Finiteness Analysis in Polynomial Time

Chin Soon Lee*

Datalogisk Institut, Copenhagen University, Denmark

Abstract. To achieve the termination of offline partial evaluation, it is necessary to ensure that static variables assume boundedly many values during specialization. Various works have addressed the analysis of variable boundedness, also called finiteness analysis, in the context of specializing first-order functional programs. The underlying reasoning is always: Observing arbitrarily many increases in a static variable during specialization must be *impossible*, if that would imply observing arbitrarily long sequences of size decreases among some bounded-variable values.

Static analysis is used to collect a set of bipartite graphs that describe the parameter dependencies and data size changes in possible state transitions of the specializer (operating on the program). We capture the reasoning above as a condition on the graphs. This condition is decidable, but complete for PSPACE. We therefore derive a polynomial-time approximation, by considering realistic parameter size-change behaviour.

1 Introduction

1.1 Termination of Offline Partial Evaluation

Staging the computation. Partial evaluation is a general paradigm for program specialization. However, we will limit our discussion to the specialization of first-order functional programs.

Given a program and part of its input, a residual program is generated which, when given the remainder of the input, produces the same result as the original program given all of the input at once. The idea is to perform computations not depending on the missing part of the input, and thereby avoid these computations at runtime. This is achieved by a combination of unfolding, symbolic simplification and function specialization.

Staging the partial evaluation. Offline partial evaluation implements a two-phase strategy [9]. First, binding-time analysis (BTA) is performed to classify variables as static or dynamic, to indicate whether a variable's values will be available during specialization. Such a classification is called a binding-time division. A division satisfies congruence if the actual arguments for static parameters do not depend on the dynamic parameters. In addition to satisfying congruence, the division for the program must classify any absent input as dynamic. Based

^{*} This research is supported by DAEDALUS, the RTD project IST-1999-20527 of the IST Programme within the European Union's Fifth Framework Programme.

on an acceptable binding-time division, each construct in the program is then annotated static or dynamic, to indicate reduction or residualization.

The next phase is fast, syntax-directed specialization: The binding-time of a subexpression's evaluation is *not* used to decide how to reduce an expression. This leads to a lightweight second phase.

Advantages over on-line methods, which make specialization decisions based on *type tags* for transformation-time values, include the following.

- 1. Fast specialization: Just as calling a specialized function or program multiple times increases savings in execution time, specializing an annotated function or program multiple times increases savings in transformation time, compared to specializing unannotated code.
- 2. **Annotations** can be consulted to understand the specializer's behaviour.
- 3. **Self application**: Specializing the specializer to an interpreter can yield an effective compiler [9]. This is an elegant approach to compiler generation, as only the interpreter has to be proven correct. Binding-time information is central to the success of self-application [5].

On the other hand, on-line partial evaluation can achieve more aggressive specialization. For example, conditional branches are not required to return results of matching binding-times, as required for syntax-directed specialization.

Two sources of non-termination. A congruent division is necessary to prevent binding-time type errors for offline partial evaluation, but it is not sufficient to guarantee a residual program.

There are two sources of non-termination for the specialization process. The first is infinite unfolding. This can be avoided by adapting methods for guaranteeing program termination. The principle: If every infinite unfolding would give rise to a sequence of static values whose sizes descend infinitely, then no infinite unfolding is possible [8, 14, 11]. For the rest of this paper, we will assume that finite unfolding has been ensured by marking sufficiently many function calls as dynamic. Refer to [3] for a thorough treatment of this issue.

Non-termination is also caused by accumulating parameters under dynamic control [9]. Consider specializing f below with d unknown and s equal to [].

```
f(d,s) = if d=s then s else f(d,cons(1,s))
```

Suppose that the f call is annotated dynamic. Then the specializer would generate versions of f with s equal to [], [1], [1,1], The standard prescription is to re-classify s as dynamic. This is known as *generalization*. We do not want to generalize all static computations controlled by a dynamic variable; the form of coding below, with z dynamic and xs static, is a useful way to coax the specializer to tabulate h with respect a finite set of values [10].

```
g(z,xs) = if z=hd(xs) then h(hd(xs)) else g(z,tl(xs))
```

These examples illustrate the concern of this article—a sufficiently precise analysis to determine that xs above is bounded during specialization, but point

out that s is possibly unbounded in the first example. Possibly unbounded variables are generalized. By repeatedly applying BTA, a safe unfolding strategy, and generalizations, the termination of specialization can be guaranteed.

1.2 Background of This Work

The analyses of variable boundedness in [7,4,1,3,8] can all be seen as applying this reasoning: Observing arbitrarily many *increases* in a static variable during specialization must be *impossible*, if that would imply observing arbitrarily long sequences of size decreases among some bounded-variable values. As such deductions rely on *known* bounded variables, it is necessary to formulate an explicit condition to collect such variables [4,3], or settle for a weak criterion to determine an initial set of them [7]. In any case, analysis of variable boundedness is complex. This led Neil Jones to adapt the above reasoning for a simpler but related problem. The *size-change principle* for program termination states that an infinite computation must be *impossible*, if it would give rise to an infinite sequence of size decreases for some data values.

In joint work with Jones and Ben-Amram, a condition based on this principle was formulated and studied [13]. The size-change termination (SCT) condition is stated in terms of size-change graphs, bipartite graphs whose arcs describe decreasing and non-increasing data size changes in possible program state transitions. A sequence of size-change graphs that respects program control flow is called a multipath. A set of graphs satisfies SCT if every infinite multipath exhibits a connected sequence of arcs—a thread—containing infinitely many decreases. Satisfaction of SCT thus implies that possible state transition sequences are finite, by the size-change principle. The SCT condition is decidable, but complete for PSPACE.

In [12], we studied SCT approximations that work well in practice. They are based on trying to uncover a set of graphs \mathcal{S}^* that could be the *infinity set* of a multipath without infinite descent. Consider a maximal strongly-connected set of size-change graphs \mathcal{S} . (A set of size-change graphs is strongly-connected if there is a multipath that contains all the graphs, and starts and ends with the same graph.) We have developed techniques to spot decreasing arcs that occur among a bounded number of threads in any multipath composed of \mathcal{S} elements. The graphs containing these arcs clearly cannot appear in \mathcal{S}^* , so they can be removed from consideration. If after repeatedly removing graphs, no strongly-connected set remains, then SCT is established.

The decreasing arcs signal "progress" for the computation, and we normally expect to find such arcs easily. We have an efficient means to spot them among fan-in free graphs, which occur commonly in practice. This leads to a worst-case quadratic-time approximation of SCT. A more complicated approximation handles general graphs. In our extensive experiments, we have found that the approximations are as effective as SCT in practice. They are also generally much faster, and come with scalability guarantees. In this article, we adapt the polynomial-time descent detection for the analysis of variable boundedness.

1.3 Structure of This Article

Section 2 introduces the subject language L, its specialization, and state transitions for the specializer. Section 3 discusses the use of abstract state transitions to capture the parameter dependencies and data size changes in possible state transitions. Section 4 formulates the boundedness principle in terms of the abstract transitions. We show that the resulting condition is decidable but complete for PSPACE. Section 5 presents a polynomial-time approximation of this condition. We then describe a procedure to collect bounded variables, and demonstrate the procedure on an example. Section 6 contains some concluding remarks.

2 Programs and Their Specialization

2.1 The Subject Language L

The syntax of L. The subject language L is a minimal first-order functional language with eager evaluation. Programs in L are generated by the following grammar, where $c \in Cst$ is a constant, $x, x_i \in Var$ are variable identifiers, $f \in Fun$ is a function identifier, and $op \in Op$ is a primitive operator.

```
\begin{array}{ll} p \in Pgm & ::= d_1 \dots d_m \\ d_i, d \in Def & ::= f(x_1, \dots, x_n) = e^f \\ e, e_i, e^f \in Exp ::= c \mid x \mid op(e_1, \dots, e_n) \mid \text{if } e_1 \ e_2 \ e_3 \mid f(e_1, \dots, e_n) \end{array}
```

For any program, the sets Cst, Op, Var and Fun are finite. We will assume that they are mutually exclusive.

The definition of f has the form $f(x_1, \ldots, x_n) = e^f$, where e^f is called the body of f. The number n of parameters is the function's arity, written ar(f). Notation: $Param(f) = \{f^{(1)}, \ldots, f^{(n)}\}$ is the set of f parameters. In examples they are named by identifiers. Parameters are assumed to be in scope when they are used. Function calls are labelled, e.g., ${}^cf(e_1, \ldots, e_n)$. The set of function calls is denoted C. We assume that function calls and primitive operations have the correct number of arguments. Every program has a definition for the function goal. This function is never called in the program.

The Semantics of L. Programs in L are untyped. To provide L with a bigstep operational semantics [6], we equip it with a syntactic category of values containing the distinguished element true. The set Val of values is a denumerable set. Its elements are usually written u, v. Operators are interpreted by \mathcal{O} . For $op \in Op$, $\mathcal{O}[op]$ is a partial function from Val^* (the set of finite value sequences) to Val. Constants are interpreted by \mathcal{K} . For $c \in Cst$, $\mathcal{K}[c] \in Val$. Other notation: For vector $\mathbf{v} = (v_1, \ldots, v_n)$ and $i \in [1, n]$, $(\mathbf{v})_i = v_i$.

Definition 1. For $e \in Exp$, e evaluates to u on v if $v, e \downarrow u$ follows from the rules of Fig. B.1. Otherwise, e is undefined on v. Program p evaluates to u on v if e^{goal} evaluates to u on v. Otherwise, p is undefined on v.

2.2 The 2-Level Language 2L

In the first phase of offline partial evaluation, each construct in the subject program is annotated as static (to be reduced) or dynamic (to be made residual). Formally, this is a mapping from Pgm to 2Pgm, a syntactic category of 2-level programs.

As for L, te^f denotes the body of f in tp. The lift(te) construct directs the specializer to convert the result of evaluating te from a value to an expression. The set of values handled by the specializer is $2Val = Val \cup Exp$. We write a, b, a_i, b_i for 2Val values.

Definition 2. For 2L program tp, $\phi(tp) \in Pgm$ is the program obtained by dropping all bt annotations and lifts. ϕ is called the annotation-forgetting function.

Definition 3. (Well annotations)

- 1. A BT environment $\tau : Var \cup Fun \rightarrow \{S, D\}$.
- 2. Write te: S (resp. te: D) if te: S (resp. te: D) follows from the rules of Fig. B.2.
- 3. The subject 2L program tp is well-annotated wrt τ if for each $f \in Fun$, we have $te^f : \tau(f)$.
- 4. τ respects binding-time assumptions if $\tau(x) = S$ when x is a static input, $\tau(x) = D$ when x is a dynamic input, and $\tau(\text{goal}) = D$.

For the remainder of the paper, we assume that we have been given tp and τ such that tp is well-annotated wrt τ . In general, given an L program p, it is possible to derive tp and τ such that τ respects BT assumptions, tp is well-annotated wrt τ , and $\phi(tp) = p$. We can also require certain variables and expressions to be dynamic.

Example of annotated program: Below is an interpreter for an L-like language, adapted from the one in [4]. The case construct is "syntactic sugar". Functions body_of and param_of can be regarded as destructive operators. Function lkvar looks up the value of an object program variable given parallel name and value lists. For ns1 static and vs1 dynamic, lkvar calls reduce to list access expressions. Functions lkbody and lkpar perform association-list look-ups. For e1 and p1 static, calls to lkbody and lkpar reduce to Val elements. The definitions of lkvar, lkbody and lkpar have been omitted. The program is well-annotated with the following variables static: p, e1, p1, ns1, es2, p2, ns2.

Specializing the interpreter to an object program successfully eliminates the interpretative overhead of syntactic dispatch and variable dereference [9]. Variable dereference is removed as 1kvar calls evaluate to list access expressions.

Definition 4. (Specialization of 2L programs)

- 1. Let $f^{(i_1)}, \ldots, f^{(i_s)}$ and $f^{(j_1)}, \ldots, f^{(j_d)}$ be the static and dynamic parameters of f respectively, according to BT environment τ . For $\mathbf{a} = (a_1, \ldots, a_{ar(f)})$, let $S\text{-Pos}_f(\mathbf{a}) = (a_{i_1}, \ldots, a_{i_s})$ and $D\text{-Pos}_f(\mathbf{a}) = (a_{j_1}, \ldots, a_{j_d})$.
- 2. **a** respects annotations of f if $S ext{-}Pos_f(\mathbf{a}) \in Val^*$ and $D ext{-}Pos_f(\mathbf{a}) \in Exp^*$. Further, if for $f^{(k)}$ dynamic, $(\mathbf{a})_k$ is the variable $f^{(k)}$, then \mathbf{a} is a 2-level input to f.
- 3. For $f \in Fun$ and a respecting the annotations of f, write $a, te^f \downarrow b$ if it follows from the rules of Fig. B.3.
- 4. For f such that $\tau(f) = D$, the specialization of f wrt v is denoted f_v . Its definition is f_v x = e, where for some 2-level input a to f, S-Pos_f(a) = v, D-Pos_f(a) = x, and a, te^f \downarrow e.
- 5. A set of definitions is closed if every referenced function has a definition.
- 6. A specialization of 2L-program tp wrt static input v is a minimal closed set of specializations containing a definition for $goal_v$.

There are well-known problems connected with careless call unfolding: duplication of code, duplication of computation and residual programs that terminate more often than their subject programs. These problems are overcome by annotating enough calls as dynamic [9]. We will not be concerned with them here.

Definition 5. (Transitions among specializer states)

1. For a (resp. b) respecting annotations of f (resp. g), $(f, a) \stackrel{S}{\Rightarrow} (g, b)$ if $g^{S}(te_{1}, \ldots, te_{n})$ occurs in te^{f} , and for each i, we have a, $te_{i} \downarrow (b)_{i}$.

- 2. For a respecting annotations of f and b a 2-level input to g, $(f, \mathbf{a}) \stackrel{D}{\Rightarrow} (g, \mathbf{b})$ if $g^{\mathbf{D}}(te_1, \ldots, te_n)$ occurs in te^f , and for $\tau(g^{(i)}) = S$, we have \mathbf{a} , $te_i \Downarrow (\mathbf{b})_i$.
- 3. Define \Rightarrow to be $\stackrel{S}{\Rightarrow} \cup \stackrel{D}{\Rightarrow}$. Write \Rightarrow^* for the transitive closure of \Rightarrow .
- 4. 2L program to satisfies local termination if for $f \in Fun$ and a respecting the annotations of f, every sequence of $\stackrel{S}{\Rightarrow}$ transitions beginning with (f, a) is finite.
- 5. 2L program to satisfies global termination if for $f \in Fun$ and a respecting the annotations of f, the set $\{(g, b) \mid (f, a) \Rightarrow^* (g, b)\}$ is finite.

Proposition 1. Suppose that tp satisfies local termination. Then for $f \in Fun$ and a respecting the annotations of f, there exists b such that a, $te^f \downarrow b$.

Proposition 2. Suppose that tp satisfies local and global termination. Then for any static input v to tp, the specialization of tp wrt v is (finitely) derivable.

Local termination implies that any residual function definition is successfully generated given sufficient resources (because any reduction eventually terminates), while global termination implies that the set of residual function definitions to be generated is finite, i.e., function specialization is finite. Local termination is ensured by the unfolding strategy. This article is concerned with the more subtle question of global termination.

Definition 6. A variable $g^{(i)}$ is bounded if for $f \in Fun$ and a respecting the annotations of f, the set $\{(b)_i \mid (f, a) \Rightarrow^* (g, b)\}$ is finite.

Lemma 1. Given local termination, if every static variable is bounded, then global termination is satisfied.

3 Abstract Transitions

Definition 7. (Data size changes)

- 1. We will assume a size function, $| \bullet | : Val \to IN$.
- 2. For te a subexpression of te^f such that te: S, describe te as non-increasing on a static $f^{(i)}$ if for a respecting f annotations, a, te $\downarrow b$ where $b \neq \text{undef}$ implies $|(a)_i| > |b|$; te is described as decreasing on $f^{(i)}$ if $|(a)_i| > |b|$.

Typically, the size of a list is its number of constructors. We will assume that for x static, hd(x) and tl(x) are decreasing on x. The composition of decreasing operations is decreasing, thus hd(tl(x)) is decreasing on x. An expression consisting of just a static variable x is clearly non-increasing on x.

Definition 8. (Parameter dependence) Fix \leq to be any downward-closed preorder for Val [7], i.e., for any $v \in Val$, the set of values u such that $u \leq v$ is finite. Let to be a subexpression of te^f such that te : S.

1. A dependence description for te is a subset Δ of $\{\uparrow, \uparrow\} \times \{f^{(i)} \mid \tau(f^{(i)}) = S\}$. $\uparrow(x)$ (resp. $\uparrow(x)$) in Δ indicates a constructive (resp. safe) dependence.

- 2. Let $\sigma: X \to Val$, where $X = \{x \mid \uparrow(x) \in \Delta\}$. Call b fresh with respect to σ, Δ if for some 2-level input a to f:
 - $-(a)_i = \sigma(f^{(i)}) \text{ for each } \uparrow(f^{(i)}) \in \Delta,$
 - -a, $te \downarrow b$ where $b \neq$ undef, and
 - $-b \not\preceq (a)_i \text{ for each } \uparrow (f^{(i)}) \in \Delta,$
- 3. If for every σ , the set of values fresh wrt σ , Δ is finite, then Δ is safe for te.

A possibility for \leq is the homeomorphic embedding relation. The above definition is admittedly difficult to use. However, if a, $te \downarrow b$ and $b \neq$ undef implies that b is embedded in $(v)_i$, then $\{\uparrow(f^{(i)})\}$ is certainly safe for te. For example, in the annotated interpreter, $\{\uparrow(p1)\}$ is safe for lkbody(f,p1) and lkpar(f,p1). The following is always safe for te: $\{\uparrow(x) \mid x \text{ occurs in } te, \tau(x) = S\}$.

Definition 9. (Abstract transitions)

- 1. An abstract transition α has the form $f \xrightarrow{\Delta, \Gamma} g$ where $f, g \in Fun$ are called the source and target functions, and Δ and Γ are (essentially) bipartite graphs.
- 2. An element of Δ has the form $x \xrightarrow{\delta} y$, where $x \in \{z \in Param(f) \mid \tau(z) = S\}$, $y \in \{z \in Param(g) \mid \tau(z) = S\}$ and $\delta \in \{\uparrow, \uparrow\}$.
- 3. An element of Γ has the form $x \xrightarrow{\gamma} y$, where $x \in \{z \in Param(f) \mid \tau(z) = S\}$, $y \in \{z \in Param(g) \mid \tau(z) = S\}$ and $\gamma \in \{\overline{\downarrow}, \downarrow\}$.

Definition 10. (Safe abstract transitions)

- 1. Let ${}^cg(te_1,\ldots,te_n)$ be a function call in te^f . Then $f \xrightarrow{\Delta,\Gamma} g$ is safe for c if
 - $-f^{(i)} \stackrel{\downarrow}{\rightarrow} g^{(j)} \in \Gamma$ implies that te_j is decreasing on $f^{(i)}$.
 - $-f^{(i)}\stackrel{\overline{\mathbb{T}}}{
 ightarrow}g^{(j)}\in arGamma\ implies\ that\ te_j\ is\ non-increasing\ on\ f^{(i)}.$
 - $-\{\delta(f^{(i)})\mid f^{(i)}\stackrel{\delta}{\to} g^{(j)}\in\Delta\}$ is a safe dependence description for te_j .
- 2. A set of abstract transitions \mathcal{T} safe for tp has an element, denoted α_c , safe for each function call c.

Definition 11. The dependency graph \mathcal{D} has arcs corresponding to dependency arcs in every $\alpha_c \in \mathcal{T}$. Its arc set is $\{x \overset{\delta}{\leadsto}_c y \mid x \overset{\delta}{\to} y \in \Delta, \ \alpha_c = f \overset{\Delta, \Gamma}{\longrightarrow} g\}$. Write $x \leadsto^+ y$ if x and y are connected by a non-empty sequence of \mathcal{D} arcs.

It is easy to compile a safe set of abstract transitions based on simple analysis of call arguments. The set depicted in Fig. 3.1 has been derived this way. In general, proper size and dependency analysis is needed for better precision. Figure 3.1 also shows the dependency graph due to the depicted abstract transitions. Safe dependence labels and most function-call labels have been omitted.

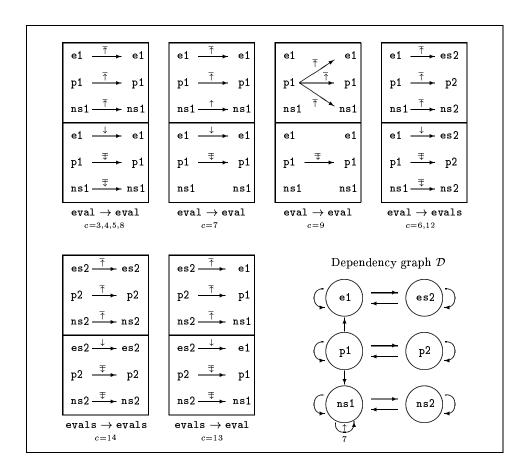


Figure 3.1: Abstract transitions and dependency graph for the interpreter example

4 The Boundedness Principle

Flow-legal abstract transitions α_{c_1} and α_{c_2} can be composed in a natural way to summarize the parameter dependencies and data size changes observed in the call sequence c_1c_2 .

Definition 12. (Composition of abstract transitions)

1. For
$$\alpha_{c_1} = f_0 \xrightarrow{\Delta, \Gamma} f_1$$
 and $\alpha_{c_2} = f_1 \xrightarrow{\Delta', \Gamma'} f_2$, define $\alpha_{c_1}; \alpha_{c_2} = f_0 \xrightarrow{\Delta'', \Gamma''} f_2$, where
$$\Delta'' = \Delta; \Delta' = \{x \xrightarrow{\uparrow} x'' \mid x \xrightarrow{\delta} x' \in \Delta, \ x' \xrightarrow{\delta'} x'' \in \Delta', \ \delta = \uparrow \ or \ \delta' = \uparrow\} \cup \{x \xrightarrow{\delta} x'' \mid x \xrightarrow{\delta} x' \in \Delta, \ x' \xrightarrow{\delta} x'' \in \Delta', \ \delta = \uparrow\}$$

$$\Gamma'' = \Gamma; \Gamma' = \{x \xrightarrow{\downarrow} x'' \mid x \xrightarrow{\gamma} x' \in \Gamma, \ x' \xrightarrow{\gamma'} x'' \in \Gamma', \ \gamma = \downarrow \ or \ \gamma' = \downarrow\} \cup \{x \xrightarrow{\gamma} x'' \mid x \xrightarrow{\gamma} x' \in \Gamma, \ x' \xrightarrow{\gamma} x'' \in \Gamma', \ \gamma = \ddagger\}$$

2. The composition closure of abstract transitions \mathcal{T} is the least $\overline{\mathcal{T}}$ that satisfies $\overline{\mathcal{T}} = \mathcal{T} \cup \{\alpha; \alpha' \mid \alpha, \alpha' \in \overline{\mathcal{T}}\}.$

Theorem 1. We are given z with $\tau(z) = S$. Let $X_0 = \{x \mid x \rightsquigarrow^+ z, z \not\rightsquigarrow^+ x\}$ and $X_1 = \{x \mid x \rightsquigarrow^+ z, z \rightsquigarrow^+ x\}$. Then z is bounded if

- 1. for $x \in X_0$, x is bounded, and
- 2. for $x \in X_1$, $\alpha = f \xrightarrow{\Delta, \Gamma} f \in \overline{\mathcal{T}}$ such that $\alpha; \alpha = \alpha$ and $x \xrightarrow{\uparrow} x \in \Delta$, there exists some bounded x' for which $x' \xrightarrow{\downarrow} x' \in \Gamma$.

The intuition behind Theorem 1 is as follows. For z to be unbounded, it is either assigned values from an unbounded variable in X_0 , or arbitrarily many increases (constructions) can be observed in an X_1 variable during some specialization.

The first condition in Theorem 1 ensures that z is not assigned values from possibly unbounded variables in X_0 . Thus the only way for z to be unbounded is for there to be arbitrarily many increases (constructions) in an X_1 variable during some specialization. In this case, Ramsey's Theorem guarantees that the specialization contains state transitions described by arbitrarily many repeats of some $\alpha \in \overline{\mathcal{T}}$ such that α ; $\alpha = \alpha$ and $x \xrightarrow{\uparrow} x$ is in the Δ part of α . By Condition 2 in Theorem 1, this implies that arbitrarily long sequences of size decreases are observed in a bounded variable x', which is impossible.

The formal proof of Theorem 1 can be found in the Appendix A.

Theorem 2. Condition 2 in Theorem 1 is complete for PSPACE.

Proof. Given a set \mathcal{G} of Γ graphs, it is PSPACE-hard to decide whether every Γ in the composition closure of \mathcal{G} for which $\Gamma; \Gamma = \Gamma$ has an arc of the form $x' \stackrel{\downarrow}{\to} x'$ [13]. This problem is easily reduced to Condition 2 in Theorem 1. Let $\mathcal{T} = \{f \stackrel{\Delta,\Gamma}{\to} g \mid \Gamma \in \mathcal{G}, \Gamma\text{-arcs} \text{ are from } Param(f) \text{ to } Param(g), \Delta = \{x \stackrel{\uparrow}{\to} x\}\},$ where x is a fresh variable, every variable is deemed static, and every variable other than x is bounded. Then the condition holds for x just when every graph Γ in the composition closure of \mathcal{G} for which $\Gamma; \Gamma = \Gamma$ has some arc $x' \stackrel{\downarrow}{\to} x'$, for some bounded x'.

The condition can clearly be refuted in non-deterministic polynomial-space. It can therefore be decided in PSPACE by standard techniques. \Box

5 A Polynomial-Time Boundedness Condition

We will adapt the approach in [12] (described briefly in Sect. 1.2) to approximate Condition 2 in Theorem 1. For simplicity, we will present a technique that works for fan-in $free\ \Gamma$ graphs. Such graphs can be handled efficiently. They also arise in a natural way. A method in [12] can be adapted for general graphs.

Definition 13. Γ is fan-in free if $x \stackrel{\gamma}{\to} z, y \stackrel{\gamma'}{\to} z \in \Gamma$ implies x = y.

Henceforth, we assume that all Γ graphs are fan-in free.

Definition 14. Let X' contain known bounded variables.

- 1. We write $G_c: f \to g$ and call G_c a size-change graph for c if $\alpha_c = f \xrightarrow{\Delta, \Gamma} g$ and G_c is the set of arcs $\{x \xrightarrow{\gamma} y \in \Gamma \mid x, y \in X'\}$.
- 2. Define $\mathcal{G} = \{G_c \mid c \in C\}$.

Definition 15. For $\mathcal{G}' \subseteq \mathcal{G}$, the descent-preservers wrt \mathcal{G}' , denoted $DP(\mathcal{G}')$, is the largest subset of Var such that

$$DP(\mathcal{G}') = \{x \mid \forall G \in \mathcal{G}' \cdot G : f \to g \land x \in Param(f) \Rightarrow \exists y \in DP(\mathcal{G}') \cdot x \xrightarrow{\gamma} y \in G\}$$

By design, any descent between $DP(\mathcal{G}')$ variables in a graph G of \mathcal{G}' implies a descent between $DP(\mathcal{G}')$ variables in the composition of G with any \mathcal{G}' graph.

Lemma 2. Let $\mathcal{G}' \subseteq \mathcal{G}$. And let N and M be the size of \mathcal{G} and \mathcal{G}' respectively. Then $DP(\mathcal{G}')$ can be computed in time O(M), with an $O(N^2)$ initialization performed on \mathcal{G} .

The algorithm maintains a set of counts for the source variables in each graph of \mathcal{G}' . These counts initially record the out-degrees for the source variables. Start by marking each variable whose count is 0. Insert these variables into a worklist. When processing x', inspect those graphs with arcs connected to x'. For such an arc $y' \stackrel{\gamma}{\to} x'$, if y' is not yet marked, reduce its count. If the count becomes 0, mark y' and insert it into the worklist. Terminate when the worklist is empty. We claim, without proof, that upon termination, the unmarked variables form $DP(\mathcal{G}')$, and that the procedure has O(M) time-complexity, with an $O(N^2)$ initialization performed on \mathcal{G} . (Initialization indexes all the functions and variables, and sets up indexing structures.)

Theorem 3. As in Theorem 1, suppose we are given z such that $\tau(z) = S$. Let $X_1 = \{x \mid x \rightsquigarrow^+ z, z \rightsquigarrow^+ x\}$. If Condition 2 in Theorem 1 is violated, there exists a strongly-connected subgraph \mathcal{D}^* of \mathcal{D} such that the following hold.

- $-\mathcal{D}^*$ has a vertex $x \in X_1$,
- $-\mathcal{D}^*$ has an arc of the form $x \stackrel{\uparrow}{\leadsto}_c y$,
- For $C^* = \{c \mid x \stackrel{\delta}{\leadsto}_c y \text{ in } \mathcal{D}^*\}$ and $\mathcal{G}^* = \{G_c \mid c \in C^*\}$, there do not exist $x', y' \in DP(\mathcal{G}^*)$ and $c \in C^*$ such that $x' \stackrel{\downarrow}{\to} y' \in G_c$.

Proof. Suppose Condition 2 in Theorem 1 is violated, and let $\alpha = f \xrightarrow{\Delta, \Gamma} f \in \overline{\mathcal{T}}$ be such that:

- $-\alpha = \alpha; \alpha,$
- $-x \xrightarrow{\uparrow} x \in \Delta \text{ for } x \in X_1, \text{ and }$
- there does not exist any bounded z' such that $z' \stackrel{\downarrow}{\to} z' \in \Gamma$.

By definition of $\overline{\mathcal{T}}$, $\alpha = \alpha_{c_1}; \ldots; \alpha_{c_n} = f_0 \xrightarrow{\Delta_1, \Gamma_1} f_1; \ldots; f_{n-1} \xrightarrow{\Delta_n, \Gamma_n} f_n$, with $f_0 = f_n = f$. It is easily proved, by induction on n, that \mathcal{D} has some arcs $x_0 \overset{\delta_1}{\leadsto}_{c_1} x_1, \ldots, x_{n-1} \overset{\delta_n}{\leadsto}_{c_n} x_n$ such that $x_0 = x_n = x$ and some $\delta_t = \uparrow$. Let \mathcal{D}^* be the subgraph of \mathcal{D} comprised of these arcs, and vertices x_0, \ldots, x_n . Then \mathcal{D}^* is strongly-connected. It has a vertex $x \in X_1$ and an arc labelled \uparrow . Suppose that for $C^* = \{c \mid x \overset{\delta}{\leadsto}_c y \text{ in } \mathcal{D}^*\}$ and $\mathcal{G}^* = \{G_c \mid c \in C^*\}$, there are $x', y' \in \mathcal{D}P(\mathcal{G}^*)$ and $c \in C^*$ such that $x' \overset{\downarrow}{\to} y' \in G_c$. We show this leads to a contradiction.

For $i=0,\ldots,n$, let $P_i=Param(f_i)\cap DP(\mathcal{G}^\star)$. By definition of DP and the assumption of no fan-ins, $|P_i|\leq |P_{i+1}|$ for $i=0,\ldots,n-1$. Thus, we have that $|P_0|\leq |P_1|\leq \ldots \leq |P_n|=|P_0|$. Let $K=|P_0|$. It is easily proved, by induction on n, that $G=G_{c_1};\ldots;G_{c_n}$ is fan-in free, and has exactly K arcs connecting the DP parameters, where one such arc is labelled \downarrow (due to $x'\stackrel{\downarrow}{\to} y'\in G_c$). Thus Γ has exactly K arcs connecting the DP parameters, where one such arc is labelled \downarrow . It follows from $\Gamma;\Gamma=\Gamma$ that each of these arcs has the form $z'\stackrel{\gamma}{\to} z'$. Therefore G has some arc of the form $z'\stackrel{\downarrow}{\to} z'$. By definition of size-change graphs, z' is bounded, so there exists a bounded z' such that $z'\stackrel{\downarrow}{\to} z'\in \Gamma$, which contradicts an assumption about α .

The idea then is to look for such a "problematic" \mathcal{D}^* . We begin by considering maximal strongly-connected subgraphs (MSCSs) of \mathcal{D} . An MSCS without both a vertex $x \in X_1$ and a constructive arc (an \uparrow -labelled one) need not be considered; its arcs can be removed from \mathcal{D} . Otherwise, if an MSCS has an arc $x \stackrel{\delta}{\leadsto}_c y$ such that G_c contains a decreasing arc between $DP(\mathcal{G}')$ parameters, where \mathcal{G}' contains the size-change graphs for calls appearing in the MSCS, it is not hard to show that $x \stackrel{\delta}{\leadsto}_c y$ cannot be in any problematic \mathcal{D}^* , so it can be removed from \mathcal{D} . If after repeatedly removing arcs from \mathcal{D} , no strongly-connected subgraph remains, then no problematic \mathcal{D}^* exists. It follows that Condition 2 in Theorem 1 holds.

5.1 Collecting Bounded Variables in Polynomial Time

A set of bounded variables can be collected as follows.

- 1. Work out the SCCs of \mathcal{D} in reverse topological order [2].
- 2. For each SCC X not marked as *processed*, mark it so, and call the procedure Classify below with the subgraph \mathcal{D}' of \mathcal{D} corresponding to X.
 - (a) If Classify succeeds, mark all the elements of X as bounded.
 - (b) Otherwise mark all the elements of X as dubious, and for each SCC X_1 reachable from X, mark X_1 as processed, and mark its elements as dubious.

procedure $Classify(\mathcal{D}')$

- 1. If \mathcal{D}' has no \uparrow arc, just return, else continue.
- 2. Compute $C' = \{c \mid x \stackrel{\delta}{\leadsto}_c y \text{ in } \mathcal{D}'\}.$

- 3. Compute $\mathcal{G}' = \{G_c \mid c \in C'\}$, based on currently known bounded variables.
- 4. Compute DP' = DP(G').
- 5. Compute $C^{\downarrow} = \{c \mid x', y' \in DP', x' \xrightarrow{\downarrow} y' \in G_c\}.$
- 6. If $C^{\downarrow} = \{\}$, fail, else continue.
- 7. Compute \mathcal{D}'' , subgraph of \mathcal{D}' excluding any arc $x \overset{\delta}{\leadsto}_c y$ where $c \in C^{\downarrow}$.
- 8. Recursively call Classify on each MSCS of \mathcal{D}'' .

Lemma 3. A variable classified as bounded is bounded.

Proof. Consider a successful Classify (\mathcal{D}') call, where \mathcal{D}' is an MSCS of \mathcal{D} . Let X_1 be the set of \mathcal{D}' vertices. Then for $z \in X_1$, $X_1 = \{x \mid x \leadsto^+ z, z \leadsto^+ x\}$. For $z \in X_1$, we claim that Condition 2 in Theorem 1 is satisfied. The lemma follows easily from this.

Suppose that Condition 2 in Theorem 1 does *not* hold. It follows from Theorem 3 that there exists a strongly-connected subgraph \mathcal{D}^* of \mathcal{D}' such that \mathcal{D}^* has some arc of the form $x \stackrel{\wedge}{\leadsto}_c y$, and for $C^* = \{c \mid x \stackrel{\delta}{\leadsto}_c y \text{ in } \mathcal{D}^*\}$ and $\mathcal{G}^* = \{G_c \mid c \in C^*\}$, there do not exist $x', y' \in DP(\mathcal{G}^*)$ and $c \in C^*$ such that $x' \stackrel{\downarrow}{\to} y' \in G_c$. We argue that this would cause $Classify(\mathcal{D}')$ to fail, which contradicts our assumption.

Initially, the input to Classify includes \mathcal{D}^* as a subgraph. Now, if input \mathcal{D}' to Classify includes \mathcal{D}^* and the call does not fail, then for $C' = \{c \mid x \stackrel{\delta}{\leadsto}_c y \text{ in } \mathcal{D}'\}$, $\mathcal{G}' = \{G_c \mid c \in C'\}$ and $C^{\downarrow} = \{c \mid x', y' \in DP(\mathcal{G}'), x' \stackrel{\downarrow}{\to} y' \in G_c\}$, we deduce that C^{\downarrow} is non-empty, and that any c in the set C^{\downarrow} does not occur in C^* (because $DP(\mathcal{G}') \subseteq DP(\mathcal{G}^*)$ for $\mathcal{G}^* \subseteq \mathcal{G}'$). We conclude that \mathcal{D}^* is included in an MSCS of the graph that remains, after removing from \mathcal{D}' any arcs of the form $x \stackrel{\delta}{\to}_c y$ where $c \in C^{\downarrow}$, i.e., \mathcal{D}^* is included in the argument of a recursive call to Classify. Since each recursive call reduces the argument size, either the procedure fails, or eventually some call has the argument \mathcal{D}^* . This call must fail.

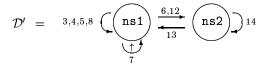
Lemma 4. Bounded variables are collected in cubic time in the worst case.

Proof. Both the dependency graph \mathcal{D} and the size-change graphs are O(N) in size, where N is the program size. For a call to Classify with argument \mathcal{D}' of size M, the recursion depth is bounded by M. At each recursion level, we calculate at most M DP sets, each taking time O(N), and perform other linear-time operations on non-overlapping subproblems. Loosely, all the Classify calls at each recursion level takes time O(MN). Thus calling Classify with \mathcal{D}' has time complexity $O(M^2N)$. Finally, the procedure to collect bounded variables has time complexity of the order $\Sigma(M_i^2N)$, where $\Sigma M_i = N$. This is $O(N^3)$. \square

The interpreter example. goal parameters are always bounded, and can be ignored. We begin with the dependency graph \mathcal{D} in Fig. 3.1.

- 1. The SCCs of \mathcal{D} , in reverse topological order, are $\{p1, p2\}$, $\{e1, es2\}$ and $\{ns1, ns2\}$.
- 2. The subgraph of \mathcal{D} corresponding to the component $\{p1, p2\}$ has no \uparrow dependencies, so classify p1 and p2 as bounded.

- 3. Similarly, classify e1 and es2 as bounded.
- 4. The subgraph of \mathcal{D} corresponding to the remaining component is depicted below. The corresponding set of calls is $C' = \{3,4,5,6,7,8,12,13,14\}$ (note the absence of 9). The set of G_c for $c \in C'$ has DP-set $\{e1, es2, p1, p2\}$, since each of these variables is connected to another one of the set in each G_c . Therefore, $C^{\downarrow} = C'$, since each G_c has one of the arcs: $e1 \xrightarrow{\downarrow} e1$, $e1 \xrightarrow{\downarrow} es2$, $es2 \xrightarrow{\downarrow} es2$, or $es2 \xrightarrow{\downarrow} e1$. The variables ns1 and ns2 are thus classified as bounded.



We see that all the static parameters are bounded for the interpreter example. Observing arbitrarily long sequences of constructions on ns1 values is impossible, because that would imply observing arbitrarily long sequences of size decreases in the values of the provably bounded e1. This is detected by the procedure. The interest in the example is in function call 9, which "resets" the value of e1. This seems dangerous, since e1 is used to "counter" the constructions on n1 values due to function call 7. Fortunately, the interpreter implements lexical scoping, which means the value of ns1 is reset along with e1. For an interpreter implementing dynamic scoping, the value of ns1 would not be reset with e1, and our analysis would safely indicate that ns1 might not be bounded.

6 Concluding Remarks

We have presented an approach to perform finiteness analysis in polynomial-time. As with such analyses in the literature [7,4,1,3], boundedness deductions are based on establishing that arbitrarily long sequences of constructions on a static variable's values would imply arbitrarily long sequences of size decreases in some bounded-variable values, which is impossible. However, it focuses on realistic descent. Related experiments [12] support this claim, and suggest that an analysis of variable boundedness based on Theorem 3 will be much faster than one based on Theorem 1, which captures the usual boundedness principle. Theorem 1 incorporates a condition that is *intrinsically* hard.

It is immediate future work to design and conduct experiments to test the effectiveness, efficiency and scalability of the proposed analysis.

Acknowledgements

Thanks go to Carl Frederiksen for his help with the paper at the eleventh hour.

References

- 1. Peter Holst Andersen and Carsten Kehler Holst. Termination analysis for offline partial evaluation of a higher-order functional language. In Static Analysis, Proceedings of the Third International Symposium, SAS '96, Aachen, Germany, Sep 24-26, 1996, volume 1145 of Lecture Notes in Computer Science, pages 67-82. Springer, 1996.
- Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. Introduction to Algorithms. MIT Press, 1990.
- 3. Arne J. Glenstrup. Terminator II: Stopping partial evaluation of fully recursive programs. Master's thesis, DIKU, University of Copenhagen, Denmark, 1999.
- 4. Arne J. Glenstrup and Neil D. Jones. BTA algorithms to ensure termination of off-line partial evaluation. In Perspectives of System Informatics, Proceedings of the Second International Andrei Ershov Memorial Conference, Russia, Jun 25-28, 1996, volume 1181 of Lecture Notes in Computer Science, pages 273-284. Springer, 1996.
- Robert Glück. Towards multiple self-application. In Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation, pages 309–320. ACM, 1991.
- Matthew Hennessy. The Semantics of Programming Languages. Wiley, 1990.
- Carsten Kehler Holst. Finiteness analysis. In John Hughes, editor, Functional Programming Languages and Computer Architecture, Cambridge, Massachusetts, Aug 1991, volume 523 of Lecture Notes in Computer Science, pages 473

 495. Springer, 1991.
- 8. Neil Jones and Arne Glenstrup. Program generation, termination, and binding-time analysis. In *Principles, Logics, and Implementations of High-Level Programming Languages, PLI 2002 (invited paper)*. Springer, 2002.
- 9. Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. Partial Evaluation and Automatic Program Generation. Prentice Hall, 1993.
- 10. Chin Soon Lee. Partial evaluation of the euclidean algorithm, revisited. *Higher-Order and Symbolic Computation*, 12(2):203–212, Sep 1999.
- 11. Chin Soon Lee. Program Termination Analysis, and Termination of Offline Partial Evaluation. PhD thesis, UWA, University of Western Australia, Australia, 2001.
- 12. Chin Soon Lee. Program termination analysis in polynomial time. In *Proceedings of Generative Programming and Component Engineering, GPCE '02.* Springer, 2002.
- Chin Soon Lee, Neil D. Jones, and Amir Ben-Amram. The size-change principle for program termination. In Proceedings of the ACM Symposium on Principles of Programming Languages, Jan 2001. ACM, 2001.
- 14. Wim Vanhoof and Maurice Bruynooghe. Binding-time annotations without binding-time analysis. In Logic for Programming, Artificial Intelligence, and Reasoning (LPAR), 8th International Conference, Havana, Cuba, Dec 3-7, 2001, volume 2250 of Lecture Notes in Computer Science, pages 707–722, 2001.

A Correctness of the Boundedness Principle

Lemma 5. Composition of abstract transitions is associative.

Definition 16. Let $seq = \alpha_1, \ldots, \alpha_T = f_0 \xrightarrow{\Delta_1, \Gamma_1} f_1, \ldots, f_{T-1} \xrightarrow{\Delta_T, \Gamma_T} f_T$ be a sequence of abstract transitions.

- 1. A δ -thread of seq from x_0 to x_T is an arc sequence $x_0 \xrightarrow{\delta_1} x_1, \ldots, x_{T-1} \xrightarrow{\delta_T} x_T$ such that for $t = 1, \ldots, T$, $x_{t-1} \xrightarrow{\delta_t} x_t \in \Delta_t$. The δ -thread is constructive if for some $t \in [1, T]$, $\delta_t = \uparrow$.
- 2. A γ -thread of seq from x_0 to x_T is an arc sequence $x_0 \stackrel{\gamma_1}{\to} x_1, \ldots, x_{T-1} \stackrel{\gamma_T}{\to} x_T$ such that for $t = 1, \ldots, T$, $x_{t-1} \stackrel{\gamma_t}{\to} x_t \in \Gamma_t$. The γ -thread is decreasing if for some $t \in [1, T]$, $\gamma_t = \downarrow$.

Lemma 6. Let $seq = \alpha_1, \ldots, \alpha_T = f_0 \xrightarrow{\Delta_1, \Gamma_1} f_1, \ldots, f_{T-1} \xrightarrow{\Delta_T, \Gamma_T} f_T$ be a sequence of abstract transitions. Note that the sequence can be composed. Let the composition be $\alpha = f \xrightarrow{\Delta, \Gamma} g$. Composition preserves thread information in the following sense:

- 1. seq has a constructive δ -thread from x to y just when $x \stackrel{\uparrow}{\to} y \in \Delta$.
- 2. seq has a decreasing γ -thread from x to y just when $x \stackrel{\downarrow}{\rightarrow} y \in \Gamma$.

Theorem 1. We are given z with $\tau(z) = S$. Let $X_0 = \{x \mid x \rightsquigarrow^+ z, z \not \rightsquigarrow^+ x\}$ and $X_1 = \{x \mid x \rightsquigarrow^+ z, z \rightsquigarrow^+ x\}$. Then z is bounded if

- 1. for $x \in X_0$, x is bounded, and
- 2. for $x \in X_1$, $\alpha = f \xrightarrow{\Delta, \Gamma} f \in \overline{\mathcal{T}}$ such that $\alpha; \alpha = \alpha$ and $x \xrightarrow{\uparrow} x \in \Delta$, there exists some bounded x' for which $x' \xrightarrow{\downarrow} x' \in \Gamma$.

Proof. Let a respect the annotations of $f \in Fun$. Consider the following set of values: $\{(b)_k \mid (f, a) \Rightarrow^* (g, b)\}$, where $z \equiv g^{(k)}$. We must show that it is finite. Define the set V_0 of input values and values of known bounded variables.

$$V_0 = \{(\boldsymbol{b})_i \mid (f, \boldsymbol{a}) \Rightarrow^* (g', \boldsymbol{b}), \ g'^{(i)} \in X_0\} \cup \{(\boldsymbol{a})_i \mid \tau(f^{(i)}) = S\}$$

Define $close: \wp(\mathit{Val}) \to \wp(\mathit{Val})$ such that close(V) is the least downward-closed set including V.

$$close(V) = \{u \mid u \prec v, \ v \in V\}$$

Define $build : \wp(Val) \to \wp(Val)$ as follows.

$$\begin{aligned} build(V) = \{u \mid te \text{ in } te^{g'}, \ te : S, \ \pmb{b} \text{ respects annotations of } g', \\ g'^{(i)} \text{ in } te \Rightarrow (\pmb{b})_i \in V, \ \pmb{b}, te \Downarrow u \} \end{aligned}$$

The intended meaning of build(V) is the set of values that can be constructed in one computation step using values in V. Finally, define inductively a sequence $(U)_i$ for $i \in \mathbb{N}$ as follows.

$$U_0 = close(V_0 \cup build(\{\}))$$

 $U_{i+1} = close(U_i \cup build(U_i))$

Note that each of U_i is finite.

Let transition sequence $(f, \mathbf{a}) = (g_0, \mathbf{b}^{(0)}) \Rightarrow \ldots \Rightarrow (g_T, \mathbf{b}^{(T)}) = (g, \mathbf{b})$ be safely described by $\alpha_1, \ldots, \alpha_T \in \mathcal{T}$. If for $t = 0, \ldots, T - 1$, every δ -thread $x_t \stackrel{\delta_{t+1}}{\to} x_{t+1}, \ldots, x_{T-1} \stackrel{\delta_T}{\to} x_T$ of $\alpha_{t+1}, \ldots, \alpha_T$ such that $x_{t+1}, \ldots, x_T \notin X_0$ and $x_T = g^{(k)}$ has no more than $N \uparrow$ -labels, then $(\mathbf{b})_k \in U_N$. This follows easily by induction on N and T. We omit the details.

Thus, for $g^{(k)}$ to violate boundedness, for any N, there exists transition sequence $seq = (f, \mathbf{a}) = (g_0, \mathbf{b}^{(0)}) \Rightarrow \ldots \Rightarrow (g_T, \mathbf{b}^{(T)}) = (g, \mathbf{b})$, safely described by $\alpha_1, \ldots, \alpha_T \in \mathcal{T}$, such that some δ -thread $x_t \stackrel{\delta_{t+1}}{\longrightarrow} x_{t+1}, \ldots, x_{T-1} \stackrel{\delta_T}{\longrightarrow} x_T$ of $\alpha_{t+1}, \ldots, \alpha_T$ with $x_{t+1}, \ldots, x_T \notin X_0$ and $x_T = g^{(k)}$ contains no fewer than N \uparrow -labels. We show this leads to a contradiction, establishing that $g^{(k)}$ is bounded.

By making N large enough, it is possible to observe in $\alpha_{t+1}, \ldots, \alpha_T$ a δ -thread of the form $\ldots \xrightarrow{\uparrow} x \ldots \xrightarrow{\uparrow} x \ldots \xrightarrow{\uparrow} x$ for some $x \in X_1$, with arbitrarily many $x \ldots \xrightarrow{\uparrow} x$ segments. By Lemma 6, each part of $\alpha_{t+1}, \ldots, \alpha_T$ corresponding to an $x \ldots \xrightarrow{\uparrow} x$ segment composes to some α containing $x \xrightarrow{\uparrow} x$. Let the number of $x \ldots \xrightarrow{\uparrow} x$ segments be M. Then M can be made arbitrarily large by taking N large enough. Let the parts of $\alpha_{t+1}, \ldots, \alpha_T$ corresponding to the $x \ldots \xrightarrow{\uparrow} x$ δ -thread segments be $\alpha^{(1)}, \ldots, \alpha^{(M)}$ (each $\alpha^{(i)}$ is a sequence of abstract transitions).

For t' < t'', let $\{t', t''\}$ be in P_{α} if the abstract transitions of $\boldsymbol{\alpha}^{(t'+1)}, \ldots, \boldsymbol{\alpha}^{(t'')}$ compose to α . Then the set of P_{α} , for $\alpha \in \overline{\mathcal{T}}$ whose Δ graph has some $x \stackrel{\uparrow}{\to} x$ with $x \in X_1$, partitions the set of pairs of distinct numbers in [0, M]. According to Ramsey's Theorem, by taking M large enough, for any K, we can find a set of numbers $t_0 < \ldots < t_K$ that is homogeneous, i.e., for some $\alpha \in \overline{\mathcal{T}}$ whose Δ graph has $x \stackrel{\uparrow}{\to} x$ with $x \in X_1$, every pair of distinct numbers $\{t_i, t_j\}$, where $i, j \in [0, K]$, is in (the one) P_{α} . By definition of P_{α} , the abstract transitions of $\boldsymbol{\alpha}^{(t_i+1)}, \ldots, \boldsymbol{\alpha}^{(t_j)}$ compose to α for $i, j \in [0, K]$ and i < j. In particular, the abstract transitions of $\boldsymbol{\alpha}^{(t_0+1)}, \ldots, \boldsymbol{\alpha}^{(t_0+1)}, \ldots, \boldsymbol{\alpha}^{(t_1)}$, those of $\boldsymbol{\alpha}^{(t_1+1)}, \ldots, \boldsymbol{\alpha}^{(t_2)}$, and those of $\boldsymbol{\alpha}^{(t_0+1)}, \ldots, \boldsymbol{\alpha}^{(t_2)}$ compose to α . We deduce that $\alpha; \alpha = \alpha$ by the associativity of composition.

By the premise in the theorem, α has a Γ graph with some decreasing arc $x' \stackrel{\downarrow}{\to} x'$ for a bounded x'. Now for each $i \in [0, K-1]$, the composition of the abstract transitions of $\boldsymbol{\alpha}^{(t_i+1)}, \ldots, \boldsymbol{\alpha}^{(t_{i+1})}$ is α . So by Lemma 6, the sequence of abstract transitions $\boldsymbol{\alpha}^{(t_i+1)}, \ldots, \boldsymbol{\alpha}^{(t_{i+1})}$ has a decreasing γ -thread from x' to x'. We deduce the existence of a γ -thread in a final section of $\alpha_1, \ldots, \alpha_T$ with $K \downarrow$ -labels, indicating that the \Rightarrow transition sequence described by $\alpha_1, \ldots, \alpha_T$ exhibits a sequence of K decreasing x' values. Since K can be made arbitrarily large but x' is bounded, we have the required contradiction.

B Program and Specializer Semantics

Figure B.1 contains the evaluation rules for L expressions. Figure B.2 and B.3 contain respectively the well-annotatedness rules and reduction rules for 2L expressions.

$$egin{aligned} egin{aligned} egin{aligned} oldsymbol{u} & oldsymbol{u} & oldsymbol{v}, c \downarrow u & oldsymbol{v}, f^{(i)} \downarrow v_i \end{aligned} \ egin{aligned} oldsymbol{v}, e_1 \downarrow \operatorname{true} & oldsymbol{v}, e_2 \downarrow u & oldsymbol{v}, if e_1 e_2 e_3 \downarrow u \end{aligned} \ egin{aligned} oldsymbol{v}, if e_1 e_2 e_3 \downarrow u & oldsymbol{v}, if e_1 e_2 e_3 \downarrow u \end{aligned} \ egin{aligned} oldsymbol{v}, e_1 \downarrow u_1 & \dots & oldsymbol{v}, e_n \downarrow u_n & u = \mathcal{O}[\![op]\!](u_1, \dots, u_n) \\ \hline oldsymbol{v}, op(e_1, \dots, e_n) \downarrow u \end{aligned} \ egin{aligned} oldsymbol{v}, e_1 \downarrow u_1 & \dots & oldsymbol{v}, e_n \downarrow u_n & (u_1, \dots, u_n), e^f \downarrow u \\ \hline oldsymbol{v}, f(e_1, \dots, e_n) \downarrow u \end{aligned}$$

Figure B.1: Evaluation of L expressions

Figure B.2: Well-annotatedness of 2L expressions

Figure B.3: Reduction of 2L expressions