Suppose call-by-value-evaluation of $\mathtt{P}$ does not terminate. Then by Lemma 6 there is an infinite call chain

$$\mathtt{P} : [] = \mathtt{e}_0 : \rho_0 \to \mathtt{e}_1 : \rho_1 \to \mathtt{e}_2 : \rho_2 \to \ldots$$

Letting $s_i = \mathtt{e}_i : \rho_i$, by safety of $\mathcal{G}$ (Definition 14), there is a size-change graph $G_i \in \mathcal{G}$ that safely describes each pair (s_i, s_{i+1}). By the size-change condition (Definition 3) the multipath $\mathcal{M} = G_0, G_1, \ldots$ has an infinite thread $t = a_j \overset{r_j}{\to} a_{j+1} \overset{r_{j+1}}{\to} \ldots$ such that $k \geq j$ implies $a_k \overset{r_k}{\to} a_{k+1} \in G_k$, and each r_k is $\downarrow$ or $=$, and there are infinitely many $r_k = \downarrow$. Consider the value sequence $\overline{s_j}(a_j), \overline{s_{j+1}}(a_{j+1}), \ldots$. By safety of G_k (Definition 13) we have $\overline{s_k}(a_k) \succeq \overline{s_{k+1}}(a_{k+1})$ for every $k \geq j$, and infinitely many proper decreases $\overline{s_k}(a_k) \succ \overline{s_{k+1}}(a_{k+1})$. However this is impossible since by Lemma 8 the relation $\succ$ on *Value* is well-founded.

Conclusion: call-by-value-evaluation of $\mathtt{P}$ terminates. □

The goal is partly achieved: We have found a sufficient condition on a set of size-change graphs to guarantee program termination. What we have not yet done is to find an algorithm to *construct* a size-change graph set $\mathcal{G}$ that is safe for $\mathtt{P}$ (The safety condition of Definition 14 is in general undecidable, so enumeration of all graphs won't work.) Our graph construction algorithm is developed in two steps:

- First, the exact evaluation and call relations are "instrumented" so as to produce safe size-change graphs during evaluation.
- Second, an *abstract interpretation* of these rules yields a computable over-approximation $\mathcal{G}$ that contains all graphs that can be built during exact evaluation.

4.2 Safely describing value flows in a single computation (Challenge 4)

First, the exact evaluation and call relations are "instrumented" so as to produce safe size-change graphs during evaluation.

Definition 15. (Evaluation and call with graph generation) *The extended evaluation and call judgement forms are* $\mathtt{e} : \rho \to \mathtt{e}' : \rho', G$ *and* $\mathtt{e} : \rho \Downarrow \mathtt{e}' : \rho', G$. *The inference rules are:*

(ValueG) $$\frac{}{\lambda\mathtt{x}.\mathtt{e} : \rho \Downarrow \lambda\mathtt{x}.\mathtt{e} : \rho, id^{=}_{\lambda\mathtt{x}.\mathtt{e}}}$$

(VarG) $$\frac{}{\mathtt{x} : \rho \Downarrow \rho(\mathtt{x}), \{\mathtt{x} \overset{=}{\to} \bullet\} \cup \{\mathtt{x} \overset{\downarrow}{\to} \mathtt{y} \mid \mathtt{y} \in fv(\mathtt{e}')\}} \quad \text{(If } \rho(\mathtt{x}) = \mathtt{e}' : \rho')$$

(OperatorG) $$\frac{}{\mathtt{e}_1@\mathtt{e}_2 : \rho \underset{r}{\to} \mathtt{e}_1 : \rho, id^{\downarrow}_{\mathtt{e}_1}}$$

(OperandG) $$\frac{\mathtt{e}_1 : \rho \Downarrow v_1}{\mathtt{e}_1@\mathtt{e}_2 : \rho \underset{d}{\to} \mathtt{e}_2 : \rho, id^{\downarrow}_{\mathtt{e}_2}}$$

(CallG) $$\frac{\mathtt{e}_1 : \rho \Downarrow \lambda\mathtt{x}.\mathtt{e}_0 : \rho_0, G_1 \qquad \mathtt{e}_2 : \rho \Downarrow v_2, G_2}{\mathtt{e}_1@\mathtt{e}_2 : \rho \underset{c}{\to} \mathtt{e}_0 : \rho_0[\mathtt{x} \mapsto v_2], G_1^{-\bullet} \cup G_2^{\bullet\mapsto\mathtt{x}}}$$

(ApplyG) $$\frac{\mathtt{e}_1@\mathtt{e}_2 : \rho \underset{c}{\to} \mathtt{e}' : \rho', G' \qquad \mathtt{e}' : \rho' \Downarrow v, G}{\mathtt{e}_1@\mathtt{e}_2 : \rho \Downarrow v, (G'; G)}$$

Notations used in the rules ($\mathtt{x}$, $\mathtt{y}$, $\mathtt{z}$ *are variables and not* $\bullet$*):*

$id^{=}_{\mathtt{e}}$	*stands for* $\{\bullet \overset{=}{\rightarrow} \bullet\} \cup \{\mathtt{x} \overset{=}{\rightarrow} \mathtt{x} \mid \mathtt{x} \in fv(\mathtt{e})\}$
$id^{\downarrow}_{\mathtt{e}}$	*stands for* $\{\bullet \overset{\downarrow}{\rightarrow} \bullet\} \cup \{\mathtt{x} \overset{=}{\rightarrow} \mathtt{x} \mid \mathtt{x} \in fv(\mathtt{e})\}$
$G_1^{-\bullet}$	*stands for* $\{\ \mathtt{y} \overset{r}{\rightarrow} \mathtt{z} \mid\ \mathtt{y} \overset{r}{\rightarrow} \mathtt{z} \in G_1\} \cup \{\ \bullet \overset{\downarrow}{\rightarrow} \mathtt{z} \mid\ \bullet \overset{r}{\rightarrow} \mathtt{z} \in G_1\}$
$G_2^{\bullet \mapsto \mathtt{x}}$	*stands for* $\{\ \mathtt{y} \overset{r}{\rightarrow} \mathtt{x} \mid\ \mathtt{y} \overset{r}{\rightarrow} \bullet \in G_2\ \} \cup \{\ \bullet \overset{\downarrow}{\rightarrow} \mathtt{x} \mid\ \bullet \overset{r}{\rightarrow} \bullet \in G_2\ \}$

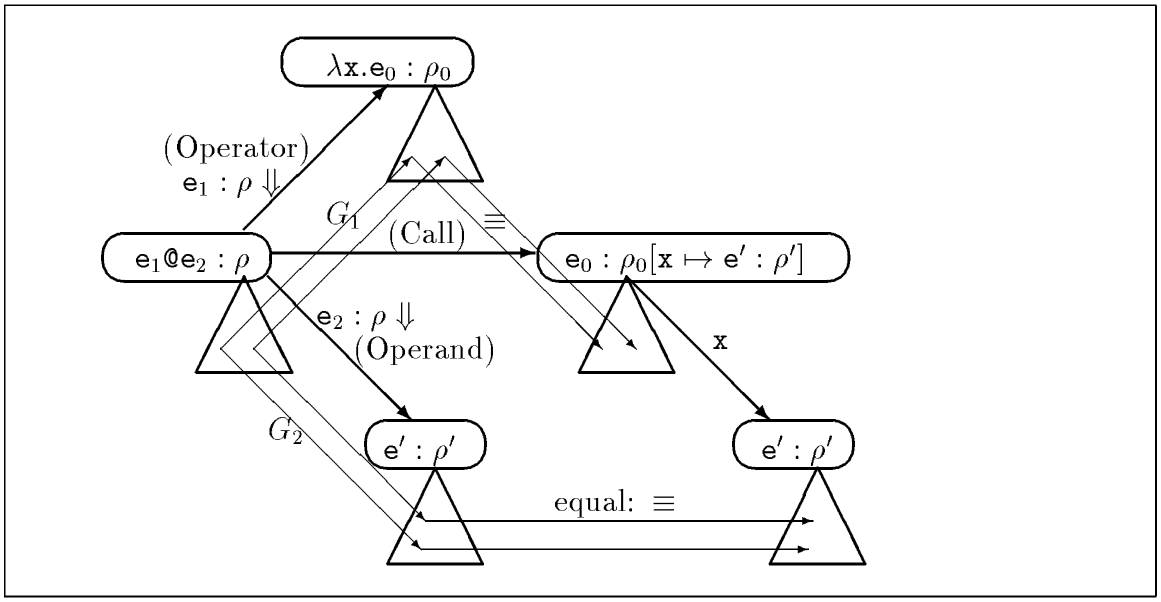


Figure 4: Data-flow in an application

The diagram of Figure 4 may be of some use in visualising data-flow during evaluation of $\mathtt{e}_1 @ \mathtt{e}_2$. States are in ovals and triangles represent environments. In the application $\mathtt{e}_1 @ \mathtt{e}_2 : \rho$ on the left, operator $\mathtt{e}_1 : \rho$ evaluates to $\lambda \mathtt{x}.\mathtt{e}_0 : \rho_0, G_1$ and operand $\mathtt{e}_2 : \rho$ evaluates to $\mathtt{e}' : \rho', G_2$. The size change graphs G_1 and G_2 show relations between variables bound in their environments. There is a call from the application $\mathtt{e}_1 @ \mathtt{e}_2 : \rho$ to $\mathtt{e}_0 : \rho_0[\mathtt{x} \mapsto \mathtt{e}' : \rho']$ the body of the operator-value with the environment extended with a binding of $\mathtt{x}$ to the operand-value $\mathtt{e}' : \rho'$.

Lemma 9. $s \rightarrow s', G$ (by Definition 15) *iff* $s \rightarrow s'$ (by Definition 7). *Further,* $s \Downarrow s', G$ *iff* $s \Downarrow s'$.

Theorem 2. (The extracted graphs are safe) $s \rightarrow s', G$ *or* $s \Downarrow s', G$ *implies* G *is safe for* (s, s').

Proof The Lemma is immediate since the new rules extend the old, without any restriction on their applicability. Proof of "safety" is by a case analysis deferred to the Appendix. □

4.3 Abstract interpretation of the "Calls" and "Evaluates-to" relations (Challenge 5)

A coarse approximation may be obtained by removing all environment components. To deal with the absence of environments the variable lookup rule is modified: If $\mathtt{e}_1 @ \mathtt{e}_2$ is *any* application in $\mathtt{P}$ such that $\mathtt{e}_1$ can evaluate to a value of form $\lambda \mathtt{x}.\mathtt{e}$ and $\mathtt{e}_2$ can evaluate to value v_2, then v_2 is regarded as a possible value of $\mathtt{x}$.

The main virtue of these rules is that there are only finitely many possible judgements $\mathtt{e} \rightarrow \mathtt{e}'$ and $\mathtt{e} \Downarrow \mathtt{e}'$. Consequently, the runtime behavior of program $\mathtt{P}$ may be (approximately) analysed by exhaustively applying these inference rules. The next section will extend the rules so they also generate size-change graphs.

Definition 16. (Approximate evaluation and call rules) *The new judgement forms are* $\mathtt{e} \Downarrow \mathtt{e}'$ *and* $\mathtt{e} \rightarrow \mathtt{e}'$. *The inference rules are:*

$$\text{(ValueA)}\ \frac{}{\lambda\mathtt{x}.\mathtt{e} \Downarrow \lambda\mathtt{x}.\mathtt{e}} \qquad \text{(VarA)}\ \frac{\mathtt{e}_1 @ \mathtt{e}_2 \in \mathit{subexp}(\mathtt{P}) \quad \mathtt{e}_1 \Downarrow \lambda\mathtt{x}.\mathtt{e}_0 \quad \mathtt{e}_2 \Downarrow v_2}{\mathtt{x} \Downarrow v_2}$$

$$\text{(OperatorA)}\ \frac{}{\mathtt{e}_1 @ \mathtt{e}_2 \underset{r}{\rightarrow} \mathtt{e}_1} \qquad \text{(OperandA)}\ \frac{}{\mathtt{e}_1 @ \mathtt{e}_2 \underset{d}{\rightarrow} \mathtt{e}_2}$$

$$\text{(CallA)}\ \frac{\mathtt{e}_1 \Downarrow \lambda\mathtt{x}.\mathtt{e}_0 \quad \mathtt{e}_2 \Downarrow v_2}{\mathtt{e}_1 @ \mathtt{e}_2 \underset{c}{\rightarrow} \mathtt{e}_0} \qquad \text{(ApplyA)}\ \frac{\mathtt{e}_1 @ \mathtt{e}_2 \underset{c}{\rightarrow} \mathtt{e}' \quad \mathtt{e}' \Downarrow v}{\mathtt{e}_1 @ \mathtt{e}_2 \Downarrow v}$$

Remark: the (VarA) rule refers globally to P, the program being analysed. Notice that the approximate evaluation is nondeterministic: An expression may evaluate to more than one value.

Lemma 10. *Suppose* $\mathtt{P} : [] \rightarrow^* \mathtt{e} : \rho$. *If* $\mathtt{e} : \rho \Downarrow \mathtt{e}' : \rho'$ *then* $\mathtt{e} \Downarrow \mathtt{e}'$. *Further, if* $\mathtt{e} : \rho \rightarrow \mathtt{e}' : \rho'$ *then* $\mathtt{e} \rightarrow \mathtt{e}'$.

Proof Straightforward; see the Appendix. □

4.4 Construction of size-change graphs by abstract interpretation

We now extend the coarse approximation to construct size-change graphs.

Definition 17. (Approximate evaluation and call with graph generation) *The judgement forms are now* $\mathtt{e} \rightarrow \mathtt{e}', G$ *and* $\mathtt{e} \Downarrow \mathtt{e}', G$.

$$\text{(ValueAG)}\ \frac{}{\lambda\mathtt{x}.\mathtt{e} \Downarrow \lambda\mathtt{x}.\mathtt{e},\, \mathit{id}^{=}_{\lambda\mathtt{x}.\mathtt{e}}} \qquad \text{(VarAG)}\ \frac{\mathtt{e}_1 @ \mathtt{e}_2 \in \mathit{subexp}(\mathtt{P}) \quad \mathtt{e}_1 \Downarrow \lambda\mathtt{x}.\mathtt{e}_0, G_1 \quad \mathtt{e}_2 \Downarrow v_2, G_2}{\mathtt{x} \Downarrow v_2, \{\mathtt{x} \overset{=}{\rightarrow} \bullet\} \cup \{\mathtt{x} \overset{\downarrow}{\rightarrow} \mathtt{y} \mid \mathtt{y} \in \mathit{fv}(v_2)\}}$$

$$\text{(OperatorAG)}\ \frac{}{\mathtt{e}_1 @ \mathtt{e}_2 \underset{r}{\rightarrow} \mathtt{e}_1,\, \mathit{id}^{\downarrow}_{\mathtt{e}_1}} \qquad \text{(OperandAG)}\ \frac{}{\mathtt{e}_1 @ \mathtt{e}_2 \underset{d}{\rightarrow} \mathtt{e}_2,\, \mathit{id}^{\downarrow}_{\mathtt{e}_2}}$$

$$\text{(CallAG)}\ \frac{\mathtt{e}_1 \Downarrow \lambda\mathtt{x}.\mathtt{e}_0, G_1 \quad \mathtt{e}_2 \Downarrow v_2, G_2}{\mathtt{e}_1 @ \mathtt{e}_2 \underset{c}{\rightarrow} \mathtt{e}_0, G_1^{-\bullet} \cup G_2^{\bullet \mapsto \mathtt{x}}} \qquad \text{(ApplyAG)}\ \frac{\mathtt{e}_1 @ \mathtt{e}_2 \underset{c}{\rightarrow} \mathtt{e}', G' \quad \mathtt{e}' \Downarrow v, G}{\mathtt{e}_1 @ \mathtt{e}_2 \Downarrow v, G'; G}$$

Lemma 11. *Suppose* $\mathtt{P} : [] \rightarrow^* \mathtt{e} : \rho$. *If* $\mathtt{e} : \rho \rightarrow \mathtt{e}' : \rho', G$ *then* $\mathtt{e} \rightarrow \mathtt{e}', G$. *Further, if* $\mathtt{e} : \rho \Downarrow \mathtt{e}' : \rho', G$ *then* $\mathtt{e} \Downarrow \mathtt{e}', G$.

Proof Straightforward; see the Appendix. □

Definition 18.

$$\mathit{absint}(\mathtt{P}) = \{\ G_j \mid j > 0 \wedge \exists \mathtt{e}_i, G_i (0 \leq i \leq j) : \mathtt{P} = \mathtt{e}_0 \wedge (\mathtt{e}_0 \rightarrow \mathtt{e}_1, G_1) \wedge \ldots \wedge (\mathtt{e}_{j-1} \rightarrow \mathtt{e}_j, G_j)\ \}$$

Theorem 3.

1. *The set* $\mathit{absint}(\mathtt{P})$ *is safe for* P.
2. *The set* $\mathit{absint}(\mathtt{P})$ *can be effectively computed from* P.

Proof Part 1: Suppose $\mathtt{P} : [] = s_0 \rightarrow s_1 \rightarrow \ldots \rightarrow s_j$. Theorem 2 implies $s_i \rightarrow s_{i+1}, G_i$ where each G_i is safe for the pair (s_i, s_{i+1}). Let $s_i = \mathtt{e}_i : \rho_i$. By Lemma 11, $\mathtt{e}_i \rightarrow \mathtt{e}_{i+1}, G_{i+1}$. By the definition of $\mathit{absint}(\mathtt{P})$, $G_j \in \mathit{absint}(\mathtt{P})$.

Part 2: There is only a fixed number of subexpressions of P, or of possible size-change graphs. Thus $\mathit{absint}(\mathtt{P})$ can be computed by applying Definition 17 exhaustively, starting with P, until no new graphs or subexpressions are obtained. □

5 Examples and experimental results

5.1 Simple example

Using Church numerals ($\texttt{n} = \lambda\texttt{s}\lambda\texttt{z}.\texttt{s}^n(\texttt{z})$), we expect `2 succ 0` to reduce to `succ(succ 0)`. However this contains unreduced redexes because call-by-value does not reduce under a λ, so we force the computation to carry on through by applying `2 succ 0` to the identity (twice). This gives:

```
2 succ 0 id1 id2 where
  succ = λm.λs.λz. m s (s z)
  id1  = λx.x
  id2  = λy.y
```

After writing this out in full as a λ-expression, our analyser yields (syntactically sugared):

```
  [λs2.λz2.(s2 @ (s2 @ z2))]      -- two --
@ [λm.λs.λz. 15: ((m@s)@(s@z))] -- succ --
@ [λs1.λz1.z1]                  -- zero --
@ [λx.x]                        -- id1 --
@ [λy.y]                        -- id2 --

   Output of loops from an analysis of this program:

   15→* 15: [(m,>,m),(s,=,s),(z,=,z)], []

   Size Change Termination: Yes
```

The loop occurs because application of `2` forces the code for the successor function to be executed twice, with decreasing argument values `m`. The notation for edges is a little different from previously, here `(m,>,m)` stands for $\texttt{m} \overset{\downarrow}{\to} \texttt{m}$.

5.2 $fnx = x + 2^n$ by Church numerals

This more interesting program computes $fnx = x + 2^n$ by higher-order primitive recursion. If `n` is a Church numeral then expression `n g x` reduces to $\texttt{g}^n(\texttt{x})$. Let `x` be the successor function, and `g` be a "double application" functional. Expressed in a readable named combinator form, we get:

```
f n x    where
f n   =  if n=0 then succ else g(f(n-1))
g r a =  r(ra)
```

As a lambda-expression (applied to values $\texttt{n} = 3, \texttt{x} = 4$) this can be written:

```
[λn.λx. n                                   -- n --
        @ [λr.λa. 11: (r@ 13: (r@a))]       -- g --
        @ [λ k.λ s.λ z.(s@((k@s)@z))]       - succ-
        @ x ]                               -- x --

@         [λs2.λz2. (s2@(s2@(s2@z2))) ]     -- 3 --
@         [λs1.λz1. (s1@(s1@(s1@(s1@z1))))] -- 4 --
```

Following is the output from program analysis. The analysis found the following loops from a program point to itself with the associated size change graph and path. The first number refer to the program point, then comes a list of edges and last a list of numbers, the other program points that the loop passes through.

```
SELF Size Change Graphs, no repetition of graphs:

11 →* 11: [(r,>,r)]             []
11 →* 11: [(a,=,a),(r,>,r)]     [13]
13 →* 13: [(a,=,a),(r,>,r)]     [11]
13 →* 13: [(r,>,r)]             [11,11]

Size Change Termination: Yes
```

5.3 Ackermann's function, second-order

This can be written without recursion using Church numerals as: `a m n` where `a = λm. m b succ` and `b = λg. λn. n g (g 1)`. Consequently `a m =` $\mathtt{b}^{\mathtt{m}}$`(succ)` and `b g n =` $\mathtt{g}^{\mathtt{n+1}}$`(1)`, which can be seen to agree with the usual first-order definition of Ackermann's function. Following is the same as a lambda-expression applied to argument values `m=2, n=3`, with numeric labels on some subexpressions.

```
(λm.m b succ) 2 3  =  (λm.m@b@succ)@2@3
(λm.m@(λg.λn.n@g@(g@1))@succ)@2@3
(λm.m@(λg.λn.9: (n@g@13: (g@1)))@succ)@2@3
 where
 1    =  λs1.λz1. 17: (s1@z1)
succ  =  λk.λs.λz. 23: (s@25: (k@s@z))
 2    =  λs2.λz2. s2@(s2@z2)
 3    =  λs3.λz3. 39: (s3@41: (s3@43: (s3@z3)))
```

Output from an analysis of this program is shown here.

```
SELF Size Change Graphs, no repetition of graphs:
(Because graphs are only taken once it is not always the case that the
same loop is shown for all program points in its path)

 9 →*  9: [(•,>,n),(g,>,g)]             [13]
 9 →*  9: [(g,>,g)]                     [17]
13 →* 13: [(g,>,g)]                     [9]
17 →* 17: [(s1,>,s1)]                   [9]
23 →* 23: [(k,>,k),(s,=,s),(z,=,z)]     [25]
23 →* 23: [(s,>,s)]                     [9]
23 →* 23: [(s,>,s),(z,>,k)]             [25,17,9]
25 →* 25: [(k,>,k),(s,=,s),(z,=,z)]     [23]
25 →* 25: [(s,>,s),(z,>,k)]             [17,9,23]
25 →* 25: [(s,>,s)]                     [23,9,23]
39 →* 39: [(s3,>,s3)]                   [9]
41 →* 41: [(s3,>,s3)]                   [9,39]
43 →* 43: [(s3,>,s3)]                   [9,39,41]

Size Change Termination: Yes
```

5.4 A minimum function, with general recursion and Y-combinator

We have another version of the termination analysis where programs can have as constants: natural numbers, predecessor, successor and zero-test, and also if-then-else expressions. In this setting it is possible to analyse the termination behaviour of a program with arbitrary natural number as input represented by •. This program computes the minimum of its two inputs using the call-by-value combinator $Y = \lambda$`p. [λq.p@(λs.q@q@s)] @ [λt.p@(λu.t@t@u)]`. The program, first as a first-order recursive definition.

```
        m x y = if x=0 then 0 else if y=0 then 0 else succ (m (pred x) (pred y))
```

Now, in λ-expression form for analysis.

```
   {λp. [λq.p@(λs.q@q@s)] @ [λt.p@(λu.t@t@u)]}  -- the Y combinator --
   @
   [ λm.λx.λy. |27:|if( (ztst @ x),
                      0,
                     |32:|   if( (ztst @ y),
                                0,
                               |37:|succ @ |39:| m @ (pred@x) @ (pred@y) ]
   @ •
   @ •
```

```
    Output of loops from an analysis of this program:

    27 →* 27: [(x,>,x),(y,>,y)]    [32,37,39]
    32 →* 32: [(x,>,x),(y,>,y)]    [37,39,27]
    37 →* 37: [(x,>,x),(y,>,y)]    [39,27,32]
    39 →* 39: [(x,>,x),(y,>,y)]    [27,32,37]

    Size Change Termination: Yes
```

5.5 Ackermann's function, second-order with constants and Y-combinator

Ackermann's function can be written as: `a m n` where `a m = b`^{m}`(suc)` and `b g n = g`$^{n+1}$`(1)`. The following program expresses the computations of both `a` and `b` by loops, using the Y combinator (twice).

```
   [λ y.λ y1.
   (y1 @
   λ a.λ m. |11:|if( (ztst@m),
                   λ v.(suc@v),
                  |19:|( (y @
                        λ b.λ f.λ n.
                       |25:|if( (ztst@n),
                              |29:|(f@1),
                              |32:|f@|34:|b @ f @ (pred@n))
                  @ |41:| a @ (pred@m)                     ]

   @ {λp. [λq.p@(λs. q@q@s)] @ [λt.p@(λu. t@t@u)]}
   @ {λp1. [λq1.p1@(λs. |72:|q1@1q@s1)] @ [λt1.p1@(λu1. |81:|t1@t1@u1)]}
   @ •
   @ •
```

```
    Output of loops from an analysis of this program:

    SELF Size Change Graphs no repetition of graphs:
```

```
11 →* 11: [(a,>,y),(m,>,m)]      [19,41,72]
11 →* 11: [(m,>,m)]              [19,41,72,11,19,41,72]
19 →* 19: [(a,>,y),(m,>,m)]      [41,72,11]
19 →* 19: [(m,>,m)]              [41,72,11,19,41,72,11]
25 →* 25: [(f,>,b),(f,>,f)]      [29]
25 →* 25: [(f,=,f),(n,>,n)]      [32,34]
25 →* 25: [(f,>,f)]              [29,25,32,34]
29 →* 29: [(f,>,f)]              [25]
32 →* 32: [(f,>,b),(f,>,f)]      [25]
32 →* 32: [(f,=,f),(n,>,n)]      [34,25]
32 →* 32: [(f,>,f)]              [25,32,34,25]
34 →* 34: [(f,=,f),(n,>,n)]      [25,32]
34 →* 34: [(f,>,b),(f,>,f)]      [25,29,25,32]
34 →* 34: [(f,>,f)]              [25,29,25,32,34,25,32]
41 →* 41: [(m,>,m)]              [72,11,19]
72 →* 72: [(s1,>,s1)]            [11,19,41]
81 →* 81: [(u1,>,u1)]            [11,19,41]

Size Change Termination: Yes
```

5.6 Imprecision of abstract interpretation

It is natural to wonder whether the gross approximation of Definition 16 comes at a cost. The (VarA) rule can in effect “mix up” different function applications, losing the coordination between operator and operand that is present in the exact semantics.

We have observed this in practice: The first time we had programmed Ackermann’s using explicit recursion, we used the same instance of Y-combinator for both loops, so the single Y-combinator expression was “shared”. The analysis did not discover that the program terminated.

However when this was replaced by the “unshared” version above, with two instances of the Y-combinator (`y` and `y1`) (one for each application), the problem disappeared and termination was correctly recognised.

6 Concluding matters

Acknowledgements: The authors gratefully acknowledge detailed and constructive comments by Arne Glenstrup and Chin Soon Lee, and insightful comments by Luke Ong and David Wahlstedt.

Related work: Papers subsequent to [5] have used size-change graphs to find bounds on program running times [1]; solved related problems, e.g., to ensure that partial evaluation will terminate [4, 6]; and found more efficient (though less precise) algorithms [7]. Further, the thesis [8] extends the first-order size-change method [5] to handle higher-order named combinator programs. It uses a different approach than ours, and appears to be less general.

It is natural to compare the size-change condition with strongly normalising subsets of the λ-calculus, in particular subsets that are typable by various disciplines, ranging from simple types up to and including Girard’s System F [2].

A notable difference is that general recursion better matches the habits of the working computer scientist: primitive recursion is unnatural for programming, and even more so when higher types are used. On the other hand, the class of functions computable by typable λ-expressions is enormous, and precisely characterised. Conclusion: more investigations need to be done.

Future work

1. Extend the analysis so it can recognise, given a program `P`, that `P @ v` will terminate for any choice of `v` from a given data set, e.g., Church or other numerals. This seems straightforward.
2. Prove or disprove **Conjecture:** The size-change method will recognise as terminating any simply typed λ-expression (sans types).
3. Prove or disprove **Conjecture:** The size-change method will recognise as terminating any System T expression (sans types, and coded using Church numerals).

Appendix

A Proof of Lemma 4

Proof *If:* Consider a proof tree of $\mathtt{P} \Downarrow v$. Each call rule of Definition 6 is associated with a use of rule (ApplyS) from Definition 5. There will be a call $\mathtt{e} \to \mathtt{e}'$ from each (ApplyS) conclusion $\mathtt{e} \Downarrow v$ to one of its three premises $\mathtt{e}' \Downarrow v'$ in the proof tree, and only these. Since the proof tree is finite, there will be no infinite call chains. (An inductive proof is omitted for brevity.)

Only if: Assume $\mathtt{P} \to^* \mathtt{e}$ and all call chains from $\mathtt{e}$ are finite. We prove by induction on the maximal length n of a call chain from $\mathtt{e}$ that $\mathtt{e} \Downarrow$.

$n = 0$: $\mathtt{e}$ is an abstraction that evaluates to itself.

$n > 0$: $\mathtt{e}$ must be an application $\mathtt{e} = \mathtt{e}_1 @ \mathtt{e}_2$. By rule (OperatorS) there is a call $\mathtt{e}_1 @ \mathtt{e}_2 \underset{d}{\to} \mathtt{e}_1$, and the maximal length of a call chain from $\mathtt{e}_1$ is less than n. By induction there exists v_1 such that $\mathtt{e}_1 \Downarrow v_1$. We now conclude by rule (OperandS) that $\mathtt{e}_1 @ \mathtt{e}_2 \underset{r}{\to} \mathtt{e}_2$. By induction there exists v_2 such that $\mathtt{e}_2 \Downarrow v_2$.

All values are abstractions, so we can write $v_1 = \lambda \mathtt{x}.\mathtt{e}_0$. We now conclude by rule (CallS) that $\mathtt{e}_1 @ \mathtt{e}_2 \underset{c}{\to} \mathtt{e}_0[\mathtt{x} := v_2]$. By induction again, $\mathtt{e}_0[\mathtt{x} \mapsto v_2] \Downarrow v$ for some v. This gives us all premises for the (ApplyS) rule of Definition 5, so $\mathtt{e} = \mathtt{e}_1 @ \mathtt{e}_2 \Downarrow v$. □

B Proof of Lemma 5

Lemma 12. $\mathtt{P} : [] \Downarrow v$ *(by Definition 9) implies* $\mathtt{P} \Downarrow F(v)$ *(by Definition 5) for any program* `P`.

Proof Let $s \in \mathit{State}$, and assume $s \Downarrow v$ by Definition 9. This has a finite proof tree. We prove by induction on the height n of the proof tree that $s \Downarrow v$ implies $F(s) \Downarrow F(v)$ by Definition 5. It then follows that $\mathtt{P} : [] \Downarrow v$ implies $\mathtt{P} \Downarrow F(v)$.

$n = 0$: Two possibilities. First, $s \in \mathit{Value}$ implies there is an abstraction such that $s = \lambda \mathtt{x}.\mathtt{e} : \rho$ and $s = v$. $F(s) = F(v)$ is an abstraction $F(s) = F(\lambda \mathtt{x}.\mathtt{e} : \rho) = \lambda \mathtt{x}.F(\mathtt{e} : \rho|_{fv(\lambda x.e)})$ hence $F(s) \Downarrow F(v)$. Second, $s = \mathtt{x} : \rho$ implies $s \Downarrow \rho(\mathtt{x}) = v = \lambda \mathtt{y}.\mathtt{e}' : \rho'$. By definition $F(s) = \mathtt{x}[F(\rho(\mathtt{x}))/\mathtt{x}] = F(\rho(\mathtt{x})) = F(v)$. Since v is an abstraction also $F(s) \Downarrow F(v)$ as before.

$n > 0$: Consider $s \in \mathit{State}$ such that $s \Downarrow v$ has evaluation tree of height $n > 0$. It must be an application $s = \mathtt{e}_1 @ \mathtt{e}_2 : \rho$, and the last rule applied must be the (Apply) rule. By induction we have $F(\mathtt{e}_1 : \rho) \Downarrow F(\lambda \mathtt{x}.\mathtt{e}_0 : \rho_0) = \lambda \mathtt{x}.F(\mathtt{e}_0 : \rho_0|_{fv(\lambda x.e_0)})$, and $F(\mathtt{e}_2 : \rho) \Downarrow F(v_2)$, and $F(\mathtt{e}_0 : \rho_0[\mathtt{x} \mapsto v_2]) \Downarrow F(v)$.

By definition of F we have $F(\mathtt{e}_0 : \rho_0[\mathtt{x} \mapsto v_2]) = F(\mathtt{e}_0 : \rho_0|_{fv(\lambda x.e_0)})[F(v_2)/x]$. All premises in the standard semantics (ApplyS) rule hold, so we conclude $F(s) = F(\mathtt{e}_1 : \rho) @ F(\mathtt{e}_2 : \rho) \Downarrow F(v)$ as required. □

Lemma 13. *For any state* $\mathtt{e} : \rho$ *it holds that* $\mathtt{e} : \rho \to \mathtt{e}' : \rho'$ *implies* $F(\mathtt{e} : \rho) \to F(\mathtt{e}' : \rho')$.

Proof $\mathtt{e} : \rho \to \mathtt{e}' : \rho'$ implies $\mathtt{e} = \mathtt{e}_1 @ \mathtt{e}_2$. Clearly $F(\mathtt{e} : \rho) = F(\mathtt{e}_1 @ \mathtt{e}_2 : \rho) = F(\mathtt{e}_1 : \rho) @ F(\mathtt{e}_2 : \rho)$. There are 3 possibilities for $\mathtt{e}' : \rho'$:

1. (Operator) rule has been applied $\mathtt{e}' : \rho' = \mathtt{e}_1 : \rho$. The (Operator) rule always applies to an application hence we also have $F(\mathtt{e}_1 : \rho)@F(\mathtt{e}_2 : \rho) \to F(\mathtt{e}_1 : \rho)$
2. (Operand) rule has been applied $\mathtt{e}' : \rho' = \mathtt{e}_2 : \rho$. Then it must hold for some v that $\mathtt{e}_1 : \rho \Downarrow v$, so by Lemma 12, $F(\mathtt{e}_1 : \rho) \Downarrow F(v)$. It follows that rule (Operand) can be applied and hence $F(\mathtt{e}_1 : \rho)@F(\mathtt{e}_2 : \rho) \to F(\mathtt{e}_2 : \rho)$
3. (Call) rule has been applied. For some $\lambda\mathtt{x}.\mathtt{e}_0 : \rho_0$ and v_2 we have $\mathtt{e}_1 : \rho \Downarrow \lambda\mathtt{x}.\mathtt{e}_0 : \rho_0$ and $\mathtt{e}_2 : \rho \Downarrow v_2$ and $\mathtt{e}' : \rho' = \mathtt{e}_0 : \rho_0[\mathtt{x} \mapsto v_2]$. By Lemma 12 we have $F(\mathtt{e}_2 : \rho) \Downarrow F(v_2)$ and $F(\mathtt{e}_1 : \rho) \Downarrow F(\lambda\mathtt{x}.\mathtt{e}_0 : \rho_0) = \lambda x.F(\mathtt{e}_0 : \rho_{0|fv(\lambda x.e_0)})$. Then by rule (Call) there is a call to $F(\mathtt{e}_0 : \rho_{0|fv(\lambda x.e_0)})[F(v_2)/\mathtt{x}]$. As before, $F(\mathtt{e}_0 : \rho_0[\mathtt{x} \mapsto v_2]) = F(\mathtt{e}_0 : \rho_{0|fv(\lambda\mathtt{x}.\mathtt{e}_0)})[F(v_2)/\mathtt{x}]$.

We conclude that in all cases when we have a call $\mathtt{e} : \rho \to \mathtt{e}' : \rho'$ then we also have a call $F(\mathtt{e} : \rho) \to F(\mathtt{e}' : \rho')$ □

Proof of Lemma 5 : "Only if" is Lemma 12. For "if" suppose $\mathtt{P} \Downarrow F(v)$. By Lemma 13 this implies $\mathtt{P} : [] \Downarrow v'$ for some v'. Lemma 2 (Determinism) implies $v = v'$. □

C Proof of Theorem 2

Proof For the "safety" theorem we use induction on proofs of $s \Downarrow s', G$ or $s \to s', G$. Safety of the constructed graphs for rules (ValueG), (OperatorG) and (OperandG) is immediate by Definitions 13 and 11.

The variable lookup rule **(VarG)** yields $\mathtt{x} : \rho \Downarrow \rho(\mathtt{x}), G$ with $G = \{\mathtt{x} \overset{=}{\to} \bullet\} \cup \{\mathtt{x} \overset{\downarrow}{\to} \mathtt{y} \mid \mathtt{y} \in fv(\mathtt{e}')\}$ and $\rho(\mathtt{x}) = \mathtt{e}' : \rho'$. By Definition 12, $\overline{\mathtt{x} : \rho}(\mathtt{x}) = \rho(\mathtt{x}) = \overline{\rho(\mathtt{x})}(\bullet)$, so arc $\mathtt{x} \overset{=}{\to} \bullet$ satisfies Definition 13. Further, if $\mathtt{x} \overset{\downarrow}{\to} \mathtt{y} \in G$ then $\mathtt{y} \in fv(\mathtt{e}')$. Thus $\rho'(\mathtt{y}) \in \mathit{support}(\mathtt{x} : \rho)$, so $\rho(\mathtt{x}) \succ \rho'(\mathtt{y})$ as required.

The rule **(CallG)** concludes $s \underset{c}{\to} s', G$, where $s = \mathtt{e}_1@\mathtt{e}_2 : \rho$ and $s' = \mathtt{e}_0 : \rho_0[\mathtt{x} \mapsto v_2]$ and $G = G_1^{-\bullet} \cup G_2^{\bullet\mapsto\mathtt{x}}$. Its premises are $\mathtt{e}_1 : \rho \Downarrow \lambda\mathtt{x}.\mathtt{e}_0 : \rho_0, G_1$ and $\mathtt{e}_2 : \rho \Downarrow v_2, G_2$. Let $v_2 = \mathtt{e}' : \rho'$. We assume inductively that G_1 is safe for $(\mathtt{e}_1 : \rho, \lambda\mathtt{x}.\mathtt{e}_0 : \rho_0)$ and that G_2 is safe for $(\mathtt{e}_2, v_2)$.

We wish to show safety: that $a \overset{=}{\to} a' \in G$ implies $\overline{s}(a) = \overline{s'}(a')$, and $a \overset{\downarrow}{\to} a' \in G$ implies $\overline{s}(a) \succ \overline{s'}(a')$. By definition of $G_1^{-\bullet}$ and $G_2^{\bullet\mapsto\mathtt{x}}$, $a \overset{r}{\to} a' \in G = G_1^{-\bullet} \cup G_2^{\bullet\mapsto\mathtt{x}}$ breaks into 6 cases:

Case 1: $\mathtt{y} \overset{\downarrow}{\to} \mathtt{z} \in G_1^{-\bullet}$ because $\mathtt{y} \overset{\downarrow}{\to} \mathtt{z} \in G_1$, where $\mathtt{y} \in fv(\mathtt{e}_1)$ and $\mathtt{z} \in fv(\lambda\mathtt{x}.\mathtt{e}_0)$. By safety of G_1, $\rho(\mathtt{y}) \succ \rho_0(\mathtt{z})$. Thus, as required,

$$\overline{s}(\mathtt{y}) = \overline{\mathtt{e}_1@\mathtt{e}_2 : \rho}(\mathtt{y}) = \rho(\mathtt{y}) \succ \rho_0(\mathtt{z}) = \rho_0[\mathtt{x} \mapsto v_2](\mathtt{z}) = \overline{s'}(\mathtt{z})$$

Case 2: $\mathtt{y} \overset{=}{\to} \mathtt{z} \in G_1^{-\bullet}$ because $\mathtt{y} \overset{=}{\to} \mathtt{z} \in G_1$. Like Case 1.

Case 3: $\bullet \overset{\downarrow}{\to} \mathtt{z} \in G_1^{-\bullet}$ because $\bullet \overset{r}{\to} \mathtt{z} \in G_1$, where $\mathtt{z} \in fv(\lambda\mathtt{x}.\mathtt{e}_0)$. Now $\bullet$ in G_1 refers to $\mathtt{e}_1 : \rho$, so $\mathtt{e}_1 : \rho \succeq \rho_0(\mathtt{z})$ by safety of G_1. Thus, as required,

$$\overline{s}(\bullet) = \mathtt{e}_1@\mathtt{e}_2 : \rho \succ \mathtt{e}_1 : \rho \succeq \rho_0(\mathtt{z}) = \rho_0[\mathtt{x} \mapsto v_2](\mathtt{z}) = \overline{s'}(\mathtt{z})$$

Case 4: $\mathtt{y} \overset{\downarrow}{\to} \mathtt{x} \in G_2^{\bullet\mapsto\mathtt{x}}$ because $\mathtt{y} \overset{\downarrow}{\to} \bullet \in G_2$, where $\mathtt{y} \in fv(\mathtt{e}_2)$. By safety of G_2, $\rho(\mathtt{y}) \succ v_2$. Thus, as required,

$$\overline{s}(\mathtt{y}) = \rho(\mathtt{y}) \succ v_2 = \rho_0[\mathtt{x} \mapsto v_2](\mathtt{x}) = \overline{s'}(\mathtt{x})$$

Case 5: $\mathtt{y} \overset{=}{\to} \mathtt{x} \in G_2^{\bullet\mapsto\mathtt{x}}$ because $\mathtt{y} \overset{=}{\to} \bullet \in G_2$. Like Case 4.

Case 6: $\bullet \overset{\downarrow}{\to} \mathtt{x} \in G_2^{\bullet\mapsto\mathtt{x}}$ because $\bullet \overset{r}{\to} \bullet \in G_2$. By safety of G_2, $\mathtt{e}_2 : \rho \succeq v_2$. Thus, as required,

$$\overline{s}(\bullet) = \mathtt{e}_1@\mathtt{e}_2 : \rho \succ \mathtt{e}_2 : \rho \succeq v_2 = \rho_0[\mathtt{x} \mapsto v_2](\mathtt{x}) = \overline{s'}(\mathtt{x})$$

The rule **(ApplyG)** concludes $s \Downarrow v, G'; G$ from premises $s \underset{c}{\to} s', G'$ and $s' \Downarrow v, G$, where $s = \mathtt{e}_1@\mathtt{e}_2 : \rho$ and $s' = \mathtt{e}' : \rho'$. We assume inductively that G' is safe for (s, s') and G is safe for (s', v). Let $G_0 = G'; G$.

We wish to show that G_0 is safe: that $a \xrightarrow{=} c \in G_0$ implies $\overline{s}(a) = \overline{v}(c)$, and $a \xrightarrow{\downarrow} c \in G_0$ implies $\overline{s}(a) \succ \overline{v}(c)$. First, consider the case $a \xrightarrow{=} c \in G_0$. Definition 1 implies $a \xrightarrow{=} b \in G'$ and $b \xrightarrow{=} c \in G$ for some b. Thus by the inductive assumptions we have $\overline{s}(a) = \overline{s'}(b) = \overline{v}(c)$, as required.

Second, consider the case $a \xrightarrow{\downarrow} c \in G_0$. Definition 1 implies $a \xrightarrow{r_1} b \in G'$ and $b \xrightarrow{r_2} c \in G$ for some b, where either or both of r_1, r_2 are $\downarrow$. By the inductive assumptions we have $\overline{s}(a) \succeq \overline{s'}(b)$ and $\overline{s'}(b) \succeq \overline{v}(c)$, and one or both of $\overline{s}(a) \succ \overline{s'}(b)$ and $\overline{s'}(b) \succ \overline{v}(c)$ hold. By Definition of $\succ$ and $\succeq$ this implies that $\overline{s}(a) \succ \overline{v}(c)$, as required.

□

D Proof of Lemma 10

Proof The proof is by cases on which rule is applied to conclude $\mathtt{e} : \rho \Downarrow \mathtt{e}' : \rho'$ or $\mathtt{e} : \rho \to \mathtt{e}' : \rho'$. In all cases we show that some corresponding abstract interpretation rules can be applied to give the desired conclusion. The induction is on the total size of the proof[4] concluding that $\mathtt{e} : \rho \Downarrow \mathtt{e}' : \rho'$ or $\mathtt{e} : \rho \to \mathtt{e}' : \rho'$. The induction hypothesis is that the Lemma holds for all calls and evaluations performed in the computation before the last conclusion giving $\mathtt{e} : \rho \Downarrow \mathtt{e}' : \rho'$ or $\mathtt{e} : \rho \to \mathtt{e}' : \rho'$, i.e., we assume that the Lemma holds for premises of the rule last applied, and for any call and evaluation in the computation until then.

Base cases: Rule (Value), (Operator) and (Operand) in the exact semantics are modeled by axioms (ValueA), (OperatorA) and (OperandA) in the abstract semantics. These are the same as their exact-evaluation counterparts, after removal of environments for (ValueA) and (OperatorA), and a premise as well for (OperandA). Hence the Lemma holds if one of these rules were the last one applied.

The (Var) rule is, however, rather different from the (VarA) rule. If (Var) was applied to a variable $\mathtt{x}$ then the assumption is $\mathtt{P} : [] \to^* \mathtt{x} : \rho$ and $\mathtt{x} : \rho \Downarrow \mathtt{e}' : \rho'$. In this case $x \in dom(\rho)$ and $\mathtt{e}' : \rho' = \rho(x)$. Now $\mathtt{P} : [] \to^* \mathtt{x} : \rho$ begins from the empty environment, and we know all calls are from state to state. The only possible way $\mathtt{x}$ can have been bound is by a previous use of the (Call) rule, the only rule that extends an environment. The premises of this rule require that operator and operand in an application $e_1 \mathtt{@} e_2 : \rho''$ have previously been evaluated.

This requires that $e_1 \mathtt{@} e_2 \in subexp(P)$. By induction we can assume that the Lemma holds for $e_1 : \rho'' \Downarrow \lambda x.e_0 : \rho_0$ and $e_2 \Downarrow e' : \rho'$, so $e_1 \Downarrow \lambda x.e_0$ and $e_2 \Downarrow e'$ in the abstract semantics. Now we have all premises of rule (VarA), so we can conclude that $\mathtt{x} \Downarrow \mathtt{e}'$ as required.

For remaining rules (Apply) and (Call), when we assume that the Lemma holds for the premises in the rule applied to conclude $\mathtt{e} \Downarrow \mathtt{e}'$ or $\mathtt{e} : \rho \to \mathtt{e}' : \rho'$, then this gives us the premises for the corresponding rule for abstract interpretation. From this we can conclude the desired result. □

E Proof of Lemma 11

Proof The rules are the same as in Section 4.3, only extended with size-change graphs. We need to add to Lemma 10 that the size-change graphs generated for calls and evaluations can also be generated by the abstract interpretation. The proof is by cases on which rule is applied to conclude $\mathtt{e} \Downarrow \mathtt{e}', G$ or $\mathtt{e} : \rho \to \mathtt{e}' : \rho', G$.

We build on Lemma 10, and we saw in the proof of this that in abstract interpretation we can always use a rule corresponding to the one used in exact computation to prove corresponding steps. The induction hypothesis is that the Lemma holds for the premises of the rule in exact semantics.

Base case (VarAG): By Lemma 10 we have $\mathtt{x} : \rho \Downarrow \mathtt{e}' : \rho'$ implies $\mathtt{x} \Downarrow \mathtt{e}'$. The size-change graph built in (VarAG) is derived in the same way from $\mathtt{x}$ and $\mathtt{e}'$ as in rule (VarG), and they will therefore be identical.

[4] This may be thought of as the number of steps in the computation of $\mathtt{e} : \rho \Downarrow \mathtt{e}' : \rho'$ or $\mathtt{e} : \rho \to \mathtt{e}' : \rho'$ starting from $P : []$.

For other call- and evaluation rules without premises, the abstract evaluation rule is as the exact-evaluation rule, only with environments removed, and the generated size-change graphs are not influenced by environments. Hence the Lemma will hold if these rules are applied.

For all other rules in a computation: When we know that Lemma 10 holds and assume that Lemma 11 hold for the premises, then we can conclude that if this rule is applied, then Lemma 11 holds by the corresponding rule from abstract interpretation. □

References

1. C.C. Frederiksen and N.D. Jones. Running-time Analysis and Implicit Complexity. Submitted to *Journal of Automated reasoning*Journal dadada.
2. J.Y. Girard, Y. Lafont, P. Taylor. *Proofs and Types.* Cambridge University Press, 1989.
3. N.D. Jones and F. Nielson. Abstract Interpretation: a Semantics-Based Tool for Program Analysis. In Handbook of Logic in Computer Science, pp. 527-629. Oxford University Press, 1994.
4. N.D. Jones and A. Glenstrup. Partial Evaluation Termination Analysis and Specialization-Point Insertion. ACM Transactions on Programming Languages and Systems, to appear in 2004.
5. C.S. Lee, N.D. Jones and A.M. Ben-Amram "The Size-Change Principle for Program Termination" POPL 2001: Proceedings 28^{th} ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, January 2001.
6. C.S. Lee. Finiteness analysis in polynomial time. In Static Analysis: 9th International Symposium, SAS 2002 (M Hermenegildo and G Puebla, eds.), pp. 493-508. Volume 2477 of Lecture Notes in Computer Science. Springer. September, 2002.
7. C.S. Lee. Program termination analysis in polynomial time. In Generative Programming and Component Engineering: ACM SIGPLAN/SIGSOFT Conference, GPCE 2002 (D Batory, C Consel, and W Taha, eds.), pp. 218-235. Volume 2487 of Lecture Notes in Computer Science.
8. C.S. Lee. "Program Termination Analysis and the Termination of Offline Partial Evaluation" Ph.D. thesis, University of Western Australia, March 2001
9. G.D. Plotkin. Call-by-name, call-by-value and the lambda-calculus. Theoretical Computer Science, 1, 1975.