PROGRAM TERMINATION ANALYSIS AND TERMINATION OF OFFLINE PARTIAL EVALUATION

This thesis is
presented to the
Department of Computer Science
for the degree of
Doctor of Philosophy
of the
University of Western Australia

By
Chin Soon Lee
March 2001
Revised August 2002

Abstract

The automatic analysis of program termination is of great interest in the context of software verification. This thesis presents a novel approach to termination analysis, of particular interest because of its connection with the termination of Offline Partial Evaluation, a popular paradigm for program specialization. This connection is pursued.

In our POPL '01 paper, we proposed the general *size-change principle* for program termination.

> An infinite computation is *impossible*, if it would give rise to an infinite sequence of size decreases for some data values.

In this work, we propose a principle, also based on size descent, for ensuring that variables assume boundedly many values during specialization. It properly subsumes the conditions seen in the literature.

> Unbounded sequences of increases in a variable are *impossible*, if they would give rise to unbounded sequences of size decreases for some bounded-variable values.

The boundedness of variables is key to the termination of specialization in offline partial evaluation.

Dataflow analysis is used to collect a set of *size-relation graphs*, bipartite graphs that describe the data size relations in possible program state transitions. Decidable conditions are formulated in terms of these graphs, based on the principles above. In this way, both analysis of program termination and variable boundedness can share abstract domains, supporting analyses, and conditions for detecting size descent. Our study leads to a *safe* strategy for the offline partial evaluation of a higher-order functional language.

Part I of this thesis is about program termination analysis. The very general size-change principle accounts for size descent in indirect recursion, among permuted arguments, and allows different descent for different function-call sequences.

For an actual analysis, size-relation graphs are used to describe the data size relations in possible program state transitions. A graph sequence that respects program control flow is called a *multipath*. The set of multipaths describe possible state transition sequences. To show that such sequences are finite, it is sufficient that the graphs satisfy *size-change termination (SCT)*: Every infinite multipath has infinite descent, according to the arcs of the graphs. This is an application of the size-change principle.

SCT is a decidable condition, but unsurprisingly, complete for PSPACE. An efficient PTIME approximation is proposed. For size-relation graphs that can loop (i.e., they give rise to an infinite multipath), and that satisfy SCT, it is usually easy to find a "progress point"— some graph whose infinite occurrence in a multipath causes infinite descent. SCT is established if after repeatedly removing such graphs, the remaining set cannot loop. We argue that the approximation is effective in practice.

Part II of this thesis is about variable boundedness analysis. By a development analogous to Part I, conditions are formulated for testing the boundedness of a variable. We establish that existing analyses use a PSPACE-complete condition. We then model a condition after the PTIME termination criterion. It accounts for *realistic* parameter size-change behaviour.

It is a long-standing problem to achieve the termination of offline partial evaluation while maintaining a good degree of specialization. This work is a step towards a practical solution.

Acknowledgements

I would like to express my deepest gratitude to Prof Neil Jones for so generously sharing his time, his experience and his research with me. When I first visited TOPPS, DIKU back in May '99, my hope was to learn about doing good research in program analysis by interacting with researchers whose work I had read and admired. To be frank, I had not expected Neil to take such an active interest in my academic development. For this, I continue to be grateful. Naturally, it was difficult to unlearn many bad research habits in the span of a few months. So even while the entirety of this work is based on research conducted at DIKU from May '99 to Jan '00, its various shortcomings remain solely my responsibility.

Thanks to Prof Olivier Danvy for his patient guidance with the GCD paper, and for setting up the DIKU visit, which would nevertheless not have eventuated had it not been for his encouragement. Thanks to Amir Ben-Amram for solving all the complexity questions I posed. And to everyone at TOPPS for a rewarding visit.

Back at UWA, I gratefully acknowledge the financial support of the OPRS and University Postgraduate Scholarship for the first three years of my degree[1]. Thanks to Prof Robyn Owens for her support, and Prof C. P. Tsang for getting me started in program analysis. My former advisor and dear friend, Dr Alan Woods, my coursemates, especially roommate Dong-Tsan Lee and my cousin Chu-Cheow Lim deserve awards for submitting themselves to my ceaseless whinging. Last but not least, thanks to my family and friends, who have put up with me all this time.

[1]The support of the DAEDALUS project during the revision of this work is also gratefully acknowledged. DAEDALUS is the RTD project IST-1999-20527 of the IST Programme within the European Union's Fifth Framework Programme.

Contents

Chapter 1

General Introduction

1.1 What this work is about

This work is concerned with *two* issues: termination of programs, and termination of the *offline partial evaluation* of programs. The latter reduces to the question of whether the subject program's variables assume boundedly many values during specialization. Hence, the title of this work, "Program Termination Analysis, and Termination of Offline Partial Evaluation," refers to two distinct, though related, problems.

A close relationship exists between program termination analysis and variable boundedness analysis. For first-order programs, Glenstrup [Gle99] has pointed out a straightforward application of variable boundedness analysis to termination. A simplified version: Add a "recursion depth" parameter to every function, which is incremented at every call. These parameters bound the program's recursion depth. If it can be established that they are bounded in every execution, then the program is terminating.

On the other hand, it is less clear how the works on automatic termination analysis can be brought to bear on the more difficult question of variable boundedness. It makes sense therefore to present the study of automatic termination analysis in a self-contained manner, before tackling variable boundedness and the termination of offline partial evaluation.

No specialized knowledge will be assumed in this work, apart from a basic familiarity with functional programming, and some offline partial evaluation. For offline partial evaluation, a thoroughly recommended introductory text is Jones, Gomard and Sestoft's textbook [JGS93], now available for download at `http://www.diku.dk/~neil/`.

1.2 Background of this work

This research originated as a study into the problem of potential non-termination for the offline partial evaluation of a functional language.

1.2.1 Non-termination of offline partial evaluation: A quick overview

Readers familiar with offline partial evaluation will be familiar with the following introduction.

> Offline partial evaluation is an *automatic* technique for program specialization. It involves the application of the following transformations: symbolic simplification, function specialization (memoization), and call unfolding. Given any subject program, and part of its input, the specializer generates a *residual program*, so that when the residual program is provided with the remaining input, it produces the same output as the original program given all of its input at once. Naturally, it is hoped that the specialized program will be more efficient than the original.

This is a good description of the nature of partial evaluation. The only contentious issue is the automatic nature of offline partial evaluation. As there is no provision for user guidance in its execution, in design, offline partial evaluation is *intended* as an automatic tool. However, the price for aggressive specialization, as we will see, is potential non-termination. This places the burden of ensuring termination on the author of the subject program. Current offline partial evaluators therefore fall short of the goal of an automatic tool: to permit blind, or "black-box" application to any subject program.

In practice, the problem is not completely detrimental to the deployment of offline partial evaluators. Let us consider the issue more closely. An introductory text continues as follows.

> Offline partial evaluation is implemented in two separate phases. In the first phase, it is determined which program variables' values will be available during specialization. This is accomplished by dataflow analysis. Available and unavailable parameters are called *static* and *dynamic* respectively. For a classification of variables to be usable by the specializer, the computation of static arguments must not depend on any dynamic variable. This is called the principle of *congruence*. The *division* of variables as static or dynamic determines which constructs are specialized away, and which remain in the residual program.

Such an explanation is typically followed by the ubiquitous example to compute *integer powers*. (In our programming examples, we employ a Haskell-like syntax [Tho96].)

```
pow x n = if n = 0 then 1
          else x * (pow x (n - 1))
```

Suppose we can determine that the `pow` function is only invoked in the subject program with `n` having the value `3` to compute cubes. It makes sense to create a specialized version of `pow` for this purpose, in which the recursive call in the body of `pow` is unfolded away, eliminating the conditional test `n = 0` completely. This leads to constant-time savings. Offline partial evaluation achieves it automatically.

Some partial evaluation terminology: Variables are classified as *static* or *dynamic*, roughly corresponding to "known" and "unknown" at specialization time. The classification of a variable is called its *binding-time*, or BT for short. A BT classification of a set of program variables is called a *division*. For example, a subject program's main function, with arity 2, may have *initial division* (D, S), where D and S represent dynamic and static respectively. The initial division induces a division of all of the program's variables, based on the principle of congruence: A variable x must be classified D if the argument expression e may be assigned to it during specialization, where the computation of e involves a dynamic variable. Variables are *generalized* as dynamic in a fixedpoint computation by repeated application of the above principle until a stable division is reached. The final division is said to satisfy *congruence*.

In the `pow` example, if every reference to `pow` has the form `(pow ... 3)`, then a congruent division can safely classify `n` as static: during specialization of `pow`, the value of `n` is available. This means the test in `pow`'s body can be performed statically, and the `if` construct can be reduced away at specialization time. We may indicate this by marking the `if` as $\texttt{if}^{\texttt{S}}$ (the `S` stands for "static"). In the same way, we mark every construct with `S` or `D`:

$$\texttt{pow x n = if}^{\texttt{S}}\texttt{ (n =}^{\texttt{S}}\texttt{ 0) then (lift 1)}$$
$$\texttt{else x *}^{\texttt{D}}\texttt{ (pow}^{\texttt{S}}\texttt{ x (n -}^{\texttt{S}}\texttt{ 1))}$$

The `lift` corresponds to a casting operation, converting a static value into an expression for the residual program. The constant in the above function is *lifted* to match the binding time of the other conditional branch, which is necessary for syntax-directed specialization. The marking of `S` on `pow` is interpreted as indicating reduction *and* unfolding for the application– in this example, unfolding is safe. A syntax-directed specializer can now specialize the call `(pow x 3)` as `(pow-3 x)`, where:

```
pow-3 x = x * x * x * 1
```

The example illustrates the notion of congruence: The arguments assigned to `n` during specialization do not involve any dynamic variable, thus `n` is classified static, and its values are used by the specializer.

The principle of congruence captures the condition on the BT division that enables fast, syntax-directed specialization. The price for this efficiency, unfortunately, is the amount of specialization. For example, reducing the body of `pow` with `n` having the value `0` results in the *expression* `1` (rather than the *value* of `1`). The result therefore cannot participate in further computation.

To understand how to avoid ineffective specialization, we must consider the principle of congruence. This leads at once to the general piece of advice: "Maintain a good separation of binding-times." Suppose the body of `pow` has another recursive call of the form `(pow x (if (x = 0) 0 (n - 1)))`, then the slight reformulation `(if (x = 0) (pow x 0) (pow x (n - 1)))` is to be preferred because it has better BT separation, which can lead to improved specialization.

As another example, consider the following function, which fetches some item corresponding to `key` in the association list `a-list`, and increments it.

```
f key = 1 + (lookup key [("key0",0), ("key1",1), ("key2",2)])

lookup key a-list = if fst (hd a-list) = key
                    then snd (hd a-list)
                    else lookup key (tl a-list)
```

It is easy to reason that the function `f` does not specialize well with `key` dynamic, as the result of `lookup` should be classified dynamic by the principle of congruence. For this example, it is also easy to see that the result of `lookup` assumes boundedly many values. Generally, this is sufficient to allow the program to be rewritten for effective specialization [JGS93, DG97, Lee99]: By reformulating the program as follows,

```
f key = lookup1 key [("key1",0),("key2",1"),("key3",2)]

lookup1 key a-list = if fst (hd a-list) = key
                     then 1 + snd (hd a-list)
                     else lookup1 key (tl a-list)
```

the computation of `1 +` is now performed at specialization-time.

Harder to reason about is whether variables assume boundedly many values during specialization. Unfortunately, this issue does not only concern the effectiveness of specialization, it concerns its *safety*. In particular, infinite function specialization arises from unbounded static computation under dynamic control. The behaviour is demonstrated by the following program fragment, where `s` has been classified static and `d` dynamic.

```
f s d = if s = d then s
        else f (1 + s) d
```

For any natural number in `d`, `(f 0 d)` normally returns the re-constructed value of `d`. The specializer will attempt to create versions of `f` with `s` having the values $0, 1, 2, 3, \ldots$.

Infinite function specialization can be prevented by ensuring that *static* variables assume boundedly many values during specialization. What is called for is a *general principle* for variable boundedness, and a study that will lead to an efficient and effective condition to determine bounded variables. Such a condition will decide, for instance, that `a-list` is bounded for the `lookup1` example, but safely classify `s` as (possibly) unbounded for the function `f` above.

It is a long-standing problem to devise a practical means to achieve the termination of offline partial evaluation, while maintaining a good degree of specialization. The long-term goal (and motivation) of our research is the design of a termination module for an actual offline partial evaluator such as Similix [Bon91b].

1.2.2 A principle behind the methods

A number of articles have been written about ensuring the termination of offline partial evaluation. Inspired by Jones' "re-examination" paper [Jon88], Holst proposed sufficient conditions to determine that every program variable assumes boundedly many values during evaluation (however long an execution runs) [Hol91]. Such a program is called *quasi-terminating*. (Application to offline partial evaluation: If a program is quasi-terminating in the static variables, then infinite function specialization is impossible, given sufficient function memoization.)

The only intuition given for the conditions of [Hol91] is that they constitute a liberal way to bound the "growth" in an argument. Another attempt was made to explain the conditions in [AH96]: "If whenever something grows, something decreases, then nothing grows unboundedly" (i.e., if whenever an argument is observed to grow over a sequence of function calls, another argument is observed to decrease in size, then no argument grows unboundedly). This suggests that it is *infinite* growth that is prohibited by the conditions, which is not correct. Consider the following example:

```
f x y = if x = 0 then ¹f (y + 1) 0
        else ²f (x - 1) (y + 1)
```

where the function calls have been labelled for easy reference. The only call sequence where an argument grows infinitely ends in infinitely many 2s, since any call 1 resets the computation in `y`. Such call sequences are impossible because the value of `x` would decrease infinitely in size. As every infinite growth is accompanied by infinite size descent, we would like to conclude that the program variables are bounded. However, this would be incorrect: Both x and y assume unboundedly many values for the execution starting with x and y having the value 0.

In fact, the conditions of [Hol91] prohibit unbounded *sequences* of growth in an argument. Continuing with the example above, the call sequences: 2, 22, 222, ... , starting with (x, y) having the values $(1,0),(2,0),(3,0),\ldots$ respectively, exhibit *unbounded* (rather than infinite) growth in the second argument of f. The conditions of [Hol91] deduce that such sequences would give rise to unbounded size descent in values of x. This would be impossible *if* x *has been determined to be bounded.* Thus for the slightly modified definition of f below:

```
f x y = if x > 42 then 0
        else if x = 0 then f (y + 1) 0
             else f (x - 1) (y + 1)
```

the result of [Hol91] establishes the boundedness of x and y.

This leads to the somewhat puzzling slogan: If whenever something grows, something *bounded* decreases, then nothing grows unboundedly (which is in fact, the same message presented on page 301 of [JGS93], except there, the authors have cleverly refrained from calling a variable static, until it is proved to be bounded, allowing them to state the more reasonable-sounding: If whenever something grows, something *static* decreases, then nothing grows unboundedly.)

Such a slogan helps one recall the conditions of [Hol91], but does not really reflect a good understanding of why they work. It does not suggest how to generalize the conditions, except perhaps in an ad hoc way, e.g.: If whenever an argument grows, a set of bounded arguments have sizes whose sum decreases, then nothing grows unboundedly. Other important questions have not been addressed. For example, the straightforward implementation of the conditions of [Hol91] is exponential-time. In [AH96], the authors expressed the belief that they were not up against *intrinsic* complexity problems. So what *is* the intrinsic complexity of the problem? Are there better algorithms/ good approximations in practice?

Even though methods based on [Hol91] are powerful enough not to over-generalize static parameters for many programs of interest (particularly interpreters), questions about effectiveness and efficiency must be tackled seriously for a convincing solution to the variable boundedness problem. We believe a good starting point is to state a *general principle* for variable boundedness, based on observing size decreases among parameter values. Such a principle captures a *reasoning* that can be used to establish variable boundedness.

> Unbounded sequences of increases in an argument are *impossible*, if they would give rise to unbounded sequences of size decreases for some bounded-variable values.
>
> Application to specialization: Unbounded sequences of increases in a *static* argument are *impossible*, if they would give rise to unbounded sequences of size decreases for some bounded-static-variable values.

The close study of the principle above that we will conduct represents a significant departure from all the works following from [Hol91]. While these works have advanced the theory of quasitermination in various directions, the conditions for variable boundedness have remained in essence the same. Jones et al. [JGS93] formulated explicit conditions for determining bounded variables based on known bounded variables. Their conditions can be iterated to harness the true power of Holst's quasitermination theorem. (For example, an adaptation of Jones et al.'s approach is able to prove the termination of the Ackermann function *automatically* by proving that the recursion depth is bounded.) Andersen and Holst [AH96] extended Holst's approach to a higher-order language. Glenstrup and Jones [GJ96] and Glenstrup [Gle99] obtained a cleaner formulation of the approach of [JGS93]. A clear separation of concerns between the abstract interpretation of parameter size relations, and the application of boundedness conditions results in an overall more powerful method. However, the contention of this work is that it is difficult to qualify exactly what the gains are without a study aimed at understanding the underlying principle thoroughly.

The variable boundedness principle above is admittedly difficult to apply, because it refers to known bounded variables. Practically, this means having to formulate explicit conditions to collect bounded variables (as in [JGS93]), or settling for a weak criterion to determine an initial set of them (as in [Hol91]).

The principle is also *conceptually* difficult, referring to unbounded sequences of increases and decreases. This suggests studying a simpler problem as a stepping stone. By investigating the variable boundedness principle via a related but simpler principle for (ordinary) program termination, we are able to formulate an effective and efficient condition for identifying bounded variables. Along the way, we establish that the conditions of [Hol91] are complete for PSPACE.

1.2.3 Offline partial evaluation and program termination analysis

We will study a principle for program termination related to the principle for variable boundedness seen before, as a stepping stone to the analysis of variable boundedness. It is based on observing size descent in some parameter values. We call it the *size-change principle* for program termination.

> An infinite computation is *impossible*, if it would give rise to an infinite sequence of size decreases for some data values.

Dataflow analysis is used to collect a set of *size-relation graphs* for the program. These are bipartite graphs that describe the data size relations in possible program state transitions. Decidable conditions are formulated in terms of these graphs, based on the principle above. In this way, both analysis of program termination and variable boundedness can share abstract domains, supporting analyses, and conditions for detecting size descent. This work demonstrates a natural progression from one analysis to the next. Before turning our attention to other termination works, we mention a few *direct* applications of automatic termination analysis in the context of offline partial evaluation.

A finite unfolding strategy. In Similix [Bon91b], a call to a named function is marked static if it should be unfolded, and dynamic if it should be memoized. A strategy is generally needed to decide when *not* to unfold to avoid termination problems. As for unnamed functions, dynamic applications do not give rise to specializer states at all, while static applications are *always* unfolded. This constitutes quite an aggressive unfolding strategy.

Careless unfolding can lead to non-termination. This is a distinct problem from infinite function specialization, which can occur even when every possible function call has been marked for residualization. Conversely, with insufficient residualization, the specializer can loop on a *single* program state. There is a weak connection between the problems: Given a safe unfolding strategy, the boundedness of *static* variables is sufficient to guarantee finite function specialization.

It should not be surprising that automatic program termination analysis can be adapted to ensure the termination of unfolding, since the specializer performs ordinary program evaluation on static computations. The following is an application of the size-change principle to ensure finite unfolding.

> An infinite sequence of *static* function-call transitions for the specializer (causing infinite unfolding) is *impossible*, if it would give rise to an infinite sequence of size decreases for some *static* variable values.

More aggressive unfolding. This is an interesting application of termination deductions, because the analyses discussed so far all aim to achieve safety at the *expense* of specialization.

Generally, any symbolic computation (including unfolding, and rewriting `0 * ...` as `0`) must be careful to *retain* possibly non-terminating or erroneous computations. The termination of particular program expressions can therefore, in some instances, enable more aggressive transformation.

While our main motivation for studying the size-change principle is its connection to the principle for variable boundedness, a quick review of other termination works reveals that it is worthy of investigation in its own right.

1.2.4 Automatic termination analysis of TRS

An application of Term Rewriting Systems (TRSs) is to model the semantics of a functional program. A functional program can be translated into a TRS such that termination of the TRS implies termination of the subject program. (A TRS is *terminating* means that it is strongly normalizing. For TRSs that model call-by-value functional programs, it is more appropriate to consider innermost termination. Consult [Art97, AG97].)

The standard approach to prove the termination of a TRS is to search for a suitable ordering (specifically, one that is well-founded, and closed under context and substitution), in which the left-hand-side of each rewrite rule is greater than the right-hand-side. The orderings turned up by known techniques are all *simplification orderings* [Tou98]. TRSs that are proven terminating by such orderings are called *simply-terminating*. Consider the TRS below.

```
quot(0,n) → 0
quot(m,s(n)) → s(quot(minus(m,s(n)),s(n)))

minus(a,0) → a
minus(s(a),s(b)) → minus(a,b)
```

It is easily deduced to be *non-simply-terminating*, as the addition of the following rewrite rules:

```
minus(a,b) → a
minus(a,b) → b
```

results in a non-terminating TRS [Art97]. Intuitively, we see that the TRS *does* in fact terminate, because the result of rewriting `minus(m,s(n))` always has fewer `s` symbols than the first argument. However, simplification orderings do not subsume this reasoning.

It is not surprising then that most programs translate naively into non-simply-terminating TRSs. To develop methods sufficiently strong for them, Arts [Art97] applied programming intuition: "Dependency pairs" are used to capture the relations among arguments in a rewrite step. Descent is then sought for every possible legal cycle of dependency-pairs. The effect of this is to enable the application of localized descent reasoning, based on existing TRS methods.

In themselves, TRS methods tend to be powerful, as they appeal to fine-grained semantic modelling: *Polynomial interpretation* produces suitable orderings for proving termination by assigning the appropriate interpretation to each function symbol [DH93]. Any size deduction (for argument expressions) is implicit in this approach. Practically speaking, it is common to perform expensive searches for orderings to solve TRS inequalities.

The TRS approach to program termination is of course *completely* general. Orderings are sought that reflect common termination reasoning. In this work, we are interested in a *particular* (but quite general) way to reason about program termination, based on considering the data size relations in program state transitions. Practically, it leads to a natural factoring of concerns for establishing the termination of functional programs:

- analysis of argument size relations in possible program state transitions (using well established dataflow principles), and
- application of some criterion expressed in terms of the information above.

Our study will be geared towards deriving a criterion that applies the size-change principle effectively and efficiently. The satisfaction of a criterion induces a well-founded ordering on the program states in which each possible state transition is seen to decrease. We will assess the effectiveness of a criterion by considering the forms of descent, equivalently the orderings on program states, subsumed by it.

1.2.5 Automatic termination analysis of logic programs

There has been extensive research on the automatic termination analysis of logic programs. As explained in [Sag91], it is not always obvious that a predicate will terminate when executed with unusual instantiation patterns, or that a predicate always terminates on backtracking. Termination information can also be used to guide the choice of evaluation orders for certain compilers.

The termination analyzer of Mercury [SSS97] and Plümer's analysis [Plü90] are termination analyses for logic programs that use a simple termination criterion: For every recursive invocation of a predicate, determine that the sum over a subset of input fields (fixed for each predicate) is strictly decreased. Such a criterion does not subsume lexicographic descent. It also does not handle indirect recursion naturally. The strength of these termination analyses derives from their aggressive size analysis, which enables various sorting examples (e.g., quicksort and minsort) to be handled automatically. However, the precision of supporting analyses is *orthogonal* to the inherent power of the termination criterion.

Termilog [LS97b, LSS97] (extended from an analysis for Datalog programs [Sag91]) and Terminweb [CT97] are logic program analyzers whose termination conditions are as powerful as our *size-change termination criterion*, a concrete formulation of the size-change principle. In fact, the satisfaction of size-change termination can be checked by suitably encoding the problem for either analyzer. The first part of this thesis can therefore be seen as an in-depth study of the principle behind Termilog and Terminweb's analyses. Our investigations reveal their *intrinsic* hardness. We also hope, by exploring polynomial-time approximations of size-change termination, it will be possible to design an effective and efficient termination analysis for Prolog programs. However, such an analysis must account for argument instantiatedness, and an efficient scheme for this has yet to be devised.

1.2.6 Other termination works

There exist other schemes for automatic termination analysis, involving theorem proving, or indirect methods.

1. *Typed functional programs*: Abel and Altenkirch [AA99] have developed a system called `foetus` that accepts mutual recursive function definitions over strict positive datatypes as input. It returns a lexical ordering on the arguments of the program's functions, if one exists. However, theorem proving techniques are not appropriate for our purposes.

2. *An approach based on variable boundedness*: For first-order programs, Glenstrup [Gle99] has pointed out a straightforward application of variable boundedness analysis to termination. A simplified version: Add a "recursion depth" parameter to every function, which is incremented at every call. These parameters bound the program's recursion depth. If it is established that they are bounded in every execution, then the subject program is terminating. However, variable boundedness is *more difficult* to analyze than program termination.

3. *Adapting TRS methods*: Giesl [Gie95] has adapted automatic methods for TRS termination to functional programs. TALP [OCM00] is an adaptation of TRS methods for Prolog programs.

1.3 Aim of this work

This thesis will focus on the study of the size-change principle for program termination and the related principle for variable boundedness, that have been discussed in this introduction. The purpose is to develop convincing conditions for program termination and variable boundedness that can lead to effective and efficient analyses. It is *not* the aim of this work to present fully-fledged methods for either problem, but promising criteria will be suggested for implementation.

Of course, it cannot be denied that supporting analyses such as size analysis play a critical role in the successful application of the principles. Thus some effort is directed towards defining a range of prototype analyses that may be suitable for incorporation in an actual tool.

In particular, it is shown how to collect a safe set of *size-relation graphs* for the subject program. These are bipartite graphs that describe the data size relations in possible program state transitions. The size-change termination criterion is formulated in terms of these graphs, based on the size-change principle. It is proved to be complete for PSPACE. We therefore explore effective and efficient approximations. This approach is first applied to program termination, and then extended for variable boundedness. Specifically, we will:

1. Define a higher-order language L analogous to Scheme, and give its semantics. Treating a higher-order language will be necessary in practice, but presents formal difficulties.

2. Propose the use of size-relation graphs (a form of bipartite graphs) to capture the data size relations in potential program state transitions of the subject program. A collecting analysis for such graphs is proved to be conservative. Size-relation graphs support program termination and variable boundedness reasoning based on size descent observable in some data values.

3. Three criteria (sufficient conditions) are formulated for termination. These are expressed as properties of the subject program's size-relation graphs.

   For each criterion, we are interested in:

   - the motivation for its study,
   - any relation to existing termination works,
   - its intrinsic complexity,
   - its effectiveness in capturing "common descent," and
   - its power and efficiency in relation to the other size-change criteria.

4. Apply an analogous approach, extending the termination framework as necessary, to study the variable boundedness principle, and present a safe specialization strategy (with safe function specialization *and* safe unfolding) for the higher-order language L.

Our POPL article [LJBA01] is pre-cursor to this work. The size-change principle is investigated there with respect to a first-order subject language. In that work, a size-relation graph is attached to each function call, i.e., each *callsite*, to describe *every* possible program state transition arising at that callsite. In that context, the size-relation graph is a convenient vehicle to summarize valid argument size relations for each function call. Information about argument size relations is deduced separately by size analysis. This led to an intuitive presentation.

For the higher-order language L, the approach is more complicated. First, the semantics is designed to use an evaluation environment that records a value for *every* program variable, but only update the relevant ones at a program state transition. The redundant information makes it always possible to discuss the size relation of a value with respect to *any* variable (of course, it also limits what size relations can be expressed). The size-relation graph is used as an abstraction of function closures, to describe valid size relations among the arguments of a closure, and the values of variables in those environments in which the closure may occur. An extension of closure analysis derives a safe set of graphs for each program point. A set of graphs describing possible program state transitions for the program is then extracted from this information. All this is considerable work, but is necessary for good results when analyzing certain higher-order programs.

The size-relation graph is well-suited to model the program state transitions of first-order programs, but could be limiting for more complicated higher-order programs. Nonetheless, from our investigations, we believe that powerful descent reasoning is possible for non-trivial higher-order programs by the use of size-relation graphs. A better reason for using size-relation graphs to capture the information for termination deductions is that their simplicity facilitates the study of termination criteria.

While the first part of this work (on termination) was intended as a "straightforward" extension of the theory in [LJBA01] to higher-order subject programs, this has turned out to be more difficult, both formally and conceptually, than anticipated. It is important future work to regard the treatment of higher-order programs more closely. We reiterate that the main interest of this work is in showing how decidable principles for program termination and variable boundedness can be formulated in terms of information collected by abstract interpretation. These criteria can then be studied for their generality and complexity. It is owing to this perspective that we have been able to derive intrinsic hardness results for many analyses in the literature, and work towards effective and efficient approximations for them.

1.4 Structure of this work

This work is divided into two parts, on the analysis of program termination and variable boundedness respectively.

Part I consists of four chapters. Chapter 2 defines the higher-order subject language L, and presents its semantics. The following chapter defines and discusses the bipartite graphs used to capture data size relations in possible program state transitions. Prototype collecting analyses are defined to derive a safe set of graphs for any subject program. Chapter 4 presents various termination criteria, expressed as properties of these graphs. Some criteria are related to existing termination works. A PTIME criterion is proposed, with the hope of developing a practical analysis based on it that has the right balance of power and efficiency. It is surprisingly robust, and has a good time-complexity. The class of programs handled by each criterion is characterized by the types of descent subsumed by it.

Difficult complexity proofs are deferred to the following chapter. Complexity results are, as usual, achieved by reductions from known problems in the same complexity class. In this case, a PSPACE-hardness result is proved for the general size-change termination criterion by reduction from boolean program termination [Jon99]. The result is applicable to a number of analyses in the literature, including Lindenstrauss and Sagiv's Termilog [LSS97] and Codish and Taboch's Terminweb [CT97].

Part II (chapters 6–9) is structured similarly. Chapter 6 discusses existing works on variable boundedness, and presents the offline partial evaluation of L. It is shown how the termination analysis developed in Part I is useful for the design of a safe (and effective) unfolding strategy. Moreover, termination analysis allows some unused code (known to terminate) to be discarded during unfolding. This can lead to more compact and efficient residual programs. Such applications of Part 1's results are discussed as additional benefits for studying automatic analysis of program termination as a stepping stone to the analysis of variable boundedness.

Chapter 7 defines a structure combining the information about argument size relations derived in chapter 3, with parameter dependency information. (The parameter dependency information is needed to characterize *increases* in a static argument, for applying the variable boundedness principle.) An abstract interpretation is defined to collect such abstract descriptions for the potential state transitions of the specializer operating on the subject program. The following chapter presents criteria for variable boundedness in terms of the collected information. These are analogous to the termination criteria of Part 1. Among them is a criterion that coincides in power with existing analyses of variable boundedness in the literature [Hol91, JGS93, AH96, GJ96, Gle99]. We prove that it is complete for PSPACE. By analogy with the PTIME termination criterion, we propose a criterion for variable boundedness that can be decided in PTIME. It appears to achieve the right balance of power and efficiency, at least in the context that we have considered. In particular, we demonstrate that static variables are not unnecessarily generalized for a number of interpreters. These investigations yield a fully-fledged specializer for L, safe with respect to function specialization and unfolding.

The conclusion lists open problems, which include experimental and implementation work, and describes the contributions of this work. The ultimate goal of this research is to port the theory to Scheme programs, and develop a termination module for the Scheme partial evaluator Similix [Bon91b], so that offline partial evaluation of Scheme programs can be carried out *safely*, *effectively* and *efficiently*. We regard this work as a step towards that goal.

Chapter 2

Syntax and Semantics of Subject Language L

2.1 The subject language L

For the purpose of this study, we consider a language L similar to the side-effect-free fragment of Scheme treated by Similix [Bon91b]. Like Scheme [ADH+98], L is dynamically typed, i.e., no type checking is performed prior to program execution. The most notable difference between the two languages is that programs in L are expressed in a named-combinator form, as in Haskell [Tho96]. (Note though that evaluation in L will be eager, *unlike* in Haskell.) Function closures are obtained by *partial application*. While local closure definition via the lambda is not possible in L, standard *lambda lifting* [Joh85] maps any side-effect-free Scheme program to an equivalent L program.

In some ways, L is even more general than side-effect-free Scheme, because an application may or may not result in a function call depending on dynamic factors, whereas in Scheme, an application (barring arity problems and argument errors) always results in a transfer of control. The absence of local (lambda) variables simplifies the semantic presentation in this material.

2.2 Syntax of L programs

Programs of L are generated by the following grammar.

$$
\begin{array}{lll}
p \in \mathit{Prog} & ::== & d_1 \ \ldots \ d_M \\
d_i \in \mathit{Def} & ::== & f \ x_1 \ \ldots \ x_N = e^f \\
e_i, e^f \in \mathit{Expr} & ::== & x \mid f \mid (\texttt{if}\ e_1\ e_2\ e_3) \mid (op\ e_1, \ldots, e_n) \mid (e_1\ e_2) \\
f \in \mathit{Fun} & ::== & \text{Function identifier} \\
x, x_i \in \mathit{Var} & ::== & \text{Parameter identifier} \\
op \in \mathit{Op} & ::== & \text{Operator identifier}
\end{array}
$$

The definition of function f has the form $f\ x_1\ \ldots\ x_N = e^f$, where e^f is called the *body* of f. The number $N > 0$ of parameters is called the arity of f, written $ar(f)$. Denote the set of f parameters by $Param(f) = \{f^{(1)}, \ldots, f^{(N)}\}$. In examples, the $f^{(i)}$'s may be named by identifiers. For any expression e, the set of free variables of e is denoted $FV(e)$. Operators with arity 0 are known as *constants*. The *entry function* refers to the initial function of the program, denoted $f_{initial}$. By convention, `typed-text` denotes actual program text. Brackets are omitted when this does not cause confusion. (As usual, applications associate to the left.) For naturalness, some operators are written in-fix in examples. The following are syntactic assumptions about an L program that apply throughout this work. Their validity can be checked by inspecting the program text.

Syntactic assumptions

[A1] All function identifiers and parameter identifiers are distinct from one another.

[A2] No reference is made to $f_{initial}$ in the program.

[A3] All variables are *in scope* when used, thus any parameter referenced in e^f is from $Param(f)$.

Remark: Assumption [A2] is mainly for the benefit of program specialization, to ensure that a single specialized version of the entry function is generated having the *specified* binding-time division. The generated definition is then taken to be the entry function for the residual program. It will be seen that the development of our analyses is completely independent of [A2] (although towards the end of this work, [A2] is invoked for the boundedness of the entry function's parameters). In example programs satisfying [A2], we adopt the convention of naming the entry function `goal`.

2.3 Semantics of L programs

Before presenting the evaluation rules for L, some semantic assumptions about L are first stated.

Semantic assumptions

[A4] For a *big-step operational semantics* [Hen90], we equip L with *syntactic* categories of *base values*, *closure values* and *environments*.

$$\begin{array}{lcl} v, v_i \in \mathit{Val} & ::== & bv \mid cl \\ cl \in \mathit{Clo} & ::== & \langle f, v_1, \ldots, v_n \rangle \\ \sigma \in \mathit{Env} & ::== & \varepsilon \mid \sigma[x \mapsto v] \\ bv \in \mathit{Baseval} & & \\ f \in \mathit{Fun} & & \\ x \in \mathit{Var} & & \end{array}$$

The set of base values *Baseval* contains two distinguished elements: `Err` and `False`. To emphasize that *Baseval* is a denumerable set of syntactic items, its elements are represented using `typed text`. $f \in \mathit{Fun}$ is a subject program function, and $x \in \mathit{Var}$ is a subject program variable.

[A5] A program input is an $ar(f_{initial})$-tuple $\vec{v}$ such that each element $(\vec{v})_i \in \mathit{Baseval} \setminus \{\texttt{Err}\}$.

[A6] $\mathcal{O} : \mathit{Op} \to \mathit{Val}^* \rightharpoonup \mathit{Baseval}$ maps operator symbols to their "natural" interpretation. This interpretation is a *partial* function f such that both f and its domain are computable. By assumption, if `Err` or any *Clo* element are among the input values, the result is undefined.

Note that $\mathcal{O}$ interprets an operator symbol as a function of *Val* sequences. For example, $\mathcal{O}[\![+]\!]$ takes two *Val* numbers and returns a *Val* number; it is *not* the mathematical function for addition.

Definition 2.3.1 (Evaluation environment) The definition of *Env* is purely syntactic [NNH00], i.e., in a formal sense, σ is literally an element of the form $\varepsilon[x_1 \mapsto v_1] \ldots [x_n \mapsto v_n]$. For ease of development, we wish to treat σ like a mapping from *Var* to *Val*. To this end, we define the following.

1. For $\sigma = \varepsilon[x_1 \mapsto v_1] \ldots [x_n \mapsto v_n] \in \mathit{Env}$,

$$\sigma(x) = \begin{cases} v_i & \text{if } i \text{ is the largest index such that } x_i = x, \\ \texttt{Err} & \text{otherwise.} \end{cases}$$

2. The operator *env* constructs an environment from a function identifier and a list of *Val* elements: $\mathit{env}(f, v_1, \ldots, v_n) = \varepsilon[f^{(1)} \mapsto v_1] \ldots [f^{(n)} \mapsto v_n]$, provided $n = ar(f)$.

3. Let $\sigma \in \mathit{Env}$. Define $\sigma[f^{(i)} \mapsto v_i]_{i=1,n}$ to be $\sigma[f^{(1)} \mapsto v_1] \ldots [f^{(n)} \mapsto v_n]$.

The evaluation rules for L, given in Fig. 2.1, define the standard semantics for a call-by-value functional language with explicit handling of errors. The higher-order fragment has straightforward named-combinator semantics.

Definition 2.3.2 (Semantics of L)

1. An *evaluation state* has the form $\sigma; e$, where $\sigma \in \mathit{Env}$ and $e \in \mathit{Expr}$.

2. The state $\sigma; e$ *evaluates to* v, written $\sigma; e \Downarrow v$, means that $\sigma; e \Downarrow v$ is provable using the rules of Fig. 2.1. Note that $\Downarrow$ is dependent on the subject program p, so strictly it should be written $\Downarrow_p$.

The rule for primitive operations (rule (6)) is somewhat more eager than in a typical programming language, since computation usually aborts once the left-to-right evaluation of the actual arguments encounters an error. For L programs, an operation is terminating just when all of its arguments are. Thus, L programs are less terminating than might be expected. This leads to a termination analysis that is conservative for the standard interpretation of primitive operations. The use of rule (6) is to ease development.

$$\frac{v = \sigma(x)}{\sigma; x \Downarrow v} \quad (1) \qquad\qquad \frac{}{\sigma; f \Downarrow \langle f \rangle} \quad (2)$$

$$\frac{\sigma; e_1 \Downarrow \mathtt{Err}}{\sigma; (\mathtt{if}\ e_1\ e_2\ e_3) \Downarrow \mathtt{Err}} \quad (3) \qquad\qquad \frac{\sigma; e_1 \Downarrow \mathtt{False} \quad \sigma; e_3 \Downarrow v}{\sigma; (\mathtt{if}\ e_1\ e_2\ e_3) \Downarrow v} \quad (4)$$

$$\frac{\sigma; e_1 \Downarrow v_1 \quad v_1 \notin \{\mathtt{Err}, \mathtt{False}\} \quad \sigma; e_2 \Downarrow v}{\sigma; (\mathtt{if}\ e_1\ e_2\ e_3) \Downarrow v} \quad (5)$$

$$\frac{\sigma; e_1 \Downarrow v_1 \ \ldots\ \sigma; e_n \Downarrow v_n \quad \nexists v : v = \mathcal{O}[\![op]\!](v_1, \ldots, v_n)}{\sigma; (op\ e_1\ \ldots\ e_n) \Downarrow \mathtt{Err}} \quad (6)$$

$$\frac{\sigma; e_1 \Downarrow v_1 \ \ldots\ \sigma; e_n \Downarrow v_n \quad v = \mathcal{O}[\![op]\!](v_1, \ldots, v_n)}{\sigma; (op\ e_1\ \ldots\ e_n) \Downarrow v} \quad (7)$$

$$\frac{\sigma; e_1 \Downarrow v_1 \quad v_1 \notin \mathit{Clo}}{\sigma; (e_1\ e_2) \Downarrow \mathtt{Err}} \quad (8)$$

$$\frac{\sigma; e_1 \Downarrow \langle f, v_1, \ldots, v_n \rangle \quad \sigma; e_2 \Downarrow \mathtt{Err}}{\sigma; (e_1\ e_2) \Downarrow \mathtt{Err}} \quad (9)$$

$$\frac{\sigma; e_1 \Downarrow \langle f, v_1, \ldots, v_n \rangle \quad \sigma; e_2 \Downarrow v_{n+1} \quad v_{n+1} \neq \mathtt{Err} \quad n \neq ar(f) - 1}{\sigma; (e_1\ e_2) \Downarrow \langle f, v_1, \ldots, v_{n+1} \rangle} \quad (10)$$

$$\frac{\begin{array}{r}\sigma; e_1 \Downarrow \langle f, v_1, \ldots, v_n \rangle \quad \sigma; e_2 \Downarrow v_{n+1} \quad v_{n+1} \neq \mathtt{Err} \quad n = ar(f) - 1 \\ \sigma' = \sigma[f^{(i)} \mapsto v_i]_{i=1,ar(f)} \quad \sigma'; e^f \Downarrow v\end{array}}{\sigma; (e_1\ e_2) \Downarrow v} \quad (11)$$

Figure 2.1: Evaluation rules for L

An alternative for rule (11) for the function call is the following rule (11a).

$$\frac{\sigma; e_1 \Downarrow \langle f, v_1, \ldots, v_n\rangle \quad \sigma; e_2 \Downarrow v_{n+1} \quad v_{n+1} \neq \texttt{Err} \quad n = ar(f) - 1 \quad env(f, v_1, \ldots, v_{n+1}); e^f \Downarrow v}{\sigma; (e_1\ e_2) \Downarrow v}$$

Evaluation rule (11) and its alternative (11a) implement dynamic and static name binding for function calls respectively. Given syntactic assumption [A3], an assertion $\sigma; e \Downarrow v$ is provable with rules (1)–(11) just when it is provable with rules (1)–(10), (11a). Thus, both sets of rules implement equivalent semantics for the L programs under consideration. (*Aside*: If syntactic assumption [A3] is lifted, then rule (11) implements *fluid-binding* [ADH$^+$98], a facility whose undiscerning use quickly leads to obfuscated code.)

For designing a termination analysis based on the "size-change principle," we must model an L program's function-call state transitions. In this context, the use of rule (11) is to allow an evaluation environment to retain some argument values in a computation's history. In particular, it makes it possible to discuss size relation of a data value with respect to $\sigma(x)$– the last-seen value of x– for each $x \in Var$. Our termination analysis therefore uses rules (1)–(11) as an *instrumented version* of rules (1)–(10),(11a) (the natural semantics for L).

Lemma 2.3.3 (Unique computation) $\sigma; e \Downarrow v$ and $\sigma; e \Downarrow v'$ implies $v = v'$ and the proof tree of $\sigma; e \Downarrow v$ is unique.

Proof The proof is by induction on the size of the proof tree, with case analysis on the form of e at the root of the tree. For this first proof, we spell out all the tedious details. Let $\sigma; e \Downarrow v$ and $\sigma; e \Downarrow v'$.

- Suppose $e \equiv x$. Then $v = v' = \sigma(x)$. The proof tree is evidently unique.
- Suppose $e \equiv f$. Then $v = v' = \langle f\rangle$. The proof tree is evidently unique.
- Suppose $e \equiv (\texttt{if}\ e_1\ e_2\ e_3)$. One of evaluation rules (3)–(5) for `if` must apply since $\sigma; e$ evaluates to something. Now, $\sigma; e_1 \Downarrow v_1$ for some v_1, whichever rule applies. By the inductive hypothesis, v_1 is unique. Therefore, exactly *one* of rules (3)–(5) is applicable for the proofs of $\sigma; e \Downarrow v$ and $\sigma; e \Downarrow v'$, since these rules specify mutually exclusive conditions on v_1.
  - If the applicable rule is (3), $v = v' = \texttt{Err}$. Further, by the inductive hypothesis, the evaluation subtree for $\sigma; e_1 \Downarrow \texttt{Err}$ is unique, therefore so is the proof tree for $\sigma; e \Downarrow v$.
  - If the applicable rule is (4), v and v' are equal to the unique v'' such that $\sigma; e_3 \Downarrow v''$. The proof tree for $\sigma; e \Downarrow v$ is unique by uniqueness of the evaluation subtrees.
  - If the applicable rule is (5), v and v' are equal to the unique v'' such that $\sigma; e_2 \Downarrow v''$. The proof tree for $\sigma; e \Downarrow v$ is unique by uniqueness of the evaluation subtrees.
- Suppose $e \equiv (op\ e_1\ \ldots\ e_n)$. Then rule (6) or (7) has to apply. By the inductive hypothesis, each $\sigma; e_i \Downarrow v_i$, for some unique v_i. If $u = \mathcal{O}[\![op]\!](v_1, \ldots, v_n)$, then $v = v' = u$ by rule (6), and if there does not exist u such that $u = \mathcal{O}[\![op]\!](v_1, \ldots, v_n)$, then $v = v' = \texttt{Err}$ by rule (7). In each case, the proof tree is unique by uniqueness of the evaluation subtrees.
- Suppose $e \equiv (e_1\ e_2)$. One of the evaluation rules (8)–(11) must apply since $\sigma; e$ evaluates to something. Now, $\sigma; e_1 \Downarrow v1$, for some $v1$, whichever rule applies. By the induction hypothesis, $v1$ is unique.
  - If $v1 \notin Clo$, the applicable rule is (8), and $v = v' = \texttt{Err}$. Uniqueness of the proof tree follows from uniqueness of the evaluation subtree.

    So let $v1 = \langle f, v_1, \ldots, v_n\rangle$ for the remaining cases. Further, let $\sigma; e_2 \Downarrow v_{n+1}$. Such a v_{n+1} always exists since otherwise, $\sigma; e \Downarrow v$ would not be provable. By the induction hypothesis, v_{n+1} is unique.
  - If $v_{n+1} = \texttt{Err}$, the applicable rule is (9), and $v = v' = \texttt{Err}$. Uniqueness of the proof tree follows from uniqueness of the evaluation subtrees.

- If $v_{n+1} \neq \texttt{Err}$, the application $(e_1\ e_2)$ is legal. If $n \neq ar(f) - 1$, the applicable rule is (10), and $v = v' = \langle f, v_1, \ldots, v_{n+1} \rangle$. The evaluation tree is unique by uniqueness of the evaluation subtrees.
- Finally, if $v_{n+1} \neq \texttt{Err}$ and $n = ar(f) - 1$, the application $(e_1\ e_2)$ is a function call, and the applicable rule is (11). Let $\sigma_1 = \sigma[f^{(i)} \mapsto v_i]_{i=1,n+1}$. Then $\sigma_1; e^f \Downarrow v''$ for some v''. By the induction hypothesis, v'' is unique. Therefore $v = v' = v''$. The evaluation tree is unique by uniqueness of the evaluation subtrees.

Definition 2.3.4 (Termination of evaluation) Evaluation of $\sigma; e$ is *terminating* if there exists v such that $\sigma; e \Downarrow v$, *non-terminating* otherwise.

Definition 2.3.5 (Behaviour of an L program p)

1. Given p and its input $\vec{v} = (v_1, \ldots, v_N)$, if $\sigma_0; e^{f_{initial}} \Downarrow u$ where $\sigma_0 = env(f_{initial}, v_1, \ldots, v_N)$, then p is said to *evaluate to u on $\vec{v}$*. Otherwise, p is *undefined on $\vec{v}$*.
2. σ_0 is called an *initial environment*, and $\sigma_0; e^{f_{initial}}$ is called an *initial state*.
3. p is *terminating* if for all inputs $\vec{v}$, p evaluates to u for some u. Equivalently, p is terminating if every initial state is terminating.

Remark: By semantic assumption [A5] and the definition of *env*, any initial environment σ_0 satisfies:

- $\sigma_0(x) \in Baseval \setminus \{\texttt{Err}\}$ for $x \in Param(f_{initial})$;
- $\sigma_0(x) = \texttt{Err}$ otherwise.

2.4 Infinite computations

The use of (finite) proof trees to capture terminating computations is apt, but does not allow anything to be expressed regarding evaluation states in an infinite computation. However, the evaluation rules have been designed to admit deterministic backward inferencing, using placeholders for unknown values, and assigning these placeholders by unification when an inference-tree leaf is discharged (see [Jon99] for a formal treatment).

One may therefore *conceive* of an infinite computation tree, possibly containing un-unified placeholders, as representing an infinite computation. Such a computation tree may be a proof tree; a finite tree with undischarged leaves (representing a computation that has become "stuck"), or an infinite tree. The evaluation rules for L have been designed so that a computation never becomes stuck. Thus, a computation tree is either a finite proof tree, or it is infinite.

While the infinite computation tree is a valuable aid in visualizing a divergent computation, it would be tedious and uninstructive to define it in full, especially when such a tree would contain, in addition to the evaluation states of interest, irrelevant information such as un-unified placeholders. A leads-to relation $\leadsto$ is defined instead, which formally captures the parent–child evaluation-state relation in the imagined finite or infinite computation tree. A chain in $\leadsto$ represents a branch in this tree. Any assertion about an infinite computation is expressed via the $\leadsto$ relation. In particular, any non-terminating computation implies the existence of an infinite $\leadsto$ chain, so by proving that any infinite $\leadsto$ chain is impossible (due to infinite descent), it follows that the subject program terminates on all inputs. (Note that the $\leadsto$ relation is in general uncomputable. It is simply a *device* to formalize arguments about the evaluation states in an infinite computation.)

Definition 2.4.1 ($\leadsto$ relation) Suppose $\sigma; e \in Env \times Expr$ is given. Then $\sigma; e \leadsto \sigma'; e'$ signifies that the evaluation of $\sigma'; e'$ is *a subcomputation of* $\sigma; e$. Specifically,

- If $e \equiv (\texttt{if}\ e_1\ e_2\ e_3)$, and $\sigma; e_1 \Downarrow \texttt{False}$, then $\sigma; e \leadsto \sigma; e_1$ and $\sigma; e \leadsto \sigma; e_3$.

- If $e \equiv (\texttt{if}\ e_1\ e_2\ e_3)$, and $\sigma; e_1 \Downarrow v_1$, with $v_1 \notin \{\texttt{Err}, \texttt{False}\}$, then $\sigma; e \leadsto \sigma; e_1$ and $\sigma; e \leadsto \sigma; e_2$.
- If $e \equiv (\texttt{if}\ e_1\ e_2\ e_3)$, but the previous cases do not apply, then $\sigma; e \leadsto \sigma; e_1$.
- If $e \equiv (op\ e_1\ \ldots\ e_n)$, then for each $i = 1, \ldots, n$, $\sigma; e \leadsto \sigma; e_i$.
- If $e \equiv (e_1\ e_2)$ and $\sigma; e_1 \Downarrow \langle f, v_1 \ldots, v_n \rangle$ where $n = ar(f) - 1$ and $\sigma; e_2 \Downarrow v_{n+1}$ such that $v_{n+1} \neq \texttt{Err}$, then $\sigma; e \leadsto \sigma; e_1$, $\sigma; e \leadsto \sigma; e_2$, and $\sigma; e \leadsto \sigma'; e^f$, where $\sigma' = \sigma[f^{(i)} \mapsto v_i]_{i=1,n+1}$. Call $\sigma; e \leadsto \sigma'; e^f$ a *function-call transition to f*.
- If $e \equiv (e_1\ e_2)$, $\sigma; e_1 \Downarrow \langle f, v_1 \ldots, v_n \rangle$, but the previous case does not apply, then $\sigma; e \leadsto \sigma; e_1$ and $\sigma; e \leadsto \sigma; e_2$.
- If $e \equiv (e_1\ e_2)$, but the previous cases do not apply, then $\sigma; e \leadsto \sigma; e_1$.
- If $e \equiv x$ or $e \equiv f$, then $\sigma; e$ has no subcomputations.

Let $\leadsto^+$ and $\leadsto^*$ denote the transitive closure and reflexive transitive closure of $\leadsto$ respectively.

Observation: If $\sigma; e \Downarrow v$, then the subcomputations of $\sigma; e$ are the states at the roots of subtrees in the proof of $\sigma; e \Downarrow v$. We frequently use the term "subcomputation" in this sense.

Definition 2.4.2 (Reachability)

1. A $\leadsto$ *chain* is a finite or infinite sequence: $\sigma_1; e_1,\ \sigma_2; e_2,\ \ldots,\ \sigma_t; e_t,\ \ldots$ such that for each t: $\sigma_t; e_t \leadsto \sigma_{t+1}; e_{t+1}$. We also write $\sigma_1; e_1 \leadsto \sigma_2; e_2 \leadsto \ldots$.
2. $\sigma; e$ is *reachable from* $\sigma_0; e_0$ if $\sigma_0; e_0 \leadsto^* \sigma; e$. An evaluation state is *reachable* if it is reachable from an initial state.

Observation: If $\sigma; e$ is reachable and $\sigma; e \Downarrow v$, then the states at the roots of subtrees in the proof of $\sigma; e \Downarrow v$, i.e., the subcomputations, are reachable. We will regard this fact as evident.

Theorem 2.4.3 The following are equivalent.

1. $\sigma; e$ is terminating.
2. $\sigma; e \leadsto \sigma'; e' \Rightarrow \sigma'; e'$ is terminating.
3. $\leadsto$ has no infinite chain starting with $\sigma; e$.

Proof
(1) $\Leftrightarrow$ (3): For the forward implication, we prove, by induction on the length of a $\leadsto$ chain with initial state $\sigma; e$, that any such chain corresponds to a branch in the proof tree of $\sigma; e \Downarrow v$. An infinite chain then implies the existence of an arbitrarily long branch in the tree, which is impossible.

The single-state chain $\sigma; e$ corresponds to the root node of the proof tree of $\sigma; e \Downarrow v$. Consider chain $\sigma; e \leadsto \ldots \leadsto \sigma'; e' \leadsto \sigma''; e''$, where $\sigma; e \leadsto \ldots \leadsto \sigma'; e'$ corresponds to a branch of the proof tree of $\sigma; e \Downarrow v$. By case analysis, $\sigma'; e' \leadsto \sigma''; e''$ only if $\sigma''; e'' \Downarrow v''$ (for some v'') is *necessary* for the proof of $\sigma'; e' \Downarrow v'$. Thus $\sigma; e \leadsto \ldots \leadsto \sigma''; e''$ is also a branch in the proof tree.

For the (more important) reverse implication, observe that if $\sigma'; e'$ is non-terminating, there must exist $\sigma''; e''$, where $\sigma'; e' \leadsto \sigma''; e''$ such that $\sigma''; e''$ is also non-terminating: The cases in the definition of $\leadsto$ are exhaustive, and in each case, the termination of all subcomputations implies termination of the present computation. Starting with $\sigma; e$, which is non-terminating by assumption, and applying the observation inductively, the existence of an infinite $\leadsto$ chain with initial state $\sigma; e$ is deduced.

(1) $\Leftrightarrow$ (2): This justifies referring to the set of $\sigma'; e'$ such that $\sigma; e \leadsto \sigma'; e'$ as the subcomputations of $\sigma; e$, and is an easy corollary of (1) $\Leftrightarrow$ (3).

Corollary 2.4.4 A program p is terminating iff there are no infinite $\leadsto$ chains starting with an initial state.

Proof is immediate from Theorem 2.4.3 and Definition 2.3.5.

2.5 CTI: An induction principle

In most respects, the evaluation rules given for L are natural and intuitive. The most problematic aspect of the semantic formulation by far is the inconvenience of making assertions regarding values captured by closures. A specialized "computation-tree induction principle" (CTI) is now formulated to facilitate the proof of correctness for abstract interpretation over L. (In the next chapter, we explain our reasons for developing abstract interpreters "from scratch.") The CTI attempts to capture the intuition behind the informal justification of abstract analyses.

The principle allows for a property of the form $P(v, \sigma, e)$, where $v \in \mathit{Val}$ is a value associated with the evaluation state $\sigma; e$. For a closure value of the form $\langle f, v_1, \ldots, v_n\rangle$, we want to associate actual argument v_i with formal parameter $f^{(i)}$, so we will define P^*, extending P to the arguments captured in a closure value. Intuitively, $P^*(v, \sigma, e)$ expresses that P holds for v and evaluation state $\sigma; e$, *and* if $v = \langle f, v_1, \ldots, v_n\rangle$, then for $i = 1, \ldots, n$, P^* holds for v_i and evaluation state $\sigma; f^{(i)}$. This allows inferences about captured arguments to be tagged to the relevant parameters. The formal definition of P^* from P is given below.

Definition 2.5.1 (Embedded values in a closure, extending predicate P to P^*)

1. For $v' \in \mathit{Clo}$, v is *embedded at* $f^{(k)}$ *in* v' means
   - $v' = \langle f, v_1, \ldots, v_n\rangle$ and $v = v_k$; or
   - $v' = \langle f', v'_1, \ldots, v'_m\rangle$ and for some j, v is embedded at $f^{(k)}$ in v'_j.
2. Given predicate $P(v', \sigma, e)$ over $\mathit{Val} \times \mathit{Env} \times \mathit{Expr}$, define the predicate $P^*(v', \sigma, e)$ to hold just when $P(v', \sigma, e)$ holds, *and* for any v embedded at $f^{(k)}$ in v', $P(v, \sigma, f^{(k)})$ holds.

Theorem 2.5.2 (CTI) Let $P(v, \sigma, e)$ be any predicate over $\mathit{Val} \times \mathit{Env} \times \mathit{Expr}$. And suppose that the following are true.

- [B1] For $v \in \mathit{Baseval} \setminus \{\texttt{Err}\}$, σ_0 an initial environment and $x \in \mathit{Param}(f_{\mathit{initial}})$, $P(v, \sigma_0, x)$ holds.

  The intuition: P associates any non `Err` base value to an f_{initial} parameter, with respect to any initial environment. This specifies a "base case" for CTI.

- [B2] For $f \in \mathit{Fun}$ and any σ, $P(\langle f\rangle, \sigma, f)$ holds.

  Since an expression of the form f evaluates to $\langle f\rangle$, P associates this closure value to the expression f in any environment.

- [I1] For $\sigma; (e_1\ e_2)$ reachable, $\sigma; e_1 \Downarrow \mathit{cl}$ where $\mathit{cl} = \langle f, v_1, \ldots, v_n\rangle$ for some $n < \mathit{ar}(f)$, and $\sigma; e_2 \Downarrow v_{n+1}$ where $v_{n+1} \neq \texttt{Err}$, if both $P^*(\mathit{cl}, \sigma, e_1)$ and $P^*(v_{n+1}, \sigma, e_2)$ hold, then $P^*(v_{n+1}, \sigma, f^{(n+1)})$ holds.

  In words, for any application $(e_1\ e_2)$, if during execution e_1 evaluates to a closure with n arguments, and P^* associates the result of evaluating e_i with the corresponding subcomputation state $\sigma; e_i$, then P^* associates the result of evaluating e_2 to the parameter $f^{(n+1)}$. *Point*: The result of evaluating e_2 is tagged to the parameter $f^{(n+1)}$, since if $(e_1\ e_2)$ gives rise to a function-call transition, or evaluates to a closure that is eventually completed, then the result of evaluating e_2 will be assigned to $f^{(n+1)}$.

- [I2] For $\sigma; e$ reachable, where e is a conditional, operation or application, and $\sigma; e \Downarrow v$, if for any subcomputation $\sigma'; e'$ with $\sigma'; e' \Downarrow v'$, it holds that $P^*(v', \sigma', e')$, then $P^*(v, \sigma, e)$ holds.

  In words, if P^* associates the result of any subcomputation to the subcomputation state, then it associates the result of the current computation to the current evaluation state.

- [I3] For $\sigma; e$ reachable, $\sigma; e \rightsquigarrow \sigma'; e^f$ and $x \in FV(e^f)$: $P^*(\sigma'(x), \sigma, x) \Rightarrow P^*(\sigma'(x), \sigma', x)$.

  During a function call, the environment is updated, but the validity of P^* is not affected. Inductive cases [I1] and [I3] allow P^* to "carry forward" from the current evaluation environment to the evaluation environment of any subcomputation.

The conclusions are:

1. Let $\sigma; e$ be reachable. If $\sigma; e \Downarrow v$, and for each $x \in FV(e)$, $P^*(\sigma(x), \sigma, x)$ holds, then $P^*(v, \sigma, e)$ holds. Informally: if the evaluation environment is anticipated by P^*, then so is the result.

2. Let $\sigma; e$ be reachable. Then $P^*(\sigma(x), \sigma, x)$ holds for all $x \in FV(e)$. Informally, P^* associates any possible value of a parameter with that parameter.

3. Let $\sigma; e$ be reachable. If $\sigma; e \Downarrow v$, then $P^*(v, \sigma, e)$ holds. Informally, P^* associates the result of any possible computation with the evaluation state.

Proof

The first claim is an assertion about finite computations, and is proved by induction on the size of the proof tree for $\sigma; e \Downarrow v$. We perform case analysis on the form of e and the final proof rule at the root of the tree. Suppose $\sigma; e$ is reachable.

- Let $e \equiv x$. By the premise, $P^*(\sigma(x), \sigma, x)$ holds. Since $v = \sigma(x)$, $P^*(v, \sigma, e)$ holds.
- Let $e \equiv f$. Now, $v = \langle f \rangle$, so by [B2], $P^*(v, \sigma, e)$ holds.
- Let $e \equiv (\texttt{if}\ e_1\ e_2\ e_3)$. For any subcomputation $\sigma'; e' \Downarrow v'$, we know $\sigma' = \sigma$, so by the inductive hypothesis, $P^*(v', \sigma', e')$ holds. By [I2], $P^*(v, \sigma, e)$ holds.
- Let $e \equiv (op\ e_1\ \ldots\ e_n)$. Argue as above.
- Let $e \equiv (e_1\ e_2)$. If the applicable proof rule is (8), (9) or (10), argue as before. Suppose it is (11). Then $\sigma; e_1 \Downarrow cl$ where $cl = \langle f, v_1, \ldots, v_n \rangle$ for $n = ar(f) - 1$, and $\sigma; e_2 \Downarrow v_{n+1}$ where $v_{n+1} \neq \texttt{Err}$. As before, argue that $P^*(cl, \sigma, e_1)$ and $P^*(v_{n+1}, \sigma, e_2)$ hold. By definition of P^*, we have that $P^*(v_i, \sigma, f^{(i)})$ holds for $i = 1, \ldots, n$. By [I1], $P^*(v_{n+1}, \sigma, f^{(n+1)})$ holds. Thus, for the updated environment $\sigma' = \sigma[f^{(i)} \mapsto v_i]_{i=1,n+1}$ and $x \in Param(f)$, we have that $P^*(\sigma'(x), \sigma, x)$ holds. It follows from [I3] that for $x \in FV(e^f)$, we have that $P^*(\sigma'(x), \sigma', x)$ holds. Let $\sigma'; e^f \Downarrow v$. By the inductive hypothesis, $P^*(v, \sigma', e^f)$ holds. Since every subcomputation $\sigma'; e' \Downarrow v'$ is such that $P^*(v', \sigma', e')$ holds, it follows from [I2] that $P^*(v, \sigma, e)$ holds.

The second claim is proved by induction on the length of a $\rightsquigarrow$ chain with initial state $\sigma_0; e^{f_{initial}}$. For the initial state, use the definition of initial environments and condition [B1] to conclude that $P^*(\sigma_0(x), \sigma_0, x)$ holds for each $x \in FV(e^{f_{initial}}) \subseteq Param(f_{initial})$.

Now consider a chain of the form $\sigma_0; e^{f_{initial}} \rightsquigarrow^* \sigma_t; e_t \rightsquigarrow \sigma_{t+1}; e_{t+1}$ where $P^*(\sigma_t(x), \sigma_t, x)$ holds for each $x \in FV(e_t)$. If the transition is *not* a function-call transition, then $\sigma_{t+1} = \sigma_t$, and we conclude that $P^*(\sigma_{t+1}(x), \sigma_{t+1}, x)$ holds for each $x \in FV(e_{t+1}) \subseteq FV(e_t)$. Otherwise, the following are true:

- e_t has the form $(e'\ e'')$;
- the applicable rule for $\sigma_t; e_t$ is (11);
- $\sigma_t; e' \Downarrow cl$ where $cl = \langle f, v_1, \ldots, v_n \rangle$ for $n = ar(f) - 1$, and $\sigma_t; e'' \Downarrow v_{n+1}$ where $v_{n+1} \neq \texttt{Err}$;
- $\sigma_{t+1} = \sigma_t[f^{(i)} \mapsto v_i]_{i=1,n+1}$; and

- $e_{t+1} = e^f$.

By the first claim and the inductive assumption, $P^*(cl, \sigma_t, e')$ and $P^*(v_{n+1}, \sigma_t, e'')$ hold. By definition of P^*, we have that $P^*(v_i, \sigma_t, f^{(i)})$ holds for $i = 1, \ldots, n$, and by [I1], $P^*(v_{n+1}, \sigma_t, f^{(n+1)})$ holds. Thus, for each $x \in FV(e^f) \subseteq Param(f)$, $P^*(\sigma_{t+1}(x), \sigma_t, x)$ holds. $P^*(\sigma_{t+1}(x), \sigma_{t+1}, x)$ follows from [I3]. The principle of mathematical induction then gives: For every reachable $\sigma; e$ and $x \in FV(e)$, we have that $P^*(\sigma(x), \sigma, x)$ holds.

The final claim follows immediately from the first two claims.

CTI is usually used in specialized ways. If argument σ is absent from the predicate P, then [I3] is trivially satisfied. If both σ and e are absent from P, then CTI makes assertions about the values occurring in arbitrary computation trees.

- [B1] becomes: $P(v)$ holds for all $v \in Baseval \setminus \{\texttt{Err}\}$.
- [B2] becomes: $P(\langle f \rangle)$ holds for all $f \in Fun$.
- [I2] becomes: For $\sigma; e$ reachable, where e is a conditional, operation or application, and $\sigma; e \Downarrow v$, if for any subcomputation $\sigma'; e'$, with $\sigma'; e' \Downarrow v'$, we have that $P^*(v')$ holds, then $P^*(v)$ holds.
- [I1] and [I3] are trivially satisfied.

The simplified CTI can be used to prove, for instance, that `Err` is never captured by a closure. The following lemma shows that the number of arguments captured by a closure never reaches or exceeds the arity of the function.

Lemma 2.5.3 For any reachable $\sigma; e$ and $x \in FV(e)$, if $\sigma(x) = \langle f, v_1, \ldots, v_n \rangle$, then $n < ar(f)$. And if $\sigma; e \Downarrow \langle f, v_1, \ldots, v_n \rangle$, then $n < ar(f)$.

Proof Let $P(v)$ be $(v = \langle f, v_1, \ldots, v_n \rangle \Rightarrow n < ar(f))$. Conditions [B1] and [B2] are trivial. Condition [I2] is also straightforward except when the proof rule is (9). So let $\sigma; e_1 \Downarrow cl$ where $cl = \langle f, v_1, \ldots, v_n \rangle$, and $\sigma; e_2 \Downarrow v_{n+1}$. Then $\sigma; (e_1\ e_2) \Downarrow v$ where $v = \langle f, v_1, \ldots, v_{n+1} \rangle$. By hypothesis $P^*(cl)$, it holds that $n < ar(f)$. A premise of rule (9) specifies that $n \neq ar(f) - 1$, thus $n < ar(f) - 1$, i.e., $n + 1 < ar(f)$, so $P(v)$ holds. Any value v' embedded at $f'^{(k)}$ in v is either one of $v_1, \ldots, v_{n+1}$, or embedded at $f'^{(k)}$ in v_j for $0 < j \leq n + 1$; therefore $P(v')$ holds by the hypotheses $P^*(cl)$ and $P^*(v_{n+1})$. This allows us to conclude that $P^*(v)$ holds. The lemma follows by CTI.

2.6 Sizes of data values

A size function (also called a *norm*) mapping data values to natural numbers, is central to termination analysis. Although it has been suggested that the synthesis of suitable norms remains an obstacle to truly automatic termination analysis [DDF93], practically speaking, only a handful of common ones are needed.

For this study, we want to define a simple, but realistic termination analysis. For a concrete *Baseval*, we assume a set *Atom*, from which we define the usual recursive *list structures*.

Definition 2.6.1 (*Baseval*)

1. *Atom* is a denumerable set containing the distinguished element `False`, but *not* containing `Err`.
2. *Base values* are defined from *Atom*.

$$\begin{array}{lcl} bv, bv_i \in Baseval & ::== & atom \mid [bv_1, \ldots, bv_q] \mid \texttt{Err} \\ atom \in Atom & & \end{array}$$

3. We define the standard operators for lists.

$$
\begin{array}{lcl}
\mathcal{O}[\![\texttt{nil}]\!]() & = & [] \\
\mathcal{O}[\![\texttt{:}]\!](bv, [bv_1, \ldots, bv_q]) & = & [bv, bv_1, \ldots, bv_q] \\
\mathcal{O}[\![\texttt{hd}]\!]([b_1, \ldots, b_q]) & = & b_1 \quad (q \geq 1) \\
\mathcal{O}[\![\texttt{tl}]\!]([b_1, \ldots, b_q]) & = & [b_2, \ldots, b_q] \quad (q \geq 1)
\end{array}
$$

The list constructor `:` is usually written in-fix, and 0-ary operators like `nil` are written without brackets. For example, `1 : 2 : 3 : nil` is syntactic sugar for `(: 1 (: 2 (: 3 (nil))))`. For readability, such an expression is also written `[1 2 3]` in program text. It is not to be confused with the corresponding *value* `[1,2,3]`, an element of *Baseval*.

4. To ease development, define the size function # to count the number of list constructors in base values (for now).

$$\# : \mathit{Val} \to \mathbb{N}$$

$$\#[bv_1, \ldots, bv_q] = q + \Sigma_{i=1}^{q} \# bv_i$$
$$\#v = 0 \text{ for } v \text{ not of the form } [bv_1, \ldots, bv_q]$$

The size function # is a frequently applicable one in the context of termination analysis, as attested by the Mercury experiments [SSS97]. With this size function, `hd` and `tl` operations always evaluate to a value with smaller size than the input, when the result of the evaluation is not `Err`.

For convenience, a non-negative integer n is treated like a list of length n for size analysis purposes. Thus, subtracting 1 from an expression known to evaluate to a positive integer results in a value with strictly smaller size. The size function # assigns a size of 0 to all closures, i.e., $\#v = 0$ for $v \in \mathit{Clo}$, although we will discuss an alternative size function for closure values to handle decreasing *continuations* in the final chapter. (This is not difficult, but it distracts from the central development.)

Note that the size function would not be very useful for characterizing size-decreasing operations if taking the head or tail of the empty list resulted in the empty list, as in older (now non-standard) implementations of Scheme [TI 87]. In this case, successful head or tail operations do not always result in a value with smaller size than the input. In the current standard for Scheme [ADH+98], taking the head or tail of the empty list is defined as erroneous.

Chapter 3

Abstract Interpretation of Argument Size Relations

3.1 Abstract interpretation

Abstract interpretation is a methodology for *modelling* and *approximating* dynamic (i.e., runtime) program properties. Operationally, the analysis performed on the subject program may be regarded as a form of interpretation over an abstract domain. Abstract interpretation was formulated and studied in depth by Patrick and Radhia Cousot [CC77, CC79]. Combined with the appropriate *instrumented semantics*, it was used to explain, and prove correct traditional dataflow analyses, such as those described in the "compiler book" [ASU86] (although the use of *path semantics*, e.g., Hoare's trace semantics [Hoa78], makes correctness proofs considerably tedious). A good introduction to abstract interpretation can be found in Patrick Cousot's article in *Program Analysis* [Cou81]. We sketch some central concepts below.

1. The set of runtime program properties, where a property is a subset of possible program states (for some formulation of program states), forms a complete lattice $\mathcal{C} = (C, \leq_C, \perp_C, \top_C, \vee_C, \wedge_C)$, with $c_1 \leq_C c_2$ if c_2 is weaker (more general) than c_1. $\mathcal{C}$ is referred to as the *concrete domain*.

2. A subset of $\mathcal{C}$ closed under $\wedge_C$, and containing the properties of interest is a suitable *abstract domain*. (A property of interest might be: variable `x` has an even integer value. Formally, this is the set of program states in which `x` has even integer values.)

   An abstract domain can be regarded as the range of an *upper closure operator (uco)* [CFG+97]. A uco formalizes the abstraction process. It is a map ρ on $\mathcal{C}$ that is monotonic: $x \leq_C y$ implies that $\rho(x) \leq_C \rho(y)$; extensive: $x \leq_C \rho(x)$; and idempotent: $\rho(\rho(x)) = \rho(x)$. We can then define the abstract domain as the lattice $\mathcal{A} = (A, \leq_A, \perp_A, \top_A, \vee_A, \wedge_A)$, where

$$\begin{array}{lcl} A & = & \{\rho(c) \mid c \in C\} \\ \leq_A & = & \{(a_1, a_2) \in \leq_C \mid a_1, a_2 \in A\} \\ \perp_A & = & \rho(\perp_C) \\ \top_A & = & \top_C \text{ (which is in } A) \\ \bigvee_A\{\rho(x) \mid x \in X\} & = & \rho(\bigvee_C\{\rho(x) \mid x \in X\}) \\ \bigwedge_A\{\rho(x) \mid x \in X\} & = & \bigwedge_C\{\rho(x) \mid x \in X\} \text{ (which is in } A) \end{array}$$

   Patrick Cousot [Cou81] also argues that it makes sense to use a complete sub-lattice of $\mathcal{C}$ as the abstract domain, at least at design time.

3. Typically, an isomorphic copy of the lattice $\mathcal{A}$ above is used. The concrete and abstract domains are then connected by a *Galois insertion*. This is a pair of monotonic functions α, γ such that α is onto, γ is one-to-one, and for every $c \in C$ and $a \in A$,

$$\begin{array}{lcl} c & \leq_C & \gamma(\alpha(c)), \text{ and} \\ \alpha(\gamma(a)) & = & a. \end{array}$$

   Functions α, γ are referred to as the abstraction and concretization map respectively. Intuitively, for each $c \in C$, $\alpha(c)$ is its representative (approximation) in A, and for $a \in A$, $\gamma(a)$ assigns a its "meaning." The first relation between α and γ above expresses the following intuition: If we work with an abstract value representing c, we are working with an assertion no stronger than c. The second relation ensures that multiple $a \in A$ do not approximate the same assertion in C. Only one of α, γ needs to be defined, since each can computed from the other as follows.

$$\begin{array}{lcl} \gamma(a) & = & \bigvee_C\{c \mid \alpha(c) \leq_A a\} \\ \alpha(c) & = & \bigwedge_A\{a \mid c \leq_C \gamma(a)\} \end{array}$$

   A standard approach to abstract interpretation (approximating the effect of a conditional by the join of the effects of each branch, approximating *op* as $\alpha \circ \mathcal{O}[\![op]\!] \circ \gamma$, etc.) yields a set of fixedpoint equations whose solution *approximates* the meet-over-all-paths solution of the dataflow problem [Cou81].

Domains for closure analysis. Let $AClo = \{\langle\!\langle f, n \rangle\!\rangle \mid f \in Fun, n < ar(f)\}$. Then $\mathcal{A} = \wp(AClo)$ and $\mathcal{C} = \wp(Val)$ are suitable abstract and concrete domains for the values of *each* variable. The Galois insertion relating them is:

$$\begin{array}{rcl} \alpha(c) & = & \{\langle\!\langle f, n \rangle\!\rangle \mid \langle f, v_1, \ldots, v_n \rangle \in c\} \\ \gamma(a) & = & Baseval \cup \{\langle f, v_1, \ldots, v_n \rangle \mid \langle\!\langle f, n \rangle\!\rangle \in a \text{ and } v_1, \ldots, v_n \in Val\} \end{array}$$

Closure analysis operator. A closure analysis is given by operator $\mathcal{C}l : Expr \to \mathcal{A}$.

$$\begin{array}{lcl} \mathcal{C}l[\![f^{(n)}]\!] & = & \{\varrho \mid (e_1\ e_2) \text{ in program}, \ \langle\!\langle f, n-1 \rangle\!\rangle \in \mathcal{C}l[\![e_1]\!], \ \varrho \in \mathcal{C}l[\![e_2]\!]\} \\ \mathcal{C}l[\![f]\!] & = & \{\langle\!\langle f, 0 \rangle\!\rangle\} \\ \mathcal{C}l[\![(\texttt{if}\ e_1\ e_2\ e_3)]\!] & = & \mathcal{C}l[\![e_2]\!] \cup \mathcal{C}l[\![e_3]\!] \\ \mathcal{C}l[\![(op\ e_1\ \ldots\ e_n)]\!] & = & \{\} \\ \mathcal{C}l[\![(e_1\ e_2)]\!] & = & \{\langle\!\langle f, n+1 \rangle\!\rangle \mid \langle\!\langle f, n \rangle\!\rangle \in \mathcal{C}l[\![e_1]\!], \ n < ar(f) - 1\} \cup \\ & & \{\varrho \mid \langle\!\langle f, ar(f)-1 \rangle\!\rangle \in \mathcal{C}l[\![e_1]\!], \ \varrho \in \mathcal{C}l[\![e^f]\!]\} \end{array}$$

Figure 3.1: Simple closure analysis $\mathcal{C}l$

4. If $\mathcal{A}$ does not have finite height, approximation should be applied to the equations modelling the effect of a loop (or a recursion), so that the system of fixedpoint equations is solvable in finite time.

Some recent works on abstract interpretation have discussed issues such as combining abstract domains [CMB+95, CC95], or complementing them [CFG+97], as part of the effort towards more systematic and modular abstract interpretation. However, the definition of static analyses for unfamiliar program properties still presents challenges. An example is the problem discussed in this work. It involves modelling the size relations among a runtime closure's arguments, and parameter values in those environments in which the closure may occur. CTI allows us to prove the correctness of this and other analyses, which is needed for a convincing account when the properties of interest are not convenient *state-based* properties.

3.2 Simple closure analysis

Closure analysis frequently features as part of more complicated analyses, as it allows first-order methods to be extended for higher-order programs. A closure can be represented abstractly as a function name, together with a number indicating the closure's level of instantiation, i.e., the size of its argument list. This leads to the abstract interpreter of Fig. 3.1, which uses a finite abstract domain. Evidently, the constraints in Fig. 3.1 are solvable by the usual fixedpoint computation, beginning with $\mathcal{C}l[\![e]\!] = \{\}$ for every program expression e: Observe that each equation is monotonic in the abstract domain. (Note that $\mathcal{C}l$ is dependent on the subject program p, so strictly, we should write $\mathcal{C}l_p$.)

Lemma 3.2.1 (Correctness of $\mathcal{C}l$) Let γ be the concretization map given in Fig. 3.1. Then for $\sigma; e$ reachable and $x \in FV(e)$, $\sigma(x) \in \gamma(\mathcal{C}l[\![x]\!])$. If $\sigma; e \Downarrow v$ then $v \in \gamma(\mathcal{C}l[\![e]\!])$. Further, for v' embedded at $f'^{(k)}$ in v, we have $v' \in \gamma(\mathcal{C}l[\![f'^{(k)}]\!])$.

Proof is by CTI. Let $P(v, e)$ be $v \in \gamma(\mathcal{C}l[\![e]\!])$. Equivalently, $P(v, e)$ asserts that if $v = \langle f, v_1, \ldots, v_n \rangle$ then $\langle\!\langle f, n \rangle\!\rangle \in \mathcal{C}l[\![e]\!]$.

- [B1]: $P(v, e)$ for $v \in Baseval \setminus \{\texttt{Err}\}$ is trivial.
- [B2]: $P(\langle f \rangle, f)$ follows from the definition of $\mathcal{C}l$.

- [I1]: Let $\sigma; (e_1\ e_2)$ be reachable. Suppose $\sigma; e_1 \Downarrow cl$ where $cl = \langle f, v_1, \ldots, v_n \rangle$ with $n < ar(f)$, and $\sigma; e_2 \Downarrow v_{n+1}$ where $v_{n+1} \neq \texttt{Err}$. By hypothesis, $P^*(cl, e_1)$ and $P^*(v_{n+1}, e_2)$ hold. $P^*(cl, e_1)$ implies that $\langle\!\langle f, n \rangle\!\rangle \in \mathcal{C}l[\![e_1]\!]$. And $P^*(v_{n+1}, e_2)$ implies that $v_{n+1} \in \gamma(\mathcal{C}l[\![e_2]\!])$. By the definition of $\mathcal{C}l$, it follows that $v_{n+1} \in \gamma(\mathcal{C}l[\![f^{(n+1)}]\!])$, so $P(v_{n+1}, f^{(n+1)})$ holds. Further, for v' embedded at $f'^{(k)}$ in v_{n+1}, we have that $P(v', f'^{(k)})$ holds, by $P^*(v_{n+1}, e_2)$. Thus $P^*(v_{n+1}, f^{(n+1)})$ holds, as desired.

- [I2]: We have to prove that if $P^*(v', e')$ holds for each subcomputation $\sigma'; e' \Downarrow v'$, then $P^*(v, e)$ holds for the present computation $\sigma; e \Downarrow v$, where $\sigma; e$ is reachable and e is a conditional, operation or application.

  We proceed by case analysis on the applicable evaluation rule for $\sigma; e$. Rule (3) is trivial, since the result v would be $\texttt{Err}$, which is in $\gamma(a)$ for any a. For rule (4), $e \equiv (\texttt{if}\ e_1\ e_2\ e_3)$ and $\sigma; e_3 \Downarrow v$. By the hypothesis, $P^*(v, e_3)$ holds. It follows that $v \in \gamma(\mathcal{C}l[\![e_3]\!])$. By the definition of $\mathcal{C}l$, $v \in \gamma(\mathcal{C}l[\![e]\!])$, so $P(v, e)$ holds. Now, for any v' embedded at $f'^{(k)}$ in v, $P(v', f'^{(k)})$ holds by $P^*(v, e_3)$. It follows that $P^*(v, e)$ holds. The reasoning is similar for rule (5). [I2] is trivial for evaluation rules (6), (7), (8) and (9). For rule (7) (primitive operations), note that v would not be a closure by semantic assumption [A6].

  For rule (10) to apply, $e \equiv (e_1\ e_2)$. Furthermore, $\sigma; e_1 \Downarrow cl$ where $cl = \langle f, v_1, \ldots, v_n \rangle$ with $n < ar(f) - 1$, and $\sigma; e_2 \Downarrow v_{n+1}$ where $v_{n+1} \neq \texttt{Err}$. By the hypothesis, $P^*(cl, e_1)$ and $P^*(v_{n+1}, e_2)$ hold. $P^*(cl, e_1)$ implies that $\langle\!\langle f, n \rangle\!\rangle \in \mathcal{C}l[\![e_1]\!]$, so by the definition of $\mathcal{C}l$, $\langle\!\langle f, n+1 \rangle\!\rangle \in \mathcal{C}l[\![e]\!]$. Now, $v = \langle f, v_1, \ldots, v_{n+1} \rangle$, so this means that $P(v, e)$ holds. Argue that $P^*(v_{n+1}, f^{(n+1)})$ holds as for [I1]. It follows from $P^*(v_{n+1}, f^{(n+1)})$ and $P^*(cl, e_1)$ that for v' embedded at $f'^{(k)}$ in v, $P(v', f'^{(k)})$ holds. We conclude that $P^*(v, e)$ holds.

  Finally, for rule (11) to apply, $e \equiv (e_1\ e_2)$. Further, $\sigma; e_1 \Downarrow cl$, where $cl = \langle f, v_1 \ldots, v_n \rangle$ with $n = ar(f) - 1$, $\sigma; e_2 \Downarrow v_{n+1}$ where $v_{n+1} \neq \texttt{Err}$ and $\sigma'; e^f \Downarrow v$ for $\sigma' = \sigma[f^{(i)} \mapsto v_i]_{i=1,n+1}$. By the hypothesis, the following hold: $P^*(cl, e_1)$ (so $\langle\!\langle f, n \rangle\!\rangle \in \mathcal{C}l[\![e_1]\!]$), $P^*(v_{n+1}, e_2)$, and $P^*(v, e^f)$. By $P^*(v, e^f)$, we have that $v \in \gamma(\mathcal{C}l[\![e^f]\!])$. Now by the definition of $\mathcal{C}l$, $v \in \gamma(\mathcal{C}l[\![e]\!])$, so $P(v, e)$ holds. For any v' embedded at $f'^{(k)}$ in v, $P(v', f'^{(k)})$ holds by $P^*(v, e^f)$. It follows that $P^*(v, e)$ holds.

By CTI, for $\sigma; e$ reachable and $x \in FV(e)$, $P^*(\sigma(x), x)$ holds, implying that $\sigma(x) \in \gamma(\mathcal{C}l[\![x]\!])$. If $\sigma; e \Downarrow v$, then $P^*(v, e)$ holds, implying that $v \in \gamma(\mathcal{C}l[\![e]\!])$. Further, for v' embedded at $f'^{(k)}$ in v, $P(v', f'^{(k)})$ holds by $P^*(v, e)$, so $v' \in \gamma(\mathcal{C}l[\![f'^{(k)}]\!])$. Informally, abstract interpretation $\mathcal{C}l$ safely approximates the runtime closures of any evaluation.

Remarks

Time complexity. A careful implementation of closure analysis [Hen91] has time-complexity cubic in the program size, but may be expected to perform well for programs that use higher-order facilities sparingly, or in unsophisticated ways. In practice, it makes sense to avoid fully-fledged closure analysis whenever possible [BJ93].

Precision of analysis. The closure analysis presented is an imprecise *context-insensitive* flow analysis. It is context-insensitive because the final output consists of a single inference for each program expression, and imprecise because it suffers from the *unrealizable path problem.* For example, consider the expressions `(id f1)` and `(id f2)`, where `id x = x`. Then, the abstract closures $\{\langle\!\langle \texttt{f1}, 0 \rangle\!\rangle, \langle\!\langle \texttt{f2}, 0 \rangle\!\rangle\}$ are associated with both expressions.

For *distributive flow problems*, precise context-insensitive solutions are possible [KU77], although their cost is probably too high to justify; for context insensitive analysis of functional programs, it is natural to allow information to flow together at the input parameters, unless extra precision is crucial.

Context-sensitive solutions are expensive to implement, although some degree of context-sensitivity is probably important for commonly-used functionals such as `map` and `fold`. In practice, it may be worthwhile to single out the library functionals for special treatment, and apply a low-precision but highly efficient method, e.g. Henglein's *linear-time simple closure analysis* [Hen91], in the general case.

3.3 Simple range analysis

A simple range analysis can be used to identify parameters that assume only natural numbers at runtime. The purpose of studying such an analysis is two-fold: to enable the handling of simple examples operating over the natural numbers, and to gain familiarity with abstract interpreters. The range analysis is interesting too because it is designed to be sensitive to control flow, unlike the simple closure analysis. Program analyses should be tailored to the expected programming style; the range analysis demonstrates how to detect common recursion over the natural numbers. The analysis is easily extended to handle more complicated situations. However, for a non-trivial use of range analysis to support difficult termination deductions, see Lindenstrauss et al.'s article on the automatic handling of the 91 function [DLSS99]. Such discussions take us too far from the object of our study, which are the termination criteria.

For the present analysis, we want to utilize certain facts:

- For expression `(if (`x` = 0) `e'` `e''`)`, if control reaches e'', then x has a non-zero value.
- For expression `(if (`x` <= 0) `e'` `e''`)`, if control reaches e'', then x has positive values.

Instead of introducing extra inferences to track the values of parameters at *every* program point, we transform the subject program, without affecting its semantics, as described below.

- For each expression of the form `(if (`x` = 0) `e'` `e''`)` in the subject program, replace e'' with a call $(f\ x_1\ \ldots\ x_n)$, where f is a fresh function identifier, and $x_1, \ldots, x_n$ are the free variables of e. The function definition $f\ x_1\ \ldots\ x_n = e$ is alpha-converted to avoid violating syntactic assumption [A1] (uniqueness of parameter identifiers), and added to the program.

  Let the new function definition be $f\ x'_1\ \ldots x'_n = e'$, and suppose that $x_i \equiv x$. Write $NE0(x'_i)$ to signify that x'_i never evaluates to (the *Baseval* representation of) 0, which follows from the semantics of L.

- For each expression of form `(if (`x` <= 0) `e'` `e''`)`, perform the same procedure as above, except substitute the predicate *GT0* for *NE0*. $GT0(x)$ signifies that x always evaluates to (the *Baseval* representation of) a positive integer.

A simple range analysis is given in Fig. 3.2.

Lemma 3.3.1 (Correctness of $\mathcal{R}$) For $\sigma; e$ reachable and $\sigma; e \Downarrow v$, we have $v \in \gamma(\mathcal{R}[\![e]\!])$, for the concretization map γ given in Fig. 3.2.

Proof Let $P(v, e)$ be $v \in \gamma(\mathcal{R}[\![e]\!])$ for the application of CTI. We omit the details.

Remarks

Time complexity. The time complexity of the standard worklist algorithm is $O(N \cdot H \cdot K)$, where N is the number of dataflow variables, in this case smaller than the program size, H is the number of insertions into the worklist due to changes in a single variable, and K is the cost of updating a dataflow variable. Updating a dataflow variable may involve evaluating the dataflow equation, as for $\mathcal{R}[\![e-1]\!]$, or computing a join for the existing value of the dataflow variable and the new value of another variable. For instance, when $\mathcal{R}[\![e^f]\!]$ receives a new value a, this causes $\mathcal{R}[\![(e_1\ e_2)]\!]$ to update to $(\mathcal{R}[\![(e_1\ e_2)]\!] \vee a)$, if we have $\langle\!\langle f, ar(f)-1\rangle\!\rangle \in \mathcal{C}l[\![e_1]\!]$. For range analysis $\mathcal{R}$, every update operation is constant-time. The value of H is $O(N)$ even though the abstract lattice has $O(1)$ height, as each dataflow variable can affect potentially $O(N)$ other variables. Time complexity of range analysis $\mathcal{R}$ is therefore $O(N^2)$.

Domains for range analysis. As for closure analysis, concrete and abstract domains are defined for the values of each variable. The concrete domain is $\mathcal{C} = \wp(\mathit{Val})$. The abstract domain $\mathcal{A}$ has just 3 elements: $\mathit{GT0} \leq \mathit{GE0} \leq \mathit{Any}$, signifying "greater than 0", "greater than or equal to 0", and "any value" respectively. The concretization map is:

$$\begin{array}{lcl}
\gamma(\mathit{GT0}) & = & \{1, 2, \ldots, \mathtt{Err}\} \\
\gamma(\mathit{GE0}) & = & \{0, 1, 2, \ldots, \mathtt{Err}\} \\
\gamma(\mathit{Any}) & = & \mathit{Val}
\end{array}$$

where $0, 1, 2, \ldots$ are the *Baseval* representations of the corresponding numbers.

Range analysis operator. The range analysis is given by operator $\mathcal{R} : \mathit{Expr} \to \mathcal{A}$ below.

$$\begin{array}{lcl}
\mathcal{R}[\![f^{(n)}]\!] & = & \left\{ \begin{array}{ll} \mathit{Any}, & \text{if } f \text{ is } f_{initial} \\ \mathit{GT0}, & \text{if } \mathit{GT0}(f^{(n)}), \\ \mathit{GT0}, & \text{if } a = \mathit{GE0} \text{ and } \mathit{NE0}(f^{(n)}), \\ a, & \text{otherwise,} \end{array} \right. \\
 & & \text{where } a = \bigvee\{\mathcal{R}[\![e_2]\!] \mid (e_1\ e_2) \text{ in program}, \langle\!\langle f, n-1 \rangle\!\rangle \in \mathcal{C}l[\![e_1]\!]\} \\
\mathcal{R}[\![f]\!] & = & \mathit{Any} \\
\mathcal{R}[\![(\mathtt{if}\ e_1\ e_2\ e_3)]\!] & = & \mathcal{R}[\![e_2]\!] \vee \mathcal{R}[\![e_3]\!] \\
\mathcal{R}[\![0]\!] & = & \mathit{GE0} \\
\mathcal{R}[\![c]\!] & = & \mathit{GT0} \text{ for } c \in \{1, 2, \ldots\} \\
\mathcal{R}[\![e-1]\!] & = & \left\{ \begin{array}{ll} \mathit{GE0}, & \text{if } \mathcal{R}[\![e]\!] = \mathit{GT0}, \\ \mathit{Any}, & \text{otherwise.} \end{array} \right. \\
\mathcal{R}[\![(op\ e_1\ \ldots\ e_n)]\!] & = & \mathit{Any} \text{ for other } op\text{'s} \\
\mathcal{R}[\![(e_1\ e_2)]\!] & = & \left\{ \begin{array}{ll} \mathit{Any}, \quad \text{if } \langle\!\langle f, n \rangle\!\rangle \in \mathcal{C}l[\![e_1]\!] \text{ and } n < ar(f) - 1, & \\ \bigvee\{\mathcal{R}[\![e^f]\!] \mid \langle\!\langle f, ar(f)-1 \rangle\!\rangle \in \mathcal{C}l[\![e_1]\!]\}, & \text{otherwise.} \end{array} \right.
\end{array}$$

Figure 3.2: Simple range analysis $\mathcal{R}$

Precision. Range analysis $\mathcal{R}$ is extremely basic. However, below are some examples demonstrating positive applications of the analysis.

1. Call with constant arguments.

```
goal a   = ack 3 3
ack m n  = if (m = 0) (n - 1)
              if (n = 0) (ack (m - 1) 1)
                 (ack (m - 1) (ack m (n - 1)))
```

   This program is transformed as follows:

```
goal a   = ack 3 3
ack m n  = if (m = 0) (n - 1)
              (f1 m n)
f1 v1 v2 = if (v2 = 0) (ack (v1 - 1) 1)
              (f2 v1 v2)
f2 v3 v4 = ack (v3 - 1) (ack v3 (v4 - 1))
```

   where it is known that $\mathit{NE0}(\texttt{v1})$ and $\mathit{NE0}(\texttt{v4})$ hold. Then $\mathcal{R}[\![\texttt{m}]\!] = \mathcal{R}[\![\texttt{n}]\!] = \mathcal{R}[\![\texttt{v2}]\!] = \mathit{GE0}$ and $\mathcal{R}[\![\texttt{v1}]\!] = \mathcal{R}[\![\texttt{v3}]\!] = \mathcal{R}[\![\texttt{v4}]\!] = \mathit{GT0}$ solve the $\mathcal{R}$-equations for the program.

2. Call with restricted arguments.

```
goal z   = if (z <= 0) 1
              (pow z z)
pow x n  = if (n = 0) x
              x * (pow x (n - 1))
```

   This program is transformed as follows:

```
goal z   = if (z <= 0) 1
              (f1 z)
f1 v1    = (pow v1 v1)
pow x n  = if (n = 0) x
              (f2 x n)
f2 v2 v3 = x * (pow v2 (v3 - 1))
```

   where it is known that $\mathit{GT0}(\texttt{v1})$ and $\mathit{NE0}(\texttt{v3})$ hold. Then $\mathcal{R}[\![\texttt{v1}]\!] = \mathcal{R}[\![\texttt{v3}]\!] = \mathit{GT0}$, $\mathcal{R}[\![\texttt{n}]\!] = \mathit{GE0}$, and $\mathcal{R}[\![\texttt{z}]\!] = \mathcal{R}[\![\texttt{x}]\!] = \mathcal{R}[\![\texttt{v2}]\!] = \mathit{Any}$ solve the $\mathcal{R}$-equations for the program.

3. An example not requiring any flow analysis.

```
oddnat k  = if (k <= 0) false
               if (k = 1) true
                  not (oddnat (k - 1))
```

   This program is transformed as follows:

```
oddnat k  = if (k <= 0) false
               (f1 k)
f1 v1     = if (v1 = 1) true
               not (oddnat (v1 - 1))
```

   where it is known, without requiring analysis, that $\mathit{GT0}(\texttt{v1})$ holds. The assignment $\mathcal{R}[\![\texttt{k}]\!] = \mathit{Any}$ and $\mathcal{R}[\![\texttt{v1}]\!] = \mathit{GT0}$ solve the $\mathcal{R}$-equations for the subject program.

Every expression in the subject program of form $e - 1$ where $\mathcal{R}[\![e]\!] = \mathit{GT0}$ can be transformed as $e \ominus 1$, where the partial subtraction operator $\ominus$ is interpreted as follows: $\mathcal{O}[\![\ominus]\!](n, 1)$ is $\mathcal{O}[\![-]\!](n, 1)$ if n represents a positive integer, and undefined otherwise. The operator $\ominus$ is also used directly in the code.

Point: After range analysis, the natural number operations are marked, in a source-level transformation. It follows from the correctness of $\mathcal{R}$ that the transformation is semantics preserving. The other analyses then regard the transformed program as the subject program. This approach allows the main analyses to be flow-insensitive (for easy porting of the theory to program specialization later).

3.4 Simple size analysis

Having briefly considered programs operating over the natural numbers, let us now turn our attention to programs operating over lists. For completeness, a simple, essentially first-order, size analysis is presented for L. The analysis is adapted from the Mercury analyzer's termination module [SSS97]. It demonstrates a more deductive approach to static program analysis.

The purpose of size analysis is to infer the size relation between the result of a function call and the sum of sizes of a subset of input arguments, *when the computation terminates and the result is not* `Err`. The selection of input arguments to sum over is determined for each function in a separate analysis.

Size analysis is critical for handling the sorting examples. For instance, consider the minsort program below.

```
sort xs      = (minsort (min (hd xs) (tl xs)) xs)

minsort m ys = m : (sort (remove m ys))

remove x zs  = if (hd zs = x) (tl zs)
                  hd zs : (remove x (tl zs))

min sofar vs = if (vs = nil) sofar
                  if (hd vs < sofar) (min (hd vs) (tl vs))
                     (min sofar (tl vs))
```

Function `sort` is recursively invoked with the argument `(remove m ys)`. If it is not determined that this evaluates to a value with smaller size than `ys` (whenever the computation is successful), then the termination of `sort` cannot be established.

3.4.1 The Mercury method

An illustrative example. Consider the usual `append` function:

```
append x y = if (x = nil) y
                (hd x : append (tl x) y)
```

Assume, by syntactic transformation, that every subject program function consists of a number of conditional cases, each of which is free from (nested) conditionals. The `append` program consists of two conditional cases: `y` and `(hd x : append (tl x) y)`.

The Mercury method assumes a *change value* γ_f for each function f, such that the size of the result of a call to f, assuming that the computation is successful, is no greater than the sum of sizes of *relevant* input arguments, plus γ_f. The choice of relevant arguments affects the precision of size analysis, but is otherwise arbitrary. They are determined in a separate analysis. For function `append`, both arguments are considered relevant.

A size expression is compiled for each conditional case, using the italicized version of a parameter identifier to denote the parameter's size, hd_x to denote the size of the result when (`hd` x) evaluates successfully (i.e., to a non `Err` value), and $\mathtt{tl}_x$ to denote the size of the result when (`tl` x) evaluates successfully. For the first conditional case of `append`, the size expression is simply y. By assumption, this is no greater than $x + y + \gamma_{\mathtt{append}}$, which leads to the following inequality:

$$x + y + \gamma_{\mathtt{append}} \geq y.$$

The size expression for the second conditional case is: $1 + hd_x + tl_x + y + \gamma_{\texttt{append}}$. Simplifying using the identity $hd_x + tl_x = x - 1$, we obtain $x + y + \gamma_{\texttt{append}}$. By assumption, this is an upper bound on the size of the result when the computation is successful. This leads to the other inequality for `append`:

$$x + y + \gamma_{\texttt{append}} \geq x + y + \gamma_{\texttt{append}},$$

which is trivially satisfied.

Claim: If these inequalities are simultaneously satisfied, then each compiled expression is indeed an upper bound on the size of whatever value the expression evaluates to successfully. For good results, the sum of γ variables can be minimized while satisfying the generated inequalities. For the `append` example, $\gamma_{\texttt{append}} = 0$ is a viable solution. Thus, it is concluded that the result of appending two lists, when successful, has size no greater than the sum of the input-list sizes. This is in accordance with intuition. The challenge is to formulate the above analysis for general L programs.

Pre-analysis preparations

1. **Program normalization.** We are familiar with the concept of *expression lifting* from the section on range analysis. By lifting, we refer to the process of replacing an expression e in the subject program by a call $(f\ x_1\ \ldots\ x_n)$, where f is a fresh function identifier, and $x_1, \ldots, x_n$ are the free variables of e. The function definition $f\ x_1\ \ldots\ x_n = e$ is alpha-converted, and then added to the program. Such a transformation is semantics preserving.

   Program normalization consists of repeated application of expression lifting, until every conditional is a function body. For technical convenience, it is assumed that in the transformed program, *every* function body is a conditional, whose branches are free from conditionals. (We may transform a non-conditional body e as (`if True` e e).)

2. **Program call graph.** The program call graph is available from closure analysis.

$$CG = \{g \to f \mid (e_1\ e_2) \in e^g, \langle\!\langle f, ar(f) - 1\rangle\!\rangle \in \mathcal{Cl}[\![e_1]\!]\}$$

   We can do the following efficiently: compute the strongly-connected components of CG, and sort them in reverse topological order [AHU75, CLR90]. Let $COM(p)$ be this list of components.

3. **Relevant arguments.** Assume, for now, a set of relevant input indices has been determined for each function of the subject program. This is treated after the size analysis.

4. **Generation of inequalities.** Given function f, with definition $f\ x_1\ \ldots x_N = e^f$, and relevant argument indices $\{i_1, \ldots, i_K\}$, define $LHS(f)$, the left-hand-side of an f inequality, as follows.

$$LHS(f) \equiv x_{i_1} + \ldots + x_{i_K} + \gamma_f$$

   $LHS(f)$ is intended as an upper bound on the size of any return value of f, i.e., any successful evaluation of e^f.

   Given a conditional case e of f, generate the size expression $RHS(e)$ recursively as described below. $RHS(e)$ is intended as an upper bound on the size of any successful evaluation of e.

   - If e is a parameter x, $RHS(e) = x$.
   - If e is a constant c, $RHS(e) = \#\mathcal{O}[\![c]\!]()$.
   - If $e \equiv (e_1 : e_2)$ then $RHS(e) = 1 + RHS(e_1) + RHS(e_2)$, provided each $RHS(e_i)$ is defined.
   - If $e \equiv$ (`hd` x) then $RHS(e) = hd_x$.
   - If $e \equiv$ (`tl` x) then $RHS(e) = tl_x$.
   - If $e \equiv (op\ e_1)$ where op is `hd` or `tl`, then $RHS(e) = RHS(e_1) - 1$, provided $RHS(e_1)$ is defined.

```
foreach C in COM(p)
   set INEQ := {} //where INEQ is a set of inequalities.
   for f ∈ C and e a conditional case of e^f
      if ineq(f, e) is defined then
        set INEQ := INEQ ∪ {ineq(f, e)}
      else
        goto Next
      end if
   end for
   minimize Σ_{f∈C}(γ_f) while satisfying INEQ
   if a solution exists then
     foreach f ∈ C record the value of γ_f according to the solution
   else
     goto Next
   end if
   Next: foreach f ∈ C record that γ_f does not exist
end for
```

Figure 3.3: Procedure for size analysis

- If $e \equiv (e_0 \ \ldots \ e_n)$, $\mathcal{Cl}[\![e_0]\!] = \{\langle\!\langle f, ar(f) - n\rangle\!\rangle\}$, each relevant argument index $i_k > ar(f) - n$ (i.e., no relevant argument is captured by the closure), $RHS(e_{i_k - ar(f)+n})$ is defined for each i_k, and γ_f is not yet determined, or is known to exist, then the size expression for e is given by $RHS(e) = \Sigma_k RHS(e_{i_k - ar(f)+n}) + \gamma_f$, where γ_f should be replaced by its value, if known.
- In every other situation, $RHS(e)$ is undefined.

5. **Simplification.** For function f and a conditional case e of f such that $RHS(e)$ is defined, the inequality $LHS(f) \geq RHS(e)$ is simplified as follows.

   - Apply identity $hd_x + tl_x = x - 1$ as far as possible.
   - Replace any remaining hd_x and tl_x with $x - 1$.
   - Drop an occurrence of the same variable from the LHS and RHS of the inequality repeatedly.
   - Drop all remaining variables (note: *not* γ unknowns) on the LHS.
   - Perform algebraic simplifications.

   These simplifications result in a *stricter* inequality (which imply the original one under the appropriate interpretation). If the simplified inequality contains only numbers and γ unknowns, assign it to $ineq(f, e)$.

3.4.2 The size analysis

The size analysis is described succinctly in Fig. 3.3 using the inequalities derived in the previous section. After executing the procedure, every γ_f is determined, for $f \in Fun$.

Definition 3.4.1 (Size analysis $\mathcal{S}$)

1. Define $S(e)$ as follows. It is intended to be an upper bound on the size of any successful evaluation of e.

$$S(e) = \begin{cases} LHS(f), & \text{if } e \text{ is a conditional } e^f \text{ and } \gamma_f \text{ exists, else} \\ RHS(e), & \text{if } RHS(e) \text{ is defined.} \end{cases}$$

Any γ unknowns in $S(e)$ should be replaced by their values.

2. Let $\mathcal{S}[\![e]\!] = \{\downarrow(x)\}$ signify that e evaluates to a value with smaller size than x, whenever the evaluation is successful. Concretely, $\mathcal{S}[\![e]\!] = \{\downarrow(x)\}$ if

   - $S(e)$ is defined and equivalent to $x - c$ for some positive c, or
   - e is $x \ominus 1$.

   Let $\mathcal{S}[\![e]\!] = \{\overline{\downarrow}(x)\}$ signify that e evaluates to a value with no greater size than x, whenever the evaluation is successful. Concretely, $\mathcal{S}[\![e]\!] = \{\overline{\downarrow}(x)\}$ if

   - $S(e)$ simplifies to x, or
   - e is the expression x.

   $\mathcal{S}[\![e]\!] = \{\}$ otherwise.

3.4.3 Properties of the size analysis

Correctness.

Definition 3.4.2 (Evaluation of size expression s wrt σ) $Ev_\#(s, \sigma)$ is a partial function defined recursively as follows.

$$
\begin{array}{lcl}
Ev_\#(x,\sigma) & = & \#\sigma(x) \\
Ev_\#(hd_x,\sigma) & = & \#\mathcal{O}[\![\mathtt{hd}]\!](\sigma(x)),\ \text{if } \sigma(x) \text{ is a non-empty list} \\
Ev_\#(tl_x,\sigma) & = & \#\mathcal{O}[\![\mathtt{tl}]\!](\sigma(x)),\ \text{if } \sigma(x) \text{ is a non-empty list} \\
Ev_\#(n,\sigma) & = & n,\ \text{if } n \text{ is a number} \\
Ev_\#(\Sigma_i s_i,\sigma) & = & \Sigma_i Ev_\#(s_i,\sigma)
\end{array}
$$

Lemma 3.4.3 Let $\sigma; e$ be reachable. Suppose that $S(e)$ is defined and $\sigma; e \Downarrow v$ where $v \neq \mathtt{Err}$. Then $Ev_\#(S(e), \sigma) \geq \#v$.

Proof Let e be a subexpression of e^g. Suppose that $\sigma; e$ is reachable, $\sigma; e \Downarrow v$ where $v \neq \mathtt{Err}$, and the size expression $S(e)$ is defined. We want to prove that $Ev_\#(S(e), \sigma) \geq \#v$, i.e., $Ev_\#(S(e), \sigma)$ is an upper bound on the size of v.

First, let F be the set of functions in the call-graph component containing g, or any earlier component in $COM(p)$. The proof of the lemma is by induction on the size of the proof tree for $\sigma; e \Downarrow v$, where e is a subexpression of e^f for some $f \in F$. Apply case analysis on the form of e.

- $e \equiv x$: $S(e) = x$, so $Ev_\#(S(e), \sigma) = \#\sigma(x) = \#v$.

- $e \equiv (\mathtt{if}\ e_1\ e_2\ e_3)$: Then e is e^f for some $f \in F$, and e_2 and e_3 are conditional cases of f. The applicable evaluation rule is (4) or (5). Since $S(e) = LHS(f)$ is defined, we deduce that γ_f exists, and that $RHS(e_2)$ and $RHS(e_3)$ are defined. If the applicable rule is (4), then $\sigma; e_2 \Downarrow v$, so by the induction hypothesis, $Ev_\#(RHS(e_2), \sigma) \geq \#v$. If the applicable rule is (5), then $\sigma; e_3 \Downarrow v$, so by the induction hypothesis, $Ev_\#(RHS(e_3), \sigma) \geq \#v$.

  By design, the simplification of $S(e) \geq RHS(e_2)$ is a valid inequality $s_1 \geq s_2$, where s_1 and s_2 are *numbers*. Thus $Ev_\#(s_1, \sigma) \geq Ev_\#(s_2, \sigma)$ is satisfied trivially for *any* σ. Check that if $s_1' \geq s_2'$ simplifies in one step to $s_1 \geq s_2$, and $Ev_\#(s_1, \sigma) \geq Ev_\#(s_2, \sigma)$ holds whenever the RHS is defined, then $Ev_\#(s_1', \sigma) \geq Ev_\#(s_2', \sigma)$ holds whenever the RHS is defined. It follows that $Ev_\#(S(e), \sigma) \geq Ev_\#(RHS(e_2), \sigma)$ holds for any σ for which $Ev_\#(RHS(e_2), \sigma)$ is defined. Similarly, $Ev_\#(S(e), \sigma) \geq Ev_\#(RHS(e_3), \sigma)$ holds for any σ for which $Ev_\#(RHS(e_3), \sigma)$ is defined. We conclude that, regardless of whether the applicable rule is (4) or (5), $Ev_\#(S(e), \sigma) \geq \#v$ holds.

- $e \equiv c$, where c is a constant: $S(e)$ is the number $\#\mathcal{O}[\![c]\!]() = \#v$.

- $e \equiv (e_1 : e_2)$: Since $v \neq \texttt{Err}$, for $i = 1, 2$, we have $\sigma; e_i \Downarrow v_i$ where $v_i \neq \texttt{Err}$. By the induction hypothesis, for each i, $Ev_{\#}(S(e_i), \sigma) \geq \#v_i$. Therefore,

$$\begin{array}{rcl} Ev_{\#}(S(e), \sigma) & = & Ev_{\#}(S(e_1) + S(e_2) + 1, \sigma) \\ & = & Ev_{\#}(S(e_1), \sigma) + Ev_{\#}(S(e_2), \sigma) + 1 \\ & \geq & \#v_1 + \#v_2 + 1 \\ & = & \#v \end{array}$$

- $e \equiv (\texttt{hd}\ x)$: As $v \neq \texttt{Err}$, we know that $\sigma(x)$ is a non-empty list. In this case, $S(e) = hd_x$ and $v = \mathcal{O}[\![\texttt{hd}]\!](\sigma(x))$. Therefore, $Ev_{\#}(S(e), \sigma) = \#\mathcal{O}[\![\texttt{hd}]\!](\sigma(x)) = \#v$.

- $e \equiv (\texttt{tl}\ x)$: As above.

- $e \equiv (op\ e_1)$, where op is `hd` or `tl`: Since $v \neq \texttt{Err}$, we know that e_1 evaluates in σ to some non-empty list v_1. By the induction hypothesis, $Ev_{\#}(S(e_1), \sigma) \geq \#v_1$. Therefore,

$$\begin{array}{rcl} Ev_{\#}(S(e), \sigma) & = & Ev_{\#}(S(e_1) - 1, \sigma) \\ & = & Ev_{\#}(S(e_1), \sigma) - 1 \\ & \geq & \#v_1 - 1 \\ & \geq & \#v \end{array}$$

- $e \equiv (e_0\ \ldots\ e_n)$ where $\mathcal{C}l[\![e_0]\!]$ has only one abstract closure of the form $\langle\!\langle f, ar(f) - n \rangle\!\rangle$: To avoid errors, e_0 must evaluate to a closure. This closure must be an f closure instantiated up to $ar(f) - n$ arguments, by Lemma 3.2.1 (correctness of $\mathcal{C}l$). Certainly, f is in F.

  By the L semantics, v is equal to the evaluation of $\sigma'; e^f$, where $\sigma'(f^{(i+ar(f)-n)})$ is some $v_i \neq \texttt{Err}$ such that $\sigma; e_i \Downarrow v_i$, for $i = 1, \ldots, n$. The evaluation $\sigma'; e^f$ is successful, since the result v is not `Err`. As $S(e)$ is defined, the relevant arguments are among $e_1, \ldots, e_n$, and γ_f exists. Let i_k represent a relevant argument index. By the inductive hypothesis, $Ev_{\#}(S(e_{i_k - ar(f)+n}), \sigma) \geq \#v_{i_k - ar(f)+n}$ for each k, and $Ev_{\#}(S(e^f), \sigma') \geq \#v$. Now, $S(e^f) = LHS(f) = \Sigma_k f^{(i_k)} + \gamma_f$. Therefore,

$$\begin{array}{rcl} Ev_{\#}(S(e), \sigma) & = & Ev_{\#}(\Sigma_k S(e_{i_k - ar(f)+n}) + \gamma_f, \sigma) \\ & = & \Sigma_k Ev_{\#}(S(e_{i_k - ar(f)+n}), \sigma) + \gamma_f \\ & \geq & \Sigma_k \#v_{i_k - ar(f)+n} + \gamma_f \\ & = & \Sigma_k \#\sigma'(f^{(i_k)}) + \gamma_f \\ & = & \Sigma_k Ev_{\#}(f^{(i_k)}, \sigma') + \gamma_f \\ & = & Ev_{\#}(\Sigma_k f^{(i_k)} + \gamma_f, \sigma') \\ & = & Ev_{\#}(S(e^f), \sigma') \\ & \geq & \#v \end{array}$$

- Every other case is vacuous since $S(e)$ would be undefined.

Corollary 3.4.4 (Correctness of $\mathcal{S}$) Let $\sigma; e$ be reachable. Suppose that $\sigma; e \Downarrow v$ where $v \neq \texttt{Err}$. Then $\downarrow(x) \in \mathcal{S}[\![e]\!]$ implies that $\#\sigma(x) > \#v$, and $\ddagger(x) \in \mathcal{S}[\![e]\!]$ implies that $\#\sigma(x) \geq \#v$.

Time complexity. The number of inequalities generated is linear in the size of the transformed program. It is NP-hard to decide if an integer linear program (ILP) is solvable. The relaxation problem can be solved in polynomial time by modern interior-point methods, although in practice, the simplex method works well. On the other hand, the linear programs encountered are usually extremely simple, such as those for the examples in the next section.

Precision. The size analysis is only geared towards first-order operations over lists, but is sufficient to support the termination analysis of various sorting routines. In practice, the definition of list-manipulating functions frequently uses functionals such as `map` and `fold`. For good results, duplication of these functionals for different invocations should be considered. This compensates for the lack of polyvariance in the analysis.

3.4.4 Examples of size analysis

The sorting programs below are examples of positive applications of the size analysis. Many other examples are available from the Mercury experiments [SSS97].

1. Minsort.

```
sort xs      = (minsort (min (hd xs) (tl xs)) xs)

minsort m ys = m : (sort (remove m ys))

remove x zs  = if (hd zs = x) (tl zs)
                  hd zs : (remove x (tl zs))

min sofar vs = if (vs = nil) sofar
                  if (hd vs < sofar) (min (hd vs) (tl vs))
                     (min sofar (tl vs))
```

   The relevant argument for `remove` is its second argument. Function `remove` is in its own strongly-connected component, and generated inequalities are: $\{\gamma_{\texttt{remove}} \geq -1, \gamma_{\texttt{remove}} \geq \gamma_{\texttt{remove}}\}$. Minimizing $\gamma_{\texttt{remove}}$, we obtain $\gamma_{\texttt{remove}} = -1$, which expresses the intuition that the result of evaluating a call to `remove`, when successful, has size at least one less than the size of `zs`. This in turn implies that the argument to `sort` in the body of `minsort` is decreased from `ys` (the argument has size expression: $ys - 1$). This ultimately enables us to deduce the termination of `sort`.

2. Quicksort.

```
qsort zs     = if (zs = nil) zs
                  (append (qsort (lt (hd zs) (tl zs)))
                          (hd zs : (qsort (ge (hd zs) (tl zs)))))

lt x xs      = if (xs = nil) nil
                  if ((hd xs) < x) (hd xs : (lt x (tl xs)))
                     (lt x (tl xs))

ge y ys      = if (ys = nil) nil
                  if ((hd ys) >= y) (hd ys : (ge y (tl ys)))
                     (ge y (tl ys))

append us vs = if (us = nil) vs
                  (hd us) : (append (tl us) vs)
```

   The relevant argument for `lt` is its second argument. Function `lt` is in its own strongly-connected component, and generated inequalities are: $\{\gamma_{\texttt{lt}} \geq 0, \gamma_{\texttt{lt}} \geq \gamma_{\texttt{lt}}, \gamma_{\texttt{lt}} \geq -1\}$. Minimizing $\gamma_{\texttt{lt}}$, we obtain: $\gamma_{\texttt{lt}} = 0$. A similar reasoning yields $\gamma_{\texttt{ge}} = 0$. This means that the recursive arguments to `qsort` are smaller in size than input argument `zs`: both of them have size expression $zs - 1$. This ultimately enables us to deduce the termination of `qsort`.

$$\mathcal{M} : \mathit{Expr} \to \wp(\mathit{Var})$$

$$\begin{array}{lcl}
\mathcal{M}[\![f^{(n)}]\!] & = & \{f^{(n)}\} \cup \{x \mid (e_1\ e_2) \text{ in program},\ \langle\!\langle f, n-1\rangle\!\rangle \in \mathcal{C}l[\![e_1]\!],\ x \in \mathcal{M}[\![e_2]\!]\} \\
\mathcal{M}[\![f]\!] & = & \{\} \\
\mathcal{M}[\![(\texttt{if}\ e_1\ e_2\ e_3)]\!] & = & \mathcal{M}[\![e_2]\!] \cup \mathcal{M}[\![e_3]\!] \\
\mathcal{M}[\![(op\ e_1\ \ldots\ e_n)]\!] & = & \bigcup_i \mathcal{M}[\![e_i]\!] \\
\mathcal{M}[\![(e_1\ e_2)]\!] & = & \bigcup\{\mathcal{M}[\![e^f]\!] \mid \langle\!\langle f, ar(f)-1\rangle\!\rangle \in \mathcal{C}l[\![e_1]\!]\}
\end{array}$$

Figure 3.4: Simple material dependency analysis $\mathcal{M}$

3.4.5 A simple size dependency analysis

An appropriate choice of relevant arguments is important for the success of the size analysis. An argument should be deemed relevant if it is used to construct the result. Thus, the arguments of `append` are relevant, but the first arguments of `lt` and `ge` in `qsort` are not.

However, an argument can affect the size of the result without being directly used in its construction, if there is recursive inspection of the argument. Consider the function `nullout` below.

```
nullout x = if (x = nil) nil
               (nil : (nullout (tl x))
```

The result of a `nullout` call is in fact related in size to the input value of `x`. While such an effect is probably rare in practice, it would be easy to design a flow analysis to detect recursively-inspected variables, and consider these relevant parameters.

The analysis given by $\mathcal{M}$ in Fig. 3.4 collects the *material dependencies* of an expression, i.e., all the parameters involved in constructing its result when it evaluates to a base value. The n-th parameter of function f is then deemed relevant if $f^{(n)} \in \mathcal{M}[\![e^f]\!]$. Using a membership-vector approach, the $\mathcal{M}$ dataflow equations can be solved in cubic time.

3.5 The Size Relation Graph (SRG)

3.5.1 Motivation of the SRG

The *size-relation graph* (henceforth SRG) describes decreasing and non-increasing data size relations observed in a function-call transition $\sigma; e \rightsquigarrow \sigma'; e'$. The intention is to derive a finite set of graphs $\mathcal{H}$ such that every possible function-call transition is described by some graph in $\mathcal{H}$. A graph sequence respecting program control flow is called a *multipath*. A set of multipaths describe the possible $\rightsquigarrow$ chains from an initial state. If it can be proved that every infinite multipath in this set has infinite descent, according to the arcs of the graphs, then no infinite $\rightsquigarrow$ chain is possible from an initial state. It follows from Corollary 2.4.4 that the subject program is terminating. This forms the semantic basis of our termination analysis.

The set of graphs $\mathcal{H}$ is said to satisfy *size-change termination* if it gives rise to infinite multipaths with infinite descent. (We define size-change termination formally at the end of this chapter.) Surprisingly, the problem of whether a given $\mathcal{H}$ satisfies size-change termination is decidable. This is discussed in the next chapter.

3.5.2 Definition of the SRG

The SRG is a bipartite graph of the form represented in Fig. 3.6 (on page 44), where between any pair of parameters x and y, there may be an arc labelled $\downarrow$ or $\ddagger$, but not both. We follow a convention when depicting SRGs: $\downarrow$ arcs are darkened, and $\overset{=}{\downarrow}$ labels are omitted. The verbal description of the SRG leads to the following definition.

Definition 3.5.1 (SRG) $G \subseteq \mathit{Var} \times \{\downarrow, \overset{=}{\downarrow}\} \times \mathit{Var}$ is a *size-relation graph (SRG)* if $x \xrightarrow{r} y, x \xrightarrow{r'} y \in G$ implies that $r = r'$. Let SRG be the (finite) set of all SRGs.

If the source and destination parameter sets of an SRG are identified with environments σ and σ' respectively, then the arc $x \xrightarrow{\downarrow} y$ in the SRG asserts that $\#\sigma(x) > \#\sigma'(y)$ and the arc $x \xrightarrow{\overset{=}{\downarrow}} y$ asserts that $\#\sigma(x) \geq \#\sigma'(y)$.

Definition 3.5.2 (Safety of SRG) SRG G is a *safe description* for environment pair (σ, σ') if

- for $x \xrightarrow{\downarrow} y \in G$, it holds that $\#\sigma(x) > \#\sigma'(y)$, and
- for $x \xrightarrow{\overset{=}{\downarrow}} y \in G$, it holds that $\#\sigma(x) \geq \#\sigma'(y)$.

SRGs are partially ordered according to the generality of the size relations they assert. The definition of $\leq_{SRG}$ below gives a partial ordering for SRG, the set of all SRGs.

Definition 3.5.3 (SRG partial ordering) For SRGs G and G', $G \leq_{SRG} G'$ if

- for $x \xrightarrow{\downarrow} y \in G'$: $x \xrightarrow{\downarrow} y \in G$, and
- for $x \xrightarrow{\overset{=}{\downarrow}} y \in G'$: there exists r such that $x \xrightarrow{r} y \in G$.

SRGs may be composed. Let G and G' be SRGs. If G asserts true size relations among parameter values of σ and σ', and G' asserts true size relations among parameter values of σ' and σ'', then their composition $G; G'$ asserts true size relations among parameter values of σ and σ'', deducible from G and G'. Formally, define composition as follows.

Definition 3.5.4 (SRG composition) For SRGs G and G', define $G;G' = G^{\downarrow} \cup G^{=}$, where

$$\begin{aligned} G^{\downarrow} &= \{x \xrightarrow{\downarrow} y \mid x \xrightarrow{r} z \in G, z \xrightarrow{r'} y \in G', r \text{ or } r' \text{ is } \downarrow\}, \\ G^{=} &= \{x \xrightarrow{\overset{=}{\downarrow}} y \mid x \xrightarrow{\overset{=}{\downarrow}} z \in G, z \xrightarrow{\overset{=}{\downarrow}} y \in G', x \xrightarrow{\downarrow} y \notin G^{\downarrow}\} \end{aligned}$$

The above composition is associative. The set of SRGs, together with composition, forms a monoid with the following identity.

Definition 3.5.5 (SRG composition identity) Define $G_{id} = \{x \xrightarrow{\overset{=}{\downarrow}} x \mid x \in \mathit{Var}\}$.

The analysis defined produces *fan-in-free* SRGs. This property turns out to be useful when approximating size-change termination.

Definition 3.5.6 (Fan-in-free SRG) An SRG G is *fan-in-free* if $x \xrightarrow{r} y, x' \xrightarrow{r'} y \in G$ implies $x = x'$.

3.5.3 SRG analysis

SRG analysis is an extension of the closure analysis of Fig. 3.1. A closure is represented abstractly by a function identifier, and a number recording the closure's level of instantiation. In addition, there is an SRG indicating the parameter size relations observed, should an environment in which the closure occurs be updated with the actual arguments of the closure. More concretely, suppose that the abstract closure $\langle\!\langle f, n, G \rangle\!\rangle$ describes $cl = \langle f, v_1, \ldots, v_n \rangle$, and let σ be an environment in which cl occurs, then G is a safe description for $(\sigma, \sigma[f^{(i)} \mapsto v_i]_{i=1,n})$, in the sense of Def. 3.5.2. It will be arranged for $\langle\!\langle f, n, G \rangle\!\rangle$ to concretize as a set containing the closure-environment pair (cl, σ), based on the size relations asserted by G.

Abstract closures and their operations

Abstract closures. The informal description above leads to the following definition. Note that an abstract closure's SRG contains arcs determined by the function identifier and level of instantiation alone, since $\sigma[f^{(i)} \mapsto v_i]_{i=1,n}$ is only slightly updated from σ. For the sake of clarity, the data representation has not been optimized.

Definition 3.5.7 An *abstract closure* has the form $\langle\!\langle f, n, G\rangle\!\rangle$, where $f \in \mathit{Fun}$, $n \leq \mathit{ar}(f)$, and G is an SRG such that

$$G = G' \cup \{x \xrightarrow{\overline{\downarrow}} x \mid x \notin \{f^{(1)}, \ldots, f^{(n)}\}\},$$

where each arc of G' has the form $x \xrightarrow{r} f^{(i)}$ with $i \leq n$. If $n = \mathit{ar}(f)$, then $\langle\!\langle f, n, G\rangle\!\rangle$ is also known as a *completed closure*. The set of all abstract closures is denoted ACl.

Ordering abstract closures. Abstract closures are partially ordered by the natural extension of $\leq_{SRG}$. For abstract closures $\langle\!\langle f, n, G\rangle\!\rangle$ and $\langle\!\langle f, n, G'\rangle\!\rangle$, we let $\langle\!\langle f, n, G\rangle\!\rangle \leq_{ACl} \langle\!\langle f, n, G'\rangle\!\rangle$ if G is more conservative (asserts stronger or a greater number of size relations) than G'. Formally, define $\leq_{ACl}$ as follows.

Definition 3.5.8 $\langle\!\langle f, n, G\rangle\!\rangle \leq_{ACl} \langle\!\langle f', n', G'\rangle\!\rangle$ if $f = f'$, $n = n'$ and $G \leq_{SRG} G'$.

Extending an abstract closure. We define operator *ext* such that the *extension* of abstract closure ϱ by size analysis output $\mathcal{S}[\![e]\!]$ corresponds to the capture of argument e by a closure that ϱ represents. Recall that $\mathcal{S}[\![e]\!]$ has the form $\{\}$, $\{\downarrow(x)\}$ or $\{\overline{\downarrow}(x)\}$.

Definition 3.5.9 For $R = \mathcal{S}[\![e]\!]$, define $\mathit{ext}_R\langle\!\langle f, n, G\rangle\!\rangle = \langle\!\langle f, n+1, G'\rangle\!\rangle$, where

$$G' = G \setminus \{f^{(n+1)} \xrightarrow{\overline{\downarrow}} f^{(n+1)}\} \cup \{x \xrightarrow{r} f^{(n+1)} \mid r(x) \in R\}.$$

Remark: The result is a legal abstract closure, and if G is fan-in-free, then so is G'.

Returning an abstract closure. An abstract closure describes a function closure *relative to an environment*, since it records the size relations among captured arguments, and parameter values in those environments in which the function closure may occur. When a closure is returned by a function call, its description in the caller environment is obtained by pre-composing with the completed closure representing the function-call transition.

Definition 3.5.10 (Composing with a completed closure) Define the composition of a completed closure with an abstract closure, $\langle\!\langle f', \mathit{ar}(f'), G'\rangle\!\rangle :: \langle\!\langle f, n, G\rangle\!\rangle$ as $\langle\!\langle f, n, G''\rangle\!\rangle$, where

$$G'' = \{x \xrightarrow{r} f^{(i)} \mid i \leq n,\ x \xrightarrow{r} f^{(i)} \in G'; G\} \cup \{x \xrightarrow{\overline{\downarrow}} x \mid x \notin \{f^{(1)}, \ldots, f^{(n)}\}\}.$$

Movement of closures and captured arguments that are closures. Let $\langle\!\langle f, n, G\rangle\!\rangle$ be an abstract closure. It is intended to represent some function closure $\langle f, v_1, \ldots, v_n\rangle$. An arc $x \xrightarrow{r} f^{(i)}$ of G asserts a size relation between v_i and the value of x in any environment in which the closure may occur. Given the semantics of L, the size relation is between v_i and the *last-observed* assignment to x. Under this rather natural interpretation, the movement of $\langle f, v_1, \ldots, v_n\rangle$ past an assignment of x is clearly seen to invalidate the arc $x \xrightarrow{r} f^{(i)}$. It is important to account for this when deriving abstract closures for each program variable.

It is also important to know whether the represented closure $\langle f, v_1, \ldots, v_n\rangle$ may occur as an argument to another closure, because any movement of the outer closure across a function-call, or function return, affects the validity of the arcs of G. The purpose of the development below will hopefully be clearer after considering the examples following the SRG analysis.

Definition 3.5.11 (Generalizing an abstract closure)

1. (f, n) CAP (f', n') is defined to hold just when a closure of form $\langle f', v_1, \ldots, v_{n'}\rangle$ may be assigned to $f^{(i)}$, for $i \leq n$.

   $$(f, n) \text{ CAP } (f', n') \equiv \langle\!\langle f', n'\rangle\!\rangle \in \mathcal{C}l[\![f^{(i)}]\!] \text{ where } i \leq n$$

   Relation CAP$^+$ is the transitive closure of CAP.

2. Parameter $f^{(i)}$ is a *dubious source parameter* (or simply *dubious*) for $\langle\!\langle f', n'\rangle\!\rangle$ if *either*

   - $(f, ar(f))$ CAP$^+$ (f', n'), *or*
   - $\langle\!\langle f'', n''\rangle\!\rangle \in \mathcal{C}l[\![e^f]\!]$ and (f'', n'') CAP$^+$ (f', n').

3. The generalization operation ∇ removes any arc with a dubious source parameter from an abstract closure's SRG:

   $$\nabla\langle\!\langle f, n, G\rangle\!\rangle = \langle\!\langle f, n, G'\rangle\!\rangle,$$
   $$\text{where } G' = \{x \xrightarrow{r} y \in G \mid y \in \{f^{(1)}, \ldots, f^{(n)}\} \text{ implies } x \text{ is not dubious for } \langle\!\langle f, n\rangle\!\rangle\}.$$

   If ς is a set of abstract closures, then $\nabla\, \varsigma = \{\nabla\, \varrho \mid \varrho \in \varsigma\}$.

Let $\langle\!\langle f', n', G'\rangle\!\rangle$ be an abstract closure. An arc of G' with a dubious source is not trusted because the actual closure represented by $\langle\!\langle f', n', G'\rangle\!\rangle$ may move into an environment where the source parameter has a value different from its value in the environment where the parameter was first captured.

The first item in Part 2 of the definition above accounts for the situation where the $\langle\!\langle f', n'\rangle\!\rangle$ closure, or some closure in which it is embedded, enters an environment where the dubious source is re-assigned. The second item accounts for the situation where the $\langle\!\langle f', n'\rangle\!\rangle$ closure is embedded in a closure *cl*, and *cl* is returned from the environment with the expected value for the dubious source.

A note on the use of ∇: An abstract closure $\langle\!\langle f, n, G\rangle\!\rangle$ is used to represent a function closure $\langle f, v_1, \ldots, v_n\rangle$. If some v_i is itself a closure, the dataflow solution attaches this information to $f^{(i)}$. In this case, the generalization ∇, applied to the abstract closures of $f^{(i)}$, accounts for changes in the environment around v_i due to any movement or completion of the f closure.

More typically, ∇ ensures that an abstract closure assigned to $f^{(i)}$ due to a call does not assert any size relations with respect to the parameters of f. The purpose of ∇ is to block out size relations involving past values of parameters. Such relations are not expected to be important for termination reasoning.

Abstract closure sets

Definition 3.5.12 (Lattice of abstract closure sets)

1. An *abstract closure set* is a set of abstract closures mutually incomparable under $\leq_{ACl}$. Formally, $\varsigma \subseteq ACl$ is an abstract closure set if for $\varrho, \varrho' \in \varsigma$, $\varrho \leq_{ACl} \varrho'$ implies that $\varrho = \varrho'$.

2. For abstract closure sets ς, ς', let $\varsigma \leq \varsigma'$ if $\forall\, \varrho \in \varsigma : \exists\, \varrho' \in \varsigma' : \varrho \leq_{ACl} \varrho'$.

Lemma 3.5.13 The abstract closure abstract sets with its ordering forms a complete lattice $\mathcal{A}$.

We shall use $\mathcal{A}$ as the abstract domain for SRG analysis.

Join for $\mathcal{A}$. For abstract closure sets ς and ς', $\varsigma \vee \varsigma'$ is obtained by repeatedly removing from $\varsigma \cup \varsigma'$ any abstract closure ϱ for which there exists ϱ' such that $\varrho \leq_{ACl} \varrho'$. Every subset $\varsigma \subseteq ACl$ can be regarded as an abstract closure set by identifying ς with $\bigvee\{\{\varrho\} \mid \varrho \in \varsigma\}$.

$$
\begin{array}{lcl}
\mathcal{G}[\![f^{(n)}]\!] & = & \nabla \bigvee\{\mathcal{G}[\![e_2]\!] \mid (e_1\ e_2) \text{ in program},\ \langle\!\langle f, n-1, G\rangle\!\rangle \in \mathcal{G}[\![e_1]\!]\} \\
\mathcal{G}[\![f]\!] & = & \{\langle\!\langle f, 0, G_{id}\rangle\!\rangle\} \\
\mathcal{G}[\![(\texttt{if}\ e_1\ e_2\ e_3)]\!] & = & \mathcal{G}[\![e_2]\!] \vee \mathcal{G}[\![e_3]\!] \\
\mathcal{G}[\![(op\ e_1\ \ldots\ e_n)]\!] & = & \{\} \\
\mathcal{G}[\![(e_1\ e_2)]\!] & = & \{ext_R\ \varrho \mid \varrho = \langle\!\langle f, n, G\rangle\!\rangle \in \mathcal{G}[\![e_1]\!],\ n < ar(f)-1\} \vee \\
 & & \{ext_R\ \varrho :: \varrho' \mid \varrho = \langle\!\langle f, ar(f)-1, G\rangle\!\rangle \in \mathcal{G}[\![e_1]\!],\ \varrho' \in \mathcal{G}[\![e^f]\!]\} \\
 & & \hfill \text{where } R = \mathcal{S}[\![e_2]\!].
\end{array}
$$

Figure 3.5: SRG analysis operator $\mathcal{G}$

The concrete domain $\mathcal{C}$. This is not the usual $\wp(Val)$. A concrete element will be taken as a pair, consisting of a closure and an environment. The concrete element (v, σ) expresses the idea of value v occurring in environment σ during an actual run of the subject program.

Actually, we will use $\mathcal{C}$ only as a convenient device to formulate the correctness criteria for SRG analysis. Since correctness is verified directly against the formal semantics of L, we are not relying on the concretization for interpreting the analysis results.

The concretization function.

Definition 3.5.14

1. Abstract closure $\langle\!\langle f, n, G\rangle\!\rangle$ is *safe for* (cl, σ) where $cl = \langle f, v_1, \ldots, v_n\rangle$ means:
   - $x \xrightarrow{\downarrow} f^{(i)} \in G$ and $i \leq n$ implies $\#\sigma(x) > \#v_i$.
   - $x \xrightarrow{\overline{\mp}} f^{(i)} \in G$ and $i \leq n$ implies $\#\sigma(x) \geq \#v_i$.
2. Take the concrete domain for SRG analysis to be $\wp(Val \times Env)$.
3. Define the concretization function γ for SRG analysis as follows.

$$
\begin{array}{lcl}
\gamma(\varsigma) & = & \{(cl, \sigma) \mid cl = \langle f, v_1, \ldots, v_n\rangle,\ \langle\!\langle f, n, G\rangle\!\rangle \in S \text{ is safe for } (cl, \sigma)\} \\
 & & \cup\ (Baseval \times Env)
\end{array}
$$

Remark: Function γ is monotonic and 1-1.

The SRG analysis operator

The SRG analysis operator $\mathcal{G}$ is defined in Fig. 3.5. The definition of $\mathcal{G}$ is mostly intuitive. Note that the equations for $\mathcal{G}$ are monotonic in $\mathcal{A}$, so a standard fixedpoint computation solves any set of $\mathcal{G}$ equations. Further, observe that every SRG generated is fan-in-free.

Definition 3.5.15 For subject program p, define $Tr(p)$ to be the set of *abstract function-call transitions* for p. These are essentially (abstract) completed closures.

$$Tr(p) = \{g \xrightarrow{G} f \mid (e_1\ e_2) \text{ is in } e^g,\ \varrho \in \mathcal{G}[\![e_1]\!],\ R = \mathcal{S}[\![e_2]\!],\ ext_R\ \varrho = \langle\!\langle f, ar(f), G\rangle\!\rangle\}$$

For $g \xrightarrow{G} f \in Tr(p)$, refer to g and f as the *source* and *target function* respectively.

Definition 3.5.16 (Safety of abstract function-call transitions) A set $\mathcal{H}$ of abstract function-call transitions is *safe for* the subject program p if for $\sigma; e$ reachable, e a subexpression of e^g, and $\sigma; e \rightsquigarrow \sigma'; e^f$ a function-call transition to f, there exists $g \xrightarrow{G} f \in \mathcal{H}$ such that G is a safe description for the environment pair (σ, σ').

Correctness of SRG analysis.

Lemma 3.5.17 Let $\sigma; (e_1\ e_2)$ be reachable, $\sigma; e_1 \Downarrow cl$ where $cl = \langle f, v_1, \ldots, v_n\rangle$, and $\sigma; e_2 \Downarrow v_{n+1}$ where $v_{n+1} \neq \texttt{Err}$.

1. Suppose that $n < ar(f) - 1$. Let $v = \langle f, v_1, \ldots, v_{n+1}\rangle$. If $\varrho = \langle\!\langle f, n, G\rangle\!\rangle$ is safe for (cl, σ), then $ext_R\ \varrho$, where $R = \mathcal{S}[\![e_2]\!]$, is safe for (v, σ).
2. Suppose that $n = ar(f) - 1$. If $\varrho = \langle\!\langle f, n, G\rangle\!\rangle$ is safe for (cl, σ), and $ext_R\ \varrho = \langle\!\langle f, n+1, G_1\rangle\!\rangle$ where $R = \mathcal{S}[\![e_2]\!]$, then G_1 is a safe description for (σ, σ') where $\sigma' = \sigma[f^{(i)} \mapsto v_i]_{i=1,n+1}$.
3. Suppose that $n = ar(f) - 1$. If $\varrho = \langle\!\langle f, n, G\rangle\!\rangle$ is safe for (cl, σ), $ext_R\ \varrho = \langle\!\langle f, n+1, G_1\rangle\!\rangle$ where $R = \mathcal{S}[\![e_2]\!]$, and $\langle\!\langle f', n', G'\rangle\!\rangle$ is safe for (cl', σ') where $\sigma' = \sigma[f^{(i)} \mapsto v_i]_{i=1,n+1}$, then the composition $\langle\!\langle f, n+1, G_1\rangle\!\rangle :: \langle\!\langle f', n', G'\rangle\!\rangle$ is safe for (cl', σ).

Proof These claims follow from Corollary 3.4.4 (correctness of $\mathcal{S}$) and the relevant definitions. We omit the details.

Lemma 3.5.18 (Safety of CAP)

1. For $\sigma; e^g$ reachable and $\sigma; e^g \Downarrow cl$ where $cl = \langle f, v_1, \ldots, v_n\rangle$, any $\langle f', v'_1, \ldots, v'_{n'}\rangle$ embedded in cl is such that (f, n) CAP$^+$ (f', n').
2. For $\sigma; e^g$ reachable, $x \in FV(e^g)$ and $\sigma(x) = \langle f, v_1, \ldots, v_n\rangle$, we have $(g, ar(g))$ CAP (f, n). Further, any $\langle f', v'_1, \ldots, v'_{n'}\rangle$ embedded in $\sigma(x)$ is such that $(g, ar(g))$ CAP$^+$ (f', n').

Proof These claims are proved by induction on the *depth of embedding* for the captured closure, using Lemma 3.2.1 (correctness of $\mathcal{C}l$). We omit the details.

Corollary 3.5.19 (of Lemma 3.5.18)

1. For $\sigma; e^g$ reachable and $\sigma; e^g \Downarrow cl$ where $cl = \langle f, v_1, \ldots, v_n\rangle$, if $\langle f', v'_1, \ldots, v'_{n'}\rangle$ is embedded in cl, then any parameter of g is dubious for $\langle\!\langle f, n'\rangle\!\rangle$.
2. Suppose that $\sigma; e^g$ is reachable, $x \in FV(e^g)$ and $\sigma(x) = \langle f, v_1, \ldots, v_n\rangle$. Then each parameter of g is dubious for $\langle\!\langle f, n\rangle\!\rangle$. Further, if $\langle f', v'_1, \ldots, v'_{n'}\rangle$ is embedded in $\sigma(x)$, then each parameter of g is dubious for $\langle\!\langle f', n'\rangle\!\rangle$.

Theorem 3.5.20 (Correctness of SRG analysis) For $\sigma; e$ reachable and $\sigma; e \Downarrow v$, the following holds: $(v, \sigma) \in \gamma(\mathcal{G}[\![e]\!])$, i.e., if v is a closure, then an abstract closure of $\mathcal{G}[\![e]\!]$ is safe for (v, σ).

Proof of the theorem is by CTI. Let $P(v, \sigma, e)$ be $(v, \sigma) \in \gamma(\mathcal{G}[\![e]\!])$. Then $P(v, \sigma, e)$ holds just when v is in *Baseval*, or an abstract closure in $\mathcal{G}[\![e]\!]$ is safe for (v, σ). Suppose that $\sigma; e$ is reachable.

- [B1]: $P(v, \sigma, e)$ for $v \in \mathit{Baseval} \setminus \{\texttt{Err}\}$ is trivial.
- [B2]: follows from $\mathcal{G}[\![f]\!] = \{\langle\!\langle f, 0, G_{id}\rangle\!\rangle\}$, and the concretization of abstract closure $\langle\!\langle f, 0, G_{id}\rangle\!\rangle$, which includes $(\langle f\rangle, \sigma)$ for any σ. By definition, $P(\langle f\rangle, \sigma, f)$ holds.
- [I1]: Let $\sigma; (e_1\ e_2)$ be reachable. And suppose that $\sigma; e_1 \Downarrow cl$, where $cl = \langle f, v_1, \ldots, v_n\rangle$ with $n < ar(f)$, and $\sigma; e_2 \Downarrow v_{n+1}$ where $v_{n+1} \neq \texttt{Err}$. By hypothesis, $P^*(cl, \sigma, e_1)$ and $P^*(v_{n+1}, \sigma, e_2)$ hold. Let $v_{n+1} \in \mathit{Clo}$. By $P^*(v_{n+1}, \sigma, e_2)$, there is an abstract closure ϱ in $\mathcal{G}[\![e_2]\!]$ safe for (v_{n+1}, σ). By $P^*(cl, \sigma, e_1)$, some $\langle\!\langle f, n, G\rangle\!\rangle$ is in $\mathcal{G}[\![e_1]\!]$, so according to the definition of $\mathcal{G}$, an approximation of ϱ is in $\mathcal{G}[\![f^{(n+1)}]\!]$, thus $P(v_{n+1}, \sigma, f^{(n+1)})$ holds. Further, for any closure cl' embedded at $f'^{(k)}$ in v_{n+1}, $P(cl', \sigma, f'^{(k)})$ holds by $P^*(v_{n+1}, \sigma, e_2)$. It follows that $P^*(v_{n+1}, \sigma, f^{(n+1)})$ holds.

- [I2]: Let $\sigma; e$ be reachable. We have to show that if $P(v', \sigma', e')$ holds for each subcomputation $\sigma'; e' \Downarrow v'$, then it holds for the present computation $\sigma; e \Downarrow v$, where e is a conditional, primitive operation or application. [I2] is trivial for primitive operations, and rules resulting in `Err`. Thus we only have to consider the rules (3)–(5) for conditionals, rule (10) for partial application and rule (11) for the function call.

  - For conditionals, [I2] follows easily from the definition of $\mathcal{G}$. Let $\sigma; e \Downarrow v$ where e has the form $(\texttt{if}\ e_1\ e_2\ e_3)$. By the semantic rules for L, v is `Err`, or $v = v_2$ where $\sigma; e_2 \Downarrow v_2$, or $v = v_3$ where $\sigma; e_3 \Downarrow v_3$. If v is `Err`, [I2] is trivially satisfied. Suppose that $v = v_2$. By hypothesis, $P^*(v_2, \sigma, e_2)$ holds. If v_2 is a closure, there is an abstract closure ϱ in $\mathcal{G}[\![e_2]\!]$ safe for (v_2, σ). By definition of $\mathcal{G}$, abstract closure ϱ also occurs in $\mathcal{G}[\![e]\!]$. Therefore, $P(v, \sigma, e)$ holds. Now by $P^*(v_2, \sigma, e_2)$, for any closure cl' embedded at $f'^{(k)}$ in v_2, there is an abstract closure in $\mathcal{G}[\![f'^{(k)}]\!]$ safe for (cl', σ). We conclude that $P^*(v, \sigma, e)$ holds. The argument is similar for $v = v_3$.

  - For partial applications, suppose $e \equiv (e_1\ e_2)$, $\sigma; e_1 \Downarrow cl$, where $cl = \langle f, v_1, \ldots, v_n \rangle$ with $n < ar(f) - 1$, and $\sigma; e_2 \Downarrow v_{n+1}$ where $v_{n+1} \neq \texttt{Err}$. Then the applicable evaluation rule is (10) and $v = \langle f, v_1, \ldots, v_{n+1} \rangle$. By hypothesis, there exists ϱ in $\mathcal{G}[\![e_1]\!]$ safe for (cl, σ). By Lemma 3.5.17, Part 1, $\varrho_1 = ext_R\ \varrho$ is safe for (v, σ), for $R = \mathcal{S}[\![e_2]\!]$. By definition of $\mathcal{G}$, ϱ_1 occurs in $\mathcal{G}[\![e]\!]$. Thus $P(v, \sigma, e)$ holds. Argue as for [I1] that $P^*(v_{n+1}, \sigma, f^{(n+1)})$ holds. It follows from $P^*(v_{n+1}, \sigma, f^{(n+1)})$ and $P^*(cl, \sigma, e_1)$ that for any closure cl' embedded at $f'^{(k)}$ in v, $P(cl', \sigma, f'^{(k)})$ holds. We conclude that $P^*(v, \sigma, e)$ holds, as desired.

  - Finally, let the applicable rule be (11). In this case, $e \equiv (e_1\ e_2)$. Suppose that $\sigma; e_1 \Downarrow cl$, where $cl = \langle f, v_1, \ldots, v_n \rangle$ with $n = ar(f) - 1$, $\sigma; e_2 \Downarrow v_{n+1}$ where $v_{n+1} \neq \texttt{Err}$, and $\sigma'; e^f \Downarrow v$, where $\sigma' = \sigma[f^{(i)} \mapsto v_i]_{i=1,n+1}$ and $v \in Clo$. By hypothesis $P^*(cl, \sigma, e_1)$, there exists ϱ in $\mathcal{G}[\![e_1]\!]$ safe for (cl, σ). By hypothesis $P^*(v, \sigma', e^f)$, there exists ϱ' in $\mathcal{G}[\![e^f]\!]$ safe for (v, σ'). By Lemma 3.5.17, Part 3, $\varrho'' = ext_R\ \varrho :: \varrho'$ is safe for (v, σ), for $R = \mathcal{S}[\![e_2]\!]$. According to the definition of $\mathcal{G}$, abstract closure ϱ'' is in $\mathcal{G}[\![e]\!]$. Thus, $P(v, \sigma, e)$ holds.

    For any closure cl' embedded at $f'^{(k)}$ in v, we have that $P(cl', \sigma', e^f)$ holds by hypothesis $P^*(v, \sigma', e^f)$, so some abstract closure ϱ in $\mathcal{G}[\![f'^{(k)}]\!]$ is safe for (cl', σ'). *Claim*: This abstract closure is safe for (cl', σ) due to ∇. By Corollary 3.5.19, Part 1, each parameter of f is dubious for any abstract closure matching the function and instantiation level of cl'. By definition of ∇, the SRG of ϱ does not contain any size relation arc with a source parameter in $Param(f)$. As σ and σ' only differ at these parameters, abstract closure ϱ is indeed safe for (cl', σ). Thus, $P(cl', \sigma, f'^{(k)})$ holds. As we have already argued that $P(v, \sigma, e)$ holds, we conclude that $P^*(v, \sigma, e)$ holds, as desired.

- [I3]: Let $\sigma; e \rightsquigarrow \sigma'; e^f$. For [I3] to be non-vacuous, $FV(e^f)$ is non-empty, so the transition is a function-call transition to f. For $x \in FV(e^f)$, we have to show that $P^*(\sigma'(x), \sigma, x)$ implies $P^*(\sigma'(x), \sigma', x)$, i.e. the validity of an inference is unaffected by a call. Now, it follows from the premise that there is an abstract closure in $\mathcal{G}[\![x]\!]$ safe for $(\sigma'(x), \sigma)$, for $\sigma'(x) \in Clo$, and for any closure cl' embedded at $f'^{(k)}$ in $\sigma'(x)$, there is an abstract closure in $\mathcal{G}[\![f'^{(k)}]\!]$ safe for (cl', σ). By Corollary 3.5.19, Part 2, the parameters of f are dubious for these abstract closures. The abstract closures therefore do not assert any size relation with source parameter in $Param(f)$ (this is the action of ∇). Safety with respect to σ then implies safety with respect to σ', since σ and σ' differ only at parameters of f. We conclude that $P^*(\sigma'(x), \sigma', x)$ holds.

Corollary 3.5.21 $Tr(p)$ is safe for p.

Proof Let e be a subexpression of e^g. Suppose that $\sigma; e$ is reachable, and $\sigma; e \rightsquigarrow \sigma'; e^f$ is a function-call transition to f. We claim that there exists $g \xrightarrow{G} f \in Tr(p)$ such that for $x \xrightarrow{\downarrow} y \in G$, it holds that $\#\sigma(x) > \#\sigma'(y)$, and for $x \xrightarrow{\overline{\downarrow}} y \in G$, it holds that $\#\sigma(x) \geq \#\sigma'(y)$.

We deduce that the expression e above has the form $(e_1\ e_2)$, and $\sigma; e_1 \Downarrow cl$ where $cl = \langle f, v_1, \ldots, v_n\rangle$ with $n = ar(f) - 1$. Further, $\sigma; e_2 \Downarrow v_{n+1}$ where $v_{n+1} \neq \texttt{Err}$. By Theorem 3.5.20, there is an abstract closure ϱ in $\mathcal{G}[\![e_1]\!]$ safe for (cl, σ). By Lemma 3.5.17, Part 2, for $R = \mathcal{S}[\![e_2]\!]$, abstract closure $\varrho_1 = ext_R\ \varrho$ has SRG G that is a safe description for environment pair (σ, σ'), where $\sigma' = \sigma[f^{(i)} \mapsto v_i]_{i=1,n+1}$. By definition of $Tr(p)$, $g \xrightarrow{G} f$ is in $Tr(p)$. The claim follows.

Time complexity.

Let A be the maximum function arity, N be the size of the subject program, and K be the number of abstract closures generated by SRG analysis. The bad news is: K can be exponential in $A \sim O(N)$. Let δ be the maximum number of non-consecutive partial applications that a closure experiences in some execution of the program. Then K is possibly exponential in δ, as each extension could conditionally add or omit an arc to an SRG, generating an exponential number of different ones. However, δ is expected to be small. In fact, for Scheme programs, $\delta = 2$.

The other source of combinatorial explosion is the composition operation in the equation for $\mathcal{G}[\![(e_1\ e_2)]\!]$. This is a problem only if the subject program uses higher-order facilities in sophisticated ways. We are consoled by the following.

1. K is small if higher-order facilities are used sparingly.

2. Return closure values occur less frequently than closure parameters, so the composition operation in $\mathcal{G}[\![(e_1\ e_2)]\!]$ may not give problems, although some approximation to curb exponential growth should be considered.

3. An abstract closure set can be represented in space $O(AK)$. If closure values are *never* returned, the cost of the analysis varies as AK^2N: cost of a dataflow variable update times the number of insertions into the worklist due to changes in the dataflow variables. We normally expect manageable values for K.

4. For essentially first-order programs, SRG analysis is $O(AN)$.

Precision.

The following are some remarks about the SRG analysis.

1. Embedded closures are not modelled explicitly: An abstract element $\langle\!\langle f, n, G\rangle\!\rangle$ is used to represent a function closure $\langle f, v_1, \ldots, v_n\rangle$, but if v_i is itself a closure, an abstract element representing this closure is attached to $f^{(i)}$ by the dataflow solution, *without noting any context information.* (Compare with the grammar-based approach of [AH96].)

   This scheme trades precision for efficiency. The information lost is a form of context, analogous to the calling context.

2. The dataflow equations for SRG analysis are insensitive to the calling context.

   A practical way to recover precision that is lost due to context insensitivity is to duplicate the more commonly referenced functionals. Attention should be paid in particular to library functionals such as `map` and `fold`. We leave the study of the context sensitivity issue as future work.

3. The examples below show that straightforward use of higher-order facilities does not obscure critical decreasing size relations from SRG analysis.

   CPS programs are *not* among the examples. Their termination requires observing the size decreases of continuation values. This is actually easy enough, although it does require defining a more sophisticated size function. To avoid distracting from the presentation of the basic analysis, we leave the treatment of CPS programs to the final chapter. CPS programs are also important enough to warrant special attention.

Example 1. Indirect recursion with obvious descent.

```
f x y = (k (f x) y)
k c z = (c (tl z))
```

Here the critical descent for termination occurs at the call `(c (tl z))`, in the argument that completes the call. For this example, SRG analysis $\mathcal{G}$ is unnecessary, because from closure analysis alone, it is known that `c` has to evaluate to a $\langle\!\langle \mathtt{f}, 1 \rangle\!\rangle$ closure, if it evaluates to a closure at all. Thus, the abstract function-call transition $\mathtt{k} \xrightarrow{G} \mathtt{f}$ where $G = \{\mathtt{z} \xrightarrow{\downarrow} \mathtt{y}\}$ is safe for any function-call transition from `k` to `f`. The abstract function-call transition $\mathtt{f} \xrightarrow{G'} \mathtt{k}$ where $G' = \{\mathtt{y} \xrightarrow{\overset{=}{\downarrow}} \mathtt{z}\}$ is clearly safe for any function-call transition from `f` to `k`. These abstract function-call transitions cover every possible function-call transition for the program. Every flow-legal sequence of them (which has $\mathtt{k} \xrightarrow{G} \mathtt{f}$ and $\mathtt{f} \xrightarrow{G'} \mathtt{k}$ in alternation) has infinite descent, according to the arcs of the SRGs. This implies that no infinite computation is possible, as we will see at the end of this chapter.

The example can therefore be handled by a combination of size analysis and closure analysis. From the point of view of SRG analysis, it is not essentially different from the following first-order example:

```
f y = (k y)
k z = (f (tl z))
```

Tracing through the SRG analysis (for the original example): $\mathcal{G}[\![\mathtt{c}]\!]$ evaluates to a set containing just the abstract closure $\langle\!\langle \mathtt{f}, 1, G_{id} \rangle\!\rangle$, representing closures due to `(f x)`. The SRG here is G_{id} because the arc $\mathtt{x} \xrightarrow{\overset{=}{\downarrow}} \mathtt{x}$ is first removed, and then re-inserted by *ext*. The operation ∇ has no effect, since an `f` closure is never captured by an `f` parameter, so `x` is not dubious. The possible abstract function-call transitions work out as expected. Any function-call transition from `k` to `f` at `(c (tl z))` has environments that are safely described by the SRG of completed closure $\langle\!\langle \mathtt{f}, 2, G \rangle\!\rangle$, where $G = \{\mathtt{x} \xrightarrow{\overset{=}{\downarrow}} \mathtt{x}, \mathtt{z} \xrightarrow{\downarrow} \mathtt{y}, \mathtt{c} \xrightarrow{\overset{=}{\downarrow}} \mathtt{c}, \mathtt{z} \xrightarrow{\overset{=}{\downarrow}} \mathtt{z}\}$. Any function-call transition from `f` to `k` has environments that are safely described by the SRG of completed closure $\langle\!\langle \mathtt{k}, 2, G' \rangle\!\rangle$, where $G' = \{\mathtt{x} \xrightarrow{\overset{=}{\downarrow}} \mathtt{x}, \mathtt{y} \xrightarrow{\overset{=}{\downarrow}} \mathtt{y}, \mathtt{y} \xrightarrow{\overset{=}{\downarrow}} \mathtt{z}\}$. A safe set of abstract function-call transitions is depicted pictorially in Fig. 3.6. It is clear from the diagram that every infinite flow-legal sequence of abstract function-call transitions has infinite descent. (Simply trace the arcs between `y` and `z` in each SRG of the sequence.)

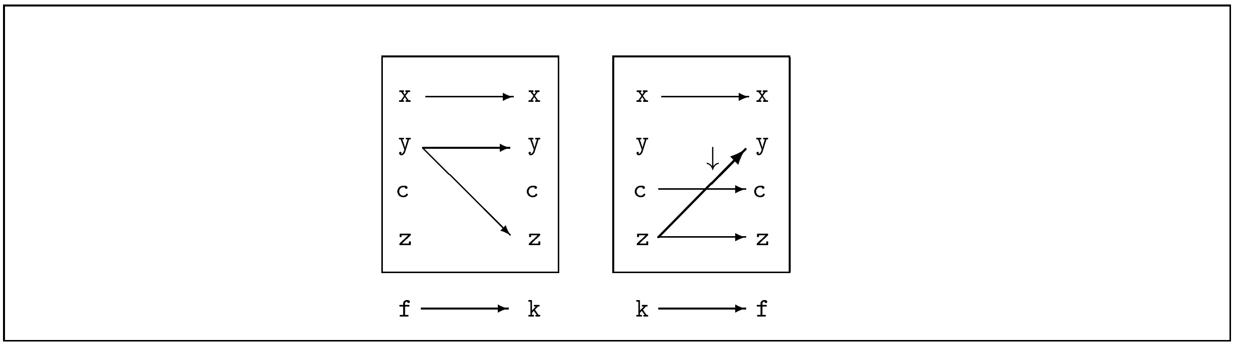


Figure 3.6: Abstract function-call transitions for Ex. 1

Example 2. Indirect recursion with less obvious descent.

```
f x y = (k (f (tl x)) y)
k c z = (c z)
```

This is a variation of Ex. 1, for which a combination of first-order size analysis and closure analysis would be insufficient to deduce termination. The problem is that the critical descent in parameter `x` occurs during the construction of an `f` closure. The closure subsequently moves into a different environment before it is completed. Tracing through the fixedpoint computations for $\mathcal{G}$ yields a similar set of abstract function-call transitions as for Ex. 1, with the $\downarrow$ label at a different place. See Fig. 3.7. It is clear from the diagram that every infinite flow-legal sequence of abstract function-call transitions has infinite descent.

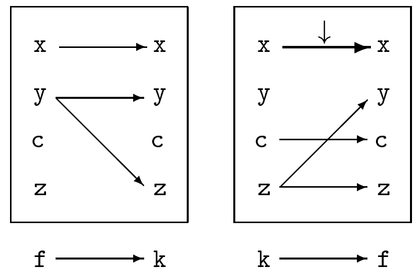


Figure 3.7: Abstract function-call transitions for Ex. 2

Example 3. Another example of "captured" descent.

```
trunc0 i st = (trunc1 i (thaw st))
trunc1 j tr = if (tr = nil) nil
                 (hd tr) : (map (trunc0 (j ⊖ 1)) (tl tr))
map f ls    = if (ls = nil) nil
                 (f (hd ls)) : (map f (tl ls))
```

Variable `st` contains a stream (i.e., a lazy list) used to represent a possibly infinite tree. Function `thaw`, whose definition is not given, constructs the top level structure, which is a list with elements $x, st_1, \ldots, st_n$, where each st_i is a stream. The purpose of `trunc0` is to extract the initial levels of the tree represented by the stream in `st`, to a depth indicated by the value in `i`.

Operation `thaw` effectively disallows any descent reasoning with respect to `st` or `tr`. The possible abstract function-call transitions are represented in Fig. 3.8. Observe that any infinite flow-legal sequence that cycles around `map` has infinite descent in `ls`, and every other infinite flow-legal sequence consists of infinite repetitions of $\rightarrow$ `trunc0` $\rightarrow$ `trunc1` $\rightarrow$ `map`$(\rightarrow$ `map`$)^*$, which implies infinite descent in `i` and `j`.

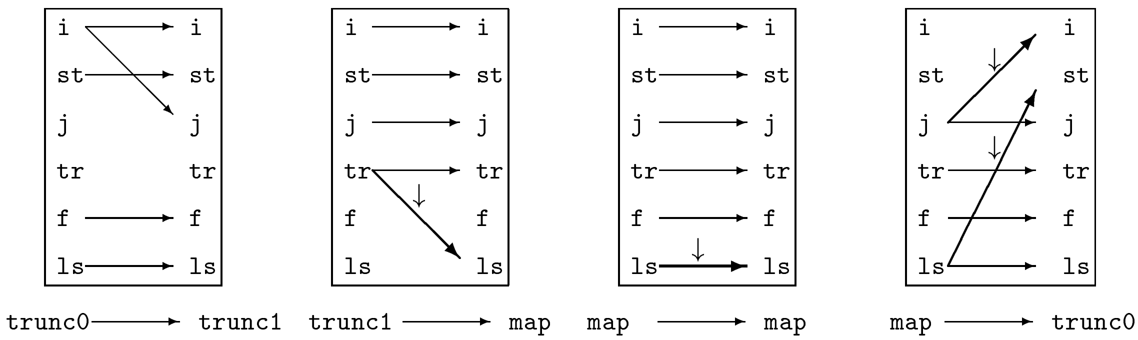


Figure 3.8: Abstract function-call transitions for Ex. 3

Example 4. Positive example for approximation with ∇.

```
f x k = if ... (f (tl x) (f (tl x)))
               (k k)
```

This example shows that ∇ is necessary. Consider the $\leadsto$ chain starting with a function-call transition due to the expression `(f (tl x) (f (tl x)))`, followed by infinitely many transitions due to the expression `(k k)`. Let `x` have initial value `[[]]`. Then the sequence of calls, which is legal, has no infinite descent, since each environment in the sequence maps `x` to `[]` and `k` to $\langle \texttt{f}, \texttt{[]} \rangle$. However, without ∇ in the equation for $\mathcal{G}[\![f^{(n)}]\!]$, we would have $\mathcal{G}[\![\mathtt{k}]\!] = \{\langle\!\langle f, 1, G \rangle\!\rangle\}$, where G contains $\texttt{x} \xrightarrow{\downarrow} \texttt{x}$. This would lead to an *erroneous* termination proof. Intuitively, the problem is that the size relation asserted by the arc $\texttt{x} \xrightarrow{\downarrow} \texttt{x}$ is invalidated after the `f` closure $\langle \texttt{f}, \texttt{[]} \rangle$ has moved into an `f` environment different from the one where the descent in `x` was first captured. The conservative abstract function-call transitions generated by $\mathcal{G}$ are given in Fig. 3.9. The $\texttt{x} \xrightarrow{\downarrow} \texttt{x}$ arc in the SRG for the abstract function-call transition due to the expression `(k k)` has been removed thanks to ∇, which recognizes that `x` is dubious for $\langle\!\langle \texttt{f}, 1 \rangle\!\rangle$ because a closure represented by it may be captured by the `f` parameter `k`.

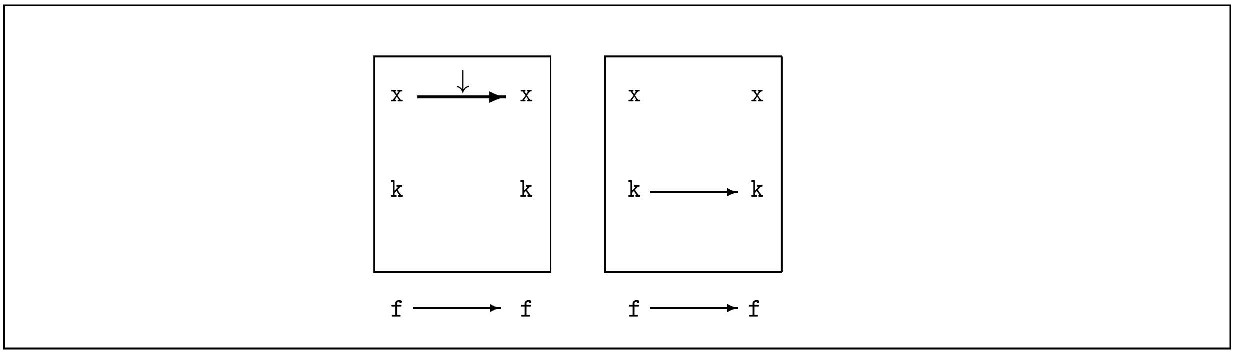


Figure 3.9: Abstract function-call transitions for Ex. 4

Example 5. Negative example for ∇.

```
f x k = if ... (f (tl x) (f (tl (tl x))))
               (k (f (tl (tl x))))
```

This is a variation of Ex. 4. It not admits a termination reasoning based on size descent, it exhibits size decreases in `x` at *every* recursive call. However, deducing this automatically would require a much more complicated analysis than SRG analysis. At the very least, it would be necessary to deduce size relations among computed arguments. (For the example, it is important to observe that `(tl (tl x))` always results in a value with smaller size than `(tl x)`.) Due to ∇ in the $\mathcal{G}[\![f^{(n)}]\!]$ equation, the abstract function-call transition due to `(k (f (tl (tl x))))` has an SRG where the arc $\texttt{x} \xrightarrow{\downarrow} \texttt{x}$ is absent. Therefore there is no descent in `x`, and it is conservatively concluded that the program might not terminate.

Example 6 Closures within closures.

```
YComb f k x = (f (k k) x)
Ffact H n   = if (n = 0) 1 (n * (H (n - 1)))
fact z      = (Ffact (YComb Ffact (YComb Ffact)) z)
```

This Y-combinator example illustrates how recursion can be implemented indirectly in a substitution calculus. Function `fact` computes the factorial of its argument. The effect of recursion is produced by the self application `(k k)`, which always evaluates to $\langle \texttt{YComb}, \texttt{Ffact}, \langle \texttt{YComb}, \texttt{Ffact} \rangle \rangle$. Despite the presence of such a closure, the critical descent for termination is obvious. SRG analysis $\mathcal{G}$ readily generates the set of abstract function-call transitions depicted in Fig. 3.10.

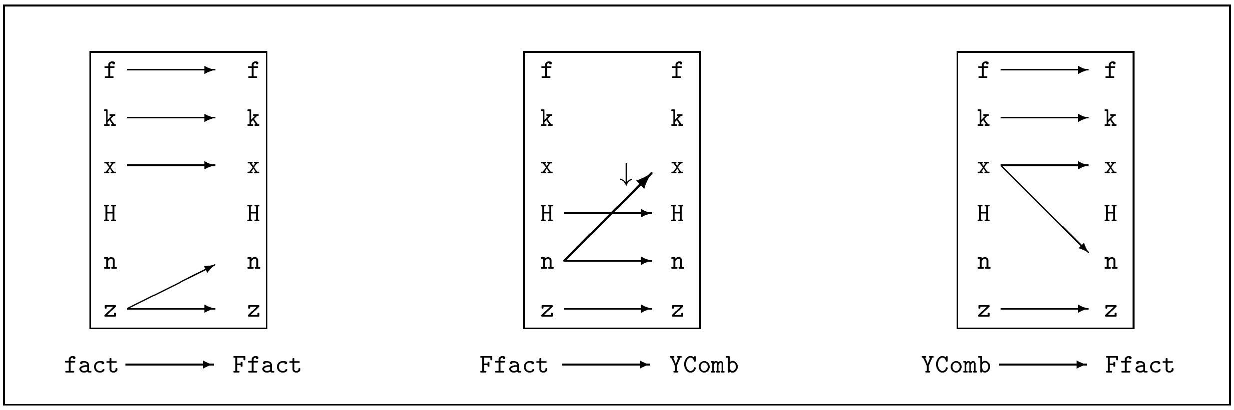


Figure 3.10: Abstract function-call transitions for Ex. 6

Example 7. Return closures.

```
f x        = (letbind x x)
letbind v  = (letbody (tl v))
letbody y z = z : (f y)
```

This example implements a let-binding in stages. Expression `letbind x` computes a value v from `x`, a parameter of `f`, and returns a closure that (when completed) evaluates an expression referring to v. This is a rather contrived way to program in L, but triggers the composition operation when computing the fixedpoint for $\mathcal{G}$ equations.

There is a call from `f` to `letbind`, but this is not recursive. The possible function-call transitions from `f` to `letbody` and vice versa are the interesting ones to model. The transitions from `letbody` to `f` are straightforward: The abstract function-call transition for them has an SRG containing the arc `y` $\overset{\overline{\downarrow}}{\to}$ `x`. The expression `letbind x` evaluates to a closure. Its abstract representation is generated by composition of the (abstract) completed closure from `letbind` to `letbody`, whose SRG has the arc `x` $\overset{\overline{\downarrow}}{\to}$ `v`, and the abstract closure for expression `(letbody (tl v))`, whose SRG has the arc `v` $\overset{\downarrow}{\to}$ `y`. The abstract closure for `letbind x` therefore has an SRG with the arc `x` $\overset{\downarrow}{\to}$ `y`. Thus, the abstract function-call transition from `f` to `letbody` has SRG with this arc. The recursive function-call transitions between `f` and `letbody` are modelled by abstract elements, for which every infinite flow-legal sequence has infinite descent in `y` and `x`. The set of abstract function-call transitions generated by $\mathcal{G}$ are depicted in Fig. 3.11.

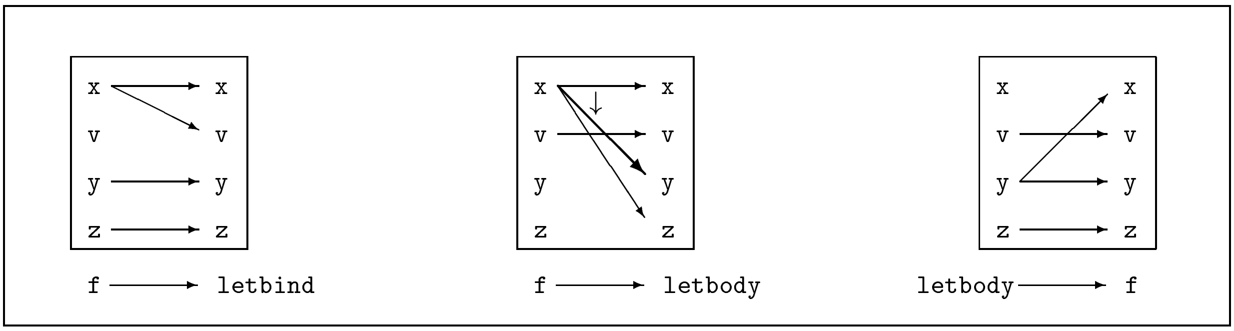


Figure 3.11: Abstract function-call transitions for Ex. 7

3.5.4 Multipaths

A multipath is a finite or infinite flow-legal sequence of abstract function-call transitions. It describes the size relations among the parameter values of consecutive environments within a $\rightsquigarrow$ chain.

Definition 3.5.22 (Multipath) Let *Tr* be a set of abstract function-call transitions.

1. A *Tr*-multipath $\mathcal{M}$ is a finite or infinite sequence of *Tr* elements of the form:

   $$f_0 \stackrel{G_1}{\rightarrow} f_1, f_1 \stackrel{G_2}{\rightarrow} f_2, \ldots, f_t \stackrel{G_{t+1}}{\rightarrow} f_{t+1}, \ldots$$

   We also write:

   $$f_0 \stackrel{G_1}{\rightarrow} f_1 \stackrel{G_2}{\rightarrow} f_2 \ldots \stackrel{G_{t+1}}{\rightarrow} f_{t+1} \ldots$$

   Fig. 3.12 shows an example of a multipath. A sequence of *Tr* elements of the above form is described as *flow-legal*, i.e., a multipath is a flow-legal sequence of abstract function-call transitions.

2. A thread *th* of multipath $\mathcal{M}$ is any finite or infinite connected sequence of arcs of the form:

   $$x_0 \stackrel{r_1}{\rightarrow} x_1, x_1 \stackrel{r_2}{\rightarrow} x_2, \ldots, x_t \stackrel{r_{t+1}}{\rightarrow} x_{t+1}, \ldots$$

   such that $\exists\, K \geq 0, \forall\, t \geq 0 : x_t \stackrel{r_{t+1}}{\rightarrow} x_{t+1} \in G_{K+t+1}$.

   We also write:

   $$x_0 \stackrel{r_1}{\rightarrow} x_1 \stackrel{r_2}{\rightarrow} x_2 \ldots \stackrel{r_{t+1}}{\rightarrow} x_{t+1} \ldots$$

   A thread is *maximal* if the sequence of arcs is maximal in the multipath. A thread is *complete* if it extends the entire length of the multipath.

3. An infinite thread *th* of the above form has *infinite descent* or is *infinitely descending* if $\{t \mid r_t = \downarrow\}$ is infinite. Multipath $\mathcal{M}$ has infinite descent if it has a thread with infinite descent. The multipath in Fig. 3.12 has infinite descent.

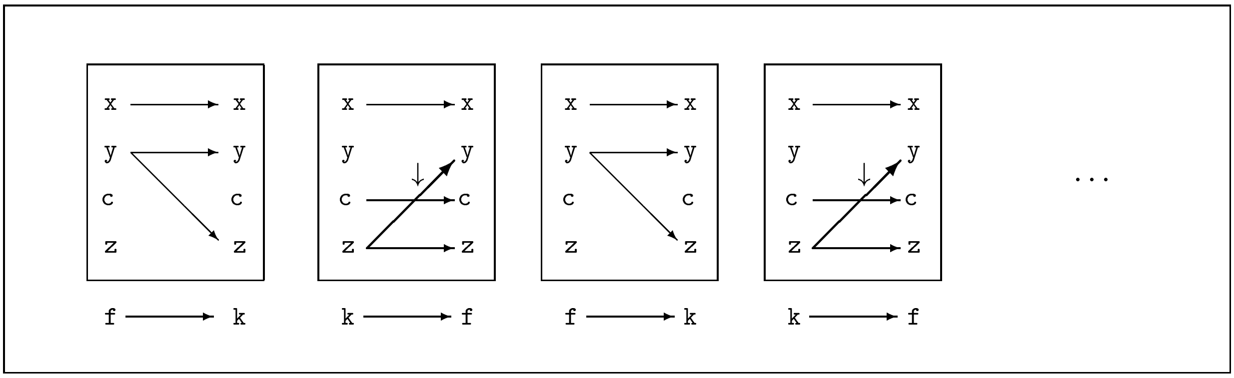


Figure 3.12: Multipath for Ex. 1

Further intuition for $\mathcal{M}$. We have already seen the pictorial representation of the set of abstract function-call transitions, $Tr(p)$, for a number of subject programs. Formally, a $Tr(p)$ multipath is simply a sequence of $Tr(p)$ elements such that the target function of each matches the source function of the next. However, the nagging question remains as to what a multipath $\mathcal{M}$ expresses with respect to the execution of the subject program. Let us address this issue briefly.

Consider again Ex. 1. If our machine implements simple static name-value binding, and execution runs unboundedly, then we expect to observe the execution stack represented in Fig. 3.13. In the diagram, an edge indicates that the top value is no greater in size than the lower value. If the edge is darkened, then the top value is strictly smaller in size.

Over time, the execution stack approximates (in some sense) the only multipath for the program starting with the initial function. (Refer to Fig. 3.12.) For the example, the arcs between `y` and `z` in the multipath's SRGs describe the size relations among parameters of successive activation records. Since the multipath has infinite descent, a sufficiently long execution would produce arbitrarily small sizes, which is impossible.

What about Ex. 2? Clearly, the size relation with source parameter `x` in the abstract function-call transition from `k` to `f` is not a size relation between parameters of successive activation records. The semantics of L in fact allows SRG analysis to track size relations more generally. The intuition is as follows. Consider multipath $\mathcal{M}$. Each abstract function-call transition of $\mathcal{M}$ corresponds to a pair of activation records in an execution stack. Call them the source and target records. If some element of $\mathcal{M}$ asserts a size relation $x \stackrel{r}{\rightarrow} y$, then the size relation is valid for the value of y in the corresponding target record, with the value of the *last-seen value of* x in a previous activation record. Thus, infinite descent in the multipath still means that any execution stack corresponding to it cannot grow indefinitely. There is only one infinite multipath for Ex. 2 starting with the initial function. It corresponds to the execution stack depicted in Fig. 3.14, which shows the descent reasoning graphically.

Recall that Ex. 4 has an infinite multipath without infinite descent. The multipath begins with the abstract function-call transition corresponding to expression `(f (tl x) (f (tl x)))`, followed by infinitely many repeats of the abstract function-call transition corresponding to expression `(k k)`. An execution stack corresponding to this multipath is depicted in Fig. 3.15, in which we have indicated the true size relations between each occurrence of `x` and the *initial* occurrence of `x`. This is the size relation that would manifest (incorrectly) as the arc $\mathtt{x} \stackrel{\downarrow}{\rightarrow} \mathtt{x}$ in the second SRG of Fig. 3.9, should ∇ be omitted from the definition of $\mathcal{G}$.

3.5.5 Size-change termination

Size-change termination was mentioned at the start of the SRG section as motivation for the SRG's definition. Let us conclude this chapter then by defining size-change termination formally as a property of $Tr(p)$.

Definition 3.5.23 A set of abstract function-call transitions Tr satisfies *size-change termination (SCT)* if every infinite Tr-multipath $f_0 \stackrel{G_1}{\rightarrow} f_1 \stackrel{G_2}{\rightarrow} f_2 \ldots$, where $f_0 = f_{initial}$, has infinite descent.

Theorem 3.5.24 $Tr(p)$ satisfies SCT implies that p is terminating.

Proof By Corollary 2.4.4, if p is non-terminating, there is an infinite $\leadsto$ chain: $\sigma_0; e_0 \leadsto \ldots \leadsto \sigma_t; e_t \leadsto \ldots$ such that σ_0 is an initial environment, and $e_0 \equiv e^{f_{initial}}$. This chain contains an infinite number of function-call transitions, since every non function-call transition decreases the size of the expression. Stating this formally, there exists an infinite sequence of functions $f_0, f_1, \ldots, f_t, \ldots$ and an infinite sequence of indices $i_0 < i_1 < \ldots < i_t < \ldots$ such that the following hold:

- $i_0 = 0$ and $f_0 = f_{initial}$;
- for $t \geq 0$, $e_{i_t} \equiv e^{f_t}$, and for $i \in (i_t, i_{t+1})$, e_i is a proper subexpression of e^{f_t};

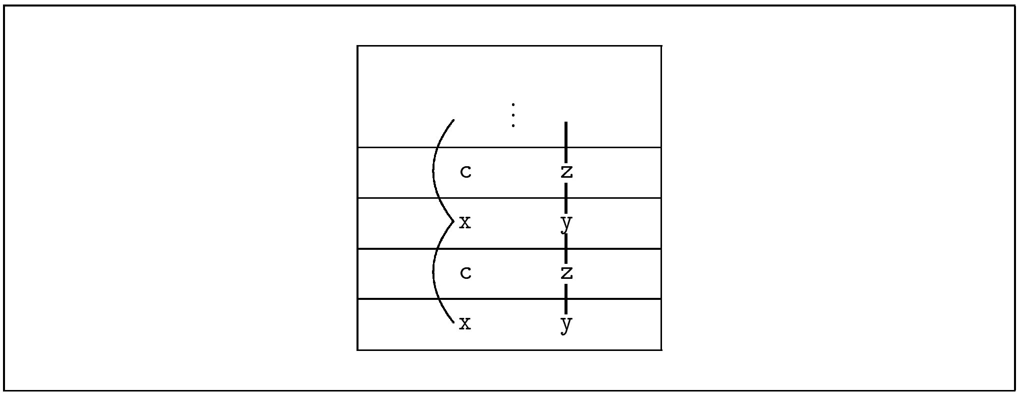


Figure 3.13: Execution stack for Ex. 1

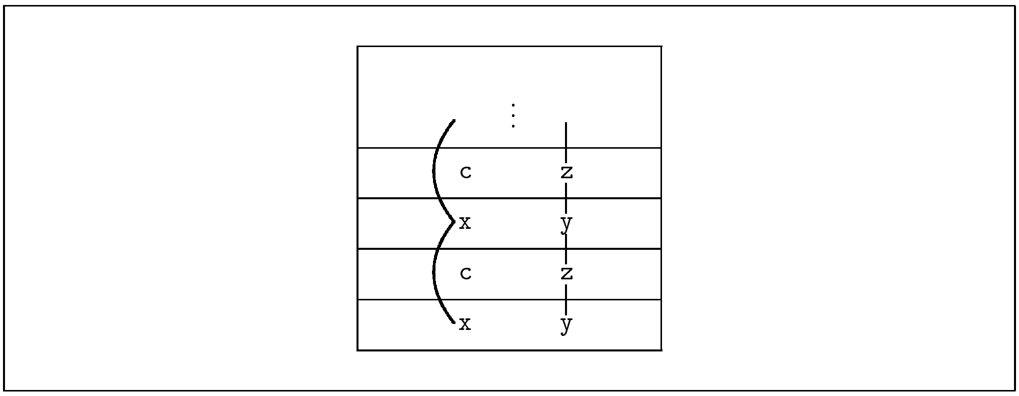
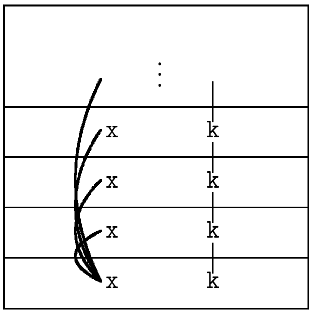


Figure 3.14: Execution stack for Ex. 2



After the initial call, all calls are due to `(k k)`.

Figure 3.15: Possible execution stack for Ex. 3

- for $t \geq 0$, σ_i are equal for all $i \in [i_t, i_{t+1})$; and,
- for $t > 0$, $\sigma_{i_t-1}; e_{i_t-1} \rightsquigarrow \sigma_{i_t}; e_{i_t}$ is a function-call transition to f_t.

By Corollary 3.5.21, for $t \geq 0$, there exists $f_t \stackrel{G_{t+1}}{\rightarrow} f_{t+1}$ in $Tr(p)$ such that G_{t+1} is safe for $(\sigma_{i_t}, \sigma_{i_{t+1}})$. By definition, $\mathcal{M} = f_0 \stackrel{G_1}{\rightarrow} f_1 \stackrel{G_2}{\rightarrow} f_2 \ldots$ is an infinite $Tr(p)$-multipath with $f_0 = f_{initial}$. By SCT, $\mathcal{M}$ has an infinitely descending thread $th = x_0 \stackrel{r_1}{\rightarrow} x_1 \stackrel{r_2}{\rightarrow} x_2 \ldots$ such that for some K, each $x_d \stackrel{r_{d+1}}{\rightarrow} x_{d+1}$ is in G_{K+d+1}. This gives rise to an infinite sequence of values corresponding to the parameters along *th*: $\sigma_{i_K}(x_0), \sigma_{i_{K+1}}(x_1), \ldots$. Now each consecutive pair of values obeys the size label on the corresponding arc of *th* (by definition of safety), thus $\sigma_{i_{K+d}}(x_d) > \sigma_{i_{K+d+1}}(x_{d+1})$ if $r_{d+1} = \downarrow$ and $\sigma_{i_{K+d}}(x_d) \geq \sigma_{i_{K+d+1}}(x_{d+1})$ if $r_{d+1} = \overline{\downarrow}$. Since $r_d = \downarrow$ for infinitely many d, we deduce that the value sequence exhibits infinite descent in the sizes of some data values, which violates the well-foundedness of $I\!N$.

Approximate SCT criteria. Any property of $Tr(p)$ that implies $Tr(p)$ satisfies SCT will be called an *(approximate) SCT criterion.* The next chapter is devoted to the study of SCT and its approximations. For each approximate criterion, we are interested in decision procedures (it is decidable whether $Tr(p)$ satisfies SCT!), complexity of the decision problem, and a characterization of the programs successfully classified as terminating using the criterion.

Chapter 4

Termination Criteria

4.1 The size-change termination criterion (SCT)

It is well-known that program termination is not Turing-decidable. In practice, the termination of many programs (program functions) is ensured by descent in the sizes of some data values over program state transitions. The SCT condition detects such descent in a general way, by tracking size decreases across different parameters non-deterministically.

SCT is a sufficient condition for program termination that captures a termination reasoning commonly used by programmers. Surprisingly, it can be decided mechanically. In this chapter, different ways to decide the SCT condition are described. The generality of the SCT approach is examined. The following is a noteworthy characteristic of the approach: Descent is not required in *every* function-call transition, so indirect recursion is handled naturally. Considering the issue more closely, we observe that SCT induces a well-founded ordering on the *evaluation states* such that every function-call transition is seen to decrease in this ordering. Such SCT orderings could reflect lexicographic descent, descent in a sum of parameter sizes, or more complicated descent. A series of examples demonstrates the forms of descent subsumed by SCT. They establish that SCT is very general for reasoning about program termination.

In the interest of applying SCT in practice, we consider the time complexity of its decision methods. However, it turns out that all our methods have exponential time complexity. This leads us to question the *complexity class* of the decision problem. We will prove in the next chapter that the SCT decision problem is in fact complete for PSPACE, so a polynomial-time algorithm to decide the condition is highly unlikely!

This result motivates the search for good approximate SCT criteria. The *in-situ descent criterion (ISD)* is related to Holst's variable boundedness analysis [Hol91], and appears to be a good candidate for an approximate criterion. It is shown to perform favourably in comparison with SCT in terms of generality. Unfortunately, ISD turns out to be PSPACE-hard.

We then derive a PTIME SCT approximation by observing that for size-relation graphs that can loop (i.e., they give rise to an infinite multipath), and that satisfy SCT, there is usually an "anchor point"—some graph whose infinite occurrence in a multipath causes infinite descent. The polynomial-time approximation is based on finding and eliminating such graphs, as far as possible. If the remaining graphs cannot loop, then SCT is established. From considering the complexity class alone, this approximation could be expected to be rather conservative with respect to SCT. However, we believe that it captures the "size-change reasoning" used in practice to ensure program termination. In particular, lexicographic descent, and descent in a sum of parameter sizes are subsumed. The PTIME criterion may be practical, because it targets the forms of descent that arise in practice *and* it can be decided efficiently.

Definition 4.1.1 For $f \in \mathit{Fun}$, f is *reachable in Tr* means that $f = f_{initial}$ or there exists a Tr-multipath of the form $f_0 \stackrel{G_1}{\to} f_1 \stackrel{G_2}{\to} f_2 \ldots \stackrel{G_{T+1}}{\to} f_{T+1}$, where $f_0 = f_{initial}$ and $f_{T+1} = f$.

For the remainder of this chapter, let Tr be the set of abstract function-call transitions in $\mathit{Tr}(p)$ whose source functions are reachable in $\mathit{Tr}(p)$. Then for each $f \stackrel{G'}{\to} f' \in \mathit{Tr}$, both f and f' are reachable in Tr. Clearly, Tr satisfies SCT if and only if $\mathit{Tr}(p)$ satisfies SCT.

4.1.1 Graph-based approach to decide SCT

The first criterion for termination that we investigate is the size-change criterion, defined at the end of the previous chapter. Given the definition of Tr, it can be restated as follows.

Criterion 4.1.2 (SCT) Every infinite Tr-multipath has infinite descent.

The criterion refers to infinite Tr-multipaths, which are difficult to work with directly. However, the Infinite Ramsey's Theorem [Ram30] implies that every infinite Tr-multipath has a tail that can be divided into consecutive sections, such that the abstract function-call transitions in each section compose to the *same* result (More about this later.) This motivates the definition of the compositional closure of Tr, which contains a Tr element corresponding to every *finite* Tr-multipath.

Definition 4.1.3 (Composition closure $\overline{Tr}$)

1. Define the composition of abstract function-call transitions in terms of SRG composition:
$$f \xrightarrow{G'} f'; f' \xrightarrow{G''} f'' = f \xrightarrow{G} f''$$
where $G = G'; G''$. The composition is undefined otherwise.

2. Define $\overline{Tr}$ as the closure of Tr under composition. Formally, $\overline{Tr}$ is the least fixedpoint of the following equation.
$$\overline{Tr} = Tr \cup \{a; a' \mid a, a' \in \overline{Tr}\}$$

Lemma 4.1.4 (Composition conserves thread information.) Let $\mathcal{M} = a_1, \ldots, a_T$. Then the composition $a_1; \ldots; a_T$ has SRG with an arc of the form $x \xrightarrow{\downarrow} y$ just when $\mathcal{M}$ has a complete thread from x to y with a $\downarrow$-labelled arc.

Proof is straightforward.

Definition 4.1.5 (Properties of Tr elements) Let $a = f \xrightarrow{G'} f' \in Tr$.

1. The element a is *idempotent* if $a; a = a$.

2. The element a has *in-situ descent* if $x \xrightarrow{\downarrow} x \in G'$ for some x.

Theorem 4.1.6 Tr satisfies SCT iff every idempotent a in $\overline{Tr}$ has in-situ descent.

Proof
($\Rightarrow$) Suppose there exists an idempotent $a \in \overline{Tr}$ whose SRG has no arc of the form $x \xrightarrow{\downarrow} x$. By the definition of $\overline{Tr}$, there is a finite Tr-multipath $\mathcal{M}'$, whose elements compose to a. Now, $\mathcal{M} = \mathcal{M}', \mathcal{M}', \ldots$ with $\mathcal{M}'$ repeated infinitely, is an infinite Tr-multipath.

For the purpose of proving a contradiction, suppose that $\mathcal{M}$ has a thread *th* of infinite descent. Consider the sequence of positions (parameters) of *th* at the start of each $\mathcal{M}'$. Some parameter x must occur infinitely in this sequence, since the set of parameters is finite. Thus, there is some thread from x to x containing a $\downarrow$-labelled arc in $\mathcal{M}', \ldots, \mathcal{M}'$, for some r repetitions of $\mathcal{M}'$. By Lemma 4.1.4, the composition a^r has SRG with the arc $x \xrightarrow{\downarrow} x$. By idempotence, $a^r = a$, so a has SRG with the arc $x \xrightarrow{\downarrow} x$, which contradicts the assumption about a. We conclude that the infinite Tr-multipath $\mathcal{M}$ has no infinite descent, so Tr does not satisfy SCT.

($\Leftarrow$) Conversely, suppose that all idempotent $a \in \overline{Tr}$ has SRG with an arc of the form $x \xrightarrow{\downarrow} x$. Consider any infinite Tr-multipath $\mathcal{M} = a_1, a_2, \ldots$. For $a \in \overline{Tr}$, define the property P_a for pairs of natural numbers as follows.
$$P_a = \{\{i, j\} \mid i < j,\ a_{i+1}; \ldots; a_j = a\}$$

Then $\{P_a \mid a \in \overline{Tr}\}$ is a finite set of non-overlapping properties covering every distinct pair of natural numbers. By the infinite Ramsey's Theorem [Ram30], there is an infinite set I of natural numbers $i_1 < i_2 < \ldots$, and property P_a, such that for every pair of distinct elements $i, j \in I$ with $i < j$, we have $\{i, j\} \in P_a$, i.e., $a_{i+1}; \ldots; a_j = a$. It follows that
$$\begin{array}{rcl} \forall i, j, k \in I \text{ with } i < j < k : a_{i+1}; \ldots; a_j; a_{j+1}; \ldots; a_k & = & a_{i+1}; \ldots; a_j \\ & = & a_{j+1}; \ldots; a_k \\ & = & a. \end{array}$$

By the associativity of composition, $(a_{i+1}; \ldots; a_j); (a_{j+1}; \ldots; a_k) = a; a = a$. Therefore a is idempotent.

By supposition, a has SRG with an arc of the form $x \stackrel{\downarrow}{\rightarrow} x$. By Lemma 4.1.4, for each k, $a_{i_k+1}, \ldots, a_{i_{k+1}}$ has a complete thread from x to x containing a $\downarrow$-labelled arc. Thus, the multipath $a_{i_1+1}, \ldots, a_{i_2}, a_{i_2+1}, \ldots$ has infinite descent in x, which implies that $\mathcal{M}$ has infinite descent. Since $\mathcal{M}$ is arbitrary, every infinite *Tr*-multipath has infinite descent, so *Tr* satisfies SCT.

A procedure for SCT. Theorem 4.1.6 leads at once to the following procedure to decide whether *Tr* satisfies SCT.

1. Compute $\overline{Tr}$ by a transitive closure procedure.
   - Include all of *Tr*.
   - For a and a' in $\overline{Tr}$, include also $a; a'$ in $\overline{Tr}$, if $a; a'$ is defined.
2. For each $f \stackrel{G}{\rightarrow} f$ in $\overline{Tr}$ such that $G = G; G$, test whether there exists $x \stackrel{\downarrow}{\rightarrow} x \in G$ for some x. (In practice, this test is conducted as $\overline{Tr}$ is generated.)

The procedure above has worst-case time-complexity $2^{O(N^2)}$, where N is the total number of parameters of all f referred to in *Tr*. The procedure computes $\overline{Tr}$, whose size is of this order.

A straightforward optimization.

Definition 4.1.7 (Strongly-connected subsets of *Tr*)

1. For $S \subseteq Tr$, define $Fun(S) = \{f, f' \mid f \stackrel{G}{\rightarrow} f' \in S\}$.
2. Call $S \subseteq Tr$ *strongly-connected* if S is non-empty and $Fun(S)$ is strongly-connected in the graph with arc set $\{(f, f') \mid f \stackrel{G}{\rightarrow} f' \in S\}$.
3. Define $SCC(Tr)$ to be the set of maximal strongly-connected subsets (i.e., *strongly connected components*) of *Tr*.

The following lemma shows that an optimization to the standard procedure is to compute the strongly-connected components of *Tr* (which can be done efficiently by an adaptation of the standard procedure [AHU75, CLR90]), and test whether each strongly-connected component S satisfies SCT. It may be reasonable to expect that each S has moderate size in practice.

Lemma 4.1.8 *Tr* satisfies SCT iff every $S \in SCC(Tr)$ satisfies SCT.

Proof If *Tr* satisfies SCT, then by definition every infinite *Tr*-multipath has infinite descent. In particular, for each $S \in SCC(Tr)$, every infinite S-multipath has infinite descent.

Conversely, suppose every strongly-connected component satisfies SCT. Let $\mathcal{M}$ be an infinite *Tr*-multipath. Then $\mathcal{M}$ has a tail $\mathcal{M}'$ containing only elements from $S = \{a \mid a \text{ occurs infinitely in } \mathcal{M}\}$. Now, S has to be strongly-connected, so it is the subset of some strongly-connected component S'. By assumption, S' satisfies SCT. Therefore $\mathcal{M}'$ has infinite descent. Hence, $\mathcal{M}$ has infinite descent. It follows that *Tr* satisfies SCT.

4.1.2 Automaton-based approach to decide SCT

Another approach to decide SCT uses the machinery of *ω-automata.* An *ω-regular language* is the analogue of a regular language, applied to infinite words. A regular language with finite alphabet Σ is a subset of Σ^* (the set of all words $a_1 \ldots a_n$ with $a_i \in \Sigma$) accepted by a finite state automaton (FSA). An ω-regular language with (finite) alphabet Σ is a subset of Σ^ω (the set of *infinite* words $a_1 a_2 \ldots$ with $a_i \in \Sigma$) accepted by a non-deterministic *Büchi automaton.* An elementary introduction to the Büchi automaton is given in the appendix.

The class of ω-regular languages is closed under union, intersection and complementation, much like regular languages. There also exist tests for emptiness and universality. In particular the following is true.

Lemma 4.1.9 Given Büchi automata for accepting L and L', it can be decided in PSPACE whether or not $L \subseteq L'$.

Proof See, for example, [Var96].

The problem is PSPACE-hard. The source of complexity is in the problem of complementation [McN66, SVW87, EJ89, Saf88].

Lemma 4.1.10 Label each element of *Tr* distinctly, and let the set of labels be Σ. Write $tr(a)$ for the abstract function-call transition labelled by $a \in \Sigma$. Define two sets, *FLOW* and *DESC*, as follows.

$$\begin{array}{lcl} \mathit{FLOW} & = & \{a_1 a_2 \ldots \mid tr(a_1), tr(a_2), \ldots \text{ is an infinite } \mathit{Tr}\text{-multipath}\} \\ \mathit{DESC} & = & \{a_1 a_2 \ldots \mid tr(a_t), tr(a_{t+1}), \ldots \text{ has a complete thread with infinite descent}\} \end{array}$$

Then $\mathit{FLOW} \subseteq \mathit{DESC}$ iff *Tr* satisfies SCT.

Proof is immediate from the definition of SCT.

Lemma 4.1.11 Let *FLOW* and *DESC* be defined as above. Then both sets are ω-regular.

Proof It turns out to be quite easy to define Büchi automata to accept *FLOW* and *DESC*. See appendix A.2 for details of the construction.

The proof of the last lemma is constructive. By Lemma 4.1.9, it is decidable whether *Tr* satisfies SCT. An inspection of the approach shows that given *Tr*, this can be done in PSPACE.

An algorithm based on [Saf88] can decide $\mathit{FLOW} \subseteq \mathit{DESC}$ with time-complexity $2^{O(N \log N)}$, where N is the total number of parameters of all f referred to in *Tr*. This is better than the graph-based approach of the last section. However, the algorithm of [Saf88] is complicated, and it is unclear whether it will lead to a better solution in practice.

4.1.3 Generality of SCT

Let us try to gauge the generality of the SCT criterion by working through a series of first-order examples handled by it. The numbering of these examples follow from the examples of the last chapter.

Example 8. Simple in-situ descent.

```
rev a   = if (a = nil) nil
             (¹app (²rev (tl a)) [a])

app x y = if (x = nil) y
             (hd x) : (³app (tl x) y)
```

The abstract function-call transitions for callsites 1, 2 and 3 are respectively: $\texttt{rev} \xrightarrow{G} \texttt{app}$ for some G, $\texttt{rev} \xrightarrow{G'} \texttt{rev}$ where G' includes $\{\texttt{a} \xrightarrow{\downarrow} \texttt{a}\}$, and $\texttt{app} \xrightarrow{G''} \texttt{app}$, where G'' includes $\{\texttt{x} \xrightarrow{\downarrow} \texttt{x}, \texttt{y} \xrightarrow{\overline{\overline{\downarrow}}} \texttt{y}\}$. Let $\mathcal{M}$ be any infinite *Tr*-multipath. Then $\mathcal{M}$ ends with infinitely many $\texttt{rev} \xrightarrow{G'} \texttt{rev}$ or infinitely many $\texttt{app} \xrightarrow{G''} \texttt{app}$. There is infinite descent in `a` and `x` respectively.

Example 9. Indirect recursion.

```
f i x   = if ... (¹g (i ⊖ 1) (x + 1) (i + 1))
                 (²f (i ⊖ 1) (x + 1))

g j y z = if ... (³f j (y + z))
                 (⁴g (j ⊖ 1) (y + 1) (z + 1))
```

This example illustrates that indirect recursion is not an issue. The set of abstract function-call transitions includes: for callsite 1, $\mathtt{f} \xrightarrow{G} \mathtt{g}$, where G includes $\{\mathtt{i} \xrightarrow{\downarrow} \mathtt{j}\}$, for callsite 2, $\mathtt{f} \xrightarrow{G'} \mathtt{f}$, where G' includes $\{\mathtt{i} \xrightarrow{\downarrow} \mathtt{i}\}$, for callsite 3, $\mathtt{g} \xrightarrow{G''} \mathtt{f}$, where G'' includes $\{\mathtt{j} \xrightarrow{\overset{=}{\downarrow}} \mathtt{i}\}$, and for callsite 4, $\mathtt{g} \xrightarrow{G'''} \mathtt{g}$, where G''' includes $\{\mathtt{j} \xrightarrow{\downarrow} \mathtt{j}\}$. Let $\mathcal{M}$ be any infinite *Tr*-multipath. If the tail of $\mathcal{M}$ has infinitely many $\mathtt{f} \xrightarrow{G} \mathtt{g}$, then it can be divided into consecutive sections, such that each section begins with $\mathtt{f} \xrightarrow{G} \mathtt{g}$ followed by any number of $\mathtt{g} \xrightarrow{G'''} \mathtt{g}$, followed by $\mathtt{g} \xrightarrow{G''} \mathtt{f}$, followed by any number of $\mathtt{f} \xrightarrow{G'} \mathtt{f}$. By induction, it can be proved that each section has descent in `i`. Thus, $\mathcal{M}$ has infinite descent in `i`. If $\mathcal{M}$ has infinitely many $\mathtt{g} \xrightarrow{G''} \mathtt{f}$, argue similarly that it has infinite descent in `j`. Otherwise, $\mathcal{M}$ ends in infinitely many $\mathtt{f} \xrightarrow{G'} \mathtt{f}$ or infinitely many $\mathtt{g} \xrightarrow{G'''} \mathtt{g}$. There is infinite descent in `i` and `j` respectively.

Example 10. Lexical descent in a pair of parameter sizes.

```
ack m n = if (m = 0) (n + 1)
              if (n = 0) (^1ack (m ⊖ 1) 1)
                 (^2ack (m ⊖ 1) (^3ack m (n ⊖ 1)))
```

The Ackermann function is extremely fast-growing, and is normally used to demonstrate how certain computations are intractable. The point of this example though is simply to illustrate descent of a pair of parameter sizes in a lexical ordering, for any function-call transition.

The abstract function-call transitions are: for callsites 1 and 2, $\mathtt{ack} \xrightarrow{G} \mathtt{ack}$, where $G = \{\mathtt{m} \xrightarrow{\downarrow} \mathtt{m}\}$ and for callsite 3, $\mathtt{ack} \xrightarrow{G'} \mathtt{ack}$, where $G' = \{\mathtt{m} \xrightarrow{\overset{=}{\downarrow}} \mathtt{m}, \mathtt{n} \xrightarrow{\downarrow} \mathtt{n}\}$. To see that the abstract function-call transitions satisfy SCT, argue as follows. If an infinite *Tr*-multipath $\mathcal{M}$ has infinitely many occurrences of $\mathtt{ack} \xrightarrow{G} \mathtt{ack}$, then there is infinite descent in `m`. Otherwise, $\mathcal{M}$ ends in infinitely many $\mathtt{ack} \xrightarrow{G'} \mathtt{ack}$, in which case there is infinite descent in `n`.

Example 11. Descent in a sum of parameter sizes.

```
p m n r = if ... (^1p m (tl r) n)
                 (^2p r (tl n) m)
```

This example illustrates "permuted arguments," which is the same behaviour observed in the usual algorithm for computing the greatest common divisor of two positive numbers. The abstract function-call transitions for this example are: for callsite 1, $\mathtt{p} \xrightarrow{G} \mathtt{p}$, where $G = \{\mathtt{m} \xrightarrow{\overset{=}{\downarrow}} \mathtt{m}, \mathtt{n} \xrightarrow{\overset{=}{\downarrow}} \mathtt{r}, \mathtt{r} \xrightarrow{\downarrow} \mathtt{n}\}$, and for callsite 2, $\mathtt{p} \xrightarrow{G'} \mathtt{p}$, where $G' = \{\mathtt{m} \xrightarrow{\overset{=}{\downarrow}} \mathtt{r}, \mathtt{n} \xrightarrow{\downarrow} \mathtt{n}, \mathtt{r} \xrightarrow{\overset{=}{\downarrow}} \mathtt{m}\}$. In any infinite *Tr*-multipath $\mathcal{M}$, there are precisely three maximal threads. An infinite number of $\downarrow$-labelled arcs must occur among these threads, thus $\mathcal{M}$ has a least one thread with infinite descent. This example is usually proved terminating by appealing to descent in the number $sx + sy + sy$, where sx, sy and sz are the sizes of `m`, `n` and `r`'s values respectively.

Example 12. Descent in the maximum of parameter sizes.

```
f x y = if ... (^1f (hd x) (tl x))
               (^2f (hd y) (tl y))
```

This descent has been observed in a type inference algorithm. The abstract function-call transitions for the example are: for callsite 1, $\mathtt{f} \xrightarrow{G} \mathtt{f}$, where $G = \{\mathtt{x} \xrightarrow{\downarrow} \mathtt{x}, \mathtt{x} \xrightarrow{\downarrow} \mathtt{y}\}$ and for callsite 2, $\mathtt{f} \xrightarrow{G'} \mathtt{f}$, where $G' = \{\mathtt{y} \xrightarrow{\downarrow} \mathtt{x}, \mathtt{y} \xrightarrow{\downarrow} \mathtt{y}\}$. By induction, it can be proved that any *Tr*-multipath starting with $\mathtt{f} \xrightarrow{G} \mathtt{f}$ has a thread from `x` whose number of $\downarrow$-labelled arcs is the same as the length of the multipath. For a multipath starting with $\mathtt{f} \xrightarrow{G'} \mathtt{f}$, there is a thread from `y` whose number of $\downarrow$-labelled arcs is the same as the length of the multipath. Thus, every infinite *Tr*-multipath has infinite descent. This example is usually proved terminating by appealing to descent in the number $max(sx, sy)$, where sx and sy are the sizes of `x` and `y`'s values respectively.

Ex.	SCT	Terminating	Description
1	Yes	Yes	High-order example with obvious descent.
2	Yes	Yes	High-order example with less obvious descent.
3	Yes	Yes	As example 2. Program truncates lazy trees.
4	No	No	`f`-closure passes into `f`-environment.
5	*No*	*Yes*	`f`*-closure passes into* `f`*-environment.*
6	Yes	Yes	Factorial by Y combinator.
7	Yes	Yes	Simulating let binding.
8	Yes	Yes	Simple in-situ descent. Program reverses lists.
9	Yes	Yes	Simple in-situ descent with indirect recursion.
10	Yes	Yes	Lexical descent. The Ackermann function.
11	Yes	Yes	Descent in sum of parameter sizes.
12	Yes	Yes	Descent in maximum of parameter sizes.
13	Yes	Yes	Permuted and discarded parameters.
14	Yes	Yes	Insertion sort.
15	Yes	Yes	Quicksort.

Figure 4.1: Examples handled by SCT

Example 13. Unusual descent.

```
f x y = if ... (¹f y (tl y))
               (²f y (tl x))
```

This example is a variation on the "permuted argument" example. It does not necessarily continue every thread, for example, during the call `(f y (tl y))`. The abstract function-call transitions are: for callsite 1, $\mathtt{f} \xrightarrow{G} \mathtt{f}$, where $G = \{\mathtt{y} \stackrel{\overline{\downarrow}}{\to} \mathtt{x}, \mathtt{y} \stackrel{\downarrow}{\to} \mathtt{y}\}$, and for callsite 2, $\mathtt{f} \xrightarrow{G'} \mathtt{f}$, where $G' = \{\mathtt{x} \stackrel{\downarrow}{\to} \mathtt{y}, \mathtt{y} \stackrel{\overline{\downarrow}}{\to} \mathtt{x}\}$. Argue that *Tr* satisfies SCT as follows. Let $\mathcal{M}$ be an infinite *Tr*-multipath. If $\mathtt{f} \xrightarrow{G} \mathtt{f}$ occurs infinitely in $\mathcal{M}$, then divide $\mathcal{M}$ into consecutive sections, such that each section consists of a single $\mathtt{f} \xrightarrow{G} \mathtt{f}$ followed by any number of $\mathtt{f} \xrightarrow{G'} \mathtt{f}$. Such a segment always has descent in `y` by induction on the number of $\mathtt{f} \xrightarrow{G'} \mathtt{f}$. Thus there is infinite descent in `y`. If $\mathtt{f} \xrightarrow{G} \mathtt{f}$ does not occur infinitely in $\mathcal{M}$, then $\mathcal{M}$ ends in infinitely many $\mathtt{f} \xrightarrow{G'} \mathtt{f}$, in which case there is infinite descent in both `x` and `y`. This example is interesting because it is connected to the proof of PSPACE-hardness for SCT.

All examples, at a glance. The table in Fig. 4.1 summarizes the examples considered so far. The minsort and quicksort programs have not been discussed in detail. However, their *Tr* sets exhibit only simple in-situ descent. The difficulty in handling them lies in the size analysis.

4.1.4 Complexity of SCT

It is worrying that the methods available for deciding SCT are EXPTIME and PSPACE. This leads us to question whether the SCT decision problem perhaps has high intrinsic complexity. The author first observed a simple reduction from the 3SAT problem to the non-satisfiability of SCT. This established that SCT is CONP-hard. Subsequently, in joint work with Jones and Ben-Amram [LJBA01], it was proved that SCT is in fact PSPACE-hard. Since SCT can be decided in PSPACE, SCT is complete for PSPACE.

The proof of PSPACE-hardness is by reduction from boolean program termination. As the details are rather involved, the proof is deferred until the next chapter. We mention, however, that as a corollary, the Prolog termination analyses Termilog [LSS97] and Terminweb [CT97] are also known to be PSPACE-hard. Our complexity result motivates the search for good approximate SCT criteria, which we now consider.

4.2 The in-situ descent criterion (ISD)

The *in-situ descent criterion (ISD)* is inspired by Holst's analysis of variable boundedness [Hol91]. There are two reasons for studying the ISD criterion. First, any result about ISD might be applicable to Holst's analysis, or at least impart insight. Second, the ISD criterion appears to be an effective approximation of SCT.

Definition 4.2.1 A *Tr*-loop is a *Tr*-multipath of the form $f \stackrel{G'}{\to} f' \ldots \stackrel{G}{\to} f$.

Criterion 4.2.2 (ISD) Every *Tr*-loop has a complete thread from x to x with a $\downarrow$ arc, for some x.

Corollary 4.2.3 *Tr* satisfies ISD iff for every *Tr*-loop, $\mathcal{M} = f \stackrel{G'}{\to} f' \ldots \stackrel{G}{\to} f$, the composition of the abstract function-call transitions of $\mathcal{M}$ has SRG with an arc of the form $x \stackrel{\downarrow}{\to} x$.

Proof The corollary follows immediately from the definition of ISD and Lemma 4.1.4.

4.2.1 Graph-based approach to decide ISD

The following theorem provides one way to decide whether any given *Tr* satisfies ISD.

Theorem 4.2.4 *Tr* satisfies ISD iff for every $f \stackrel{G}{\to} f \in \overline{Tr}$ has in-situ descent.

Proof
($\Rightarrow$) Suppose that *Tr* satisfies ISD, and let $f \stackrel{G}{\to} f \in \overline{Tr}$. By definition of $\overline{Tr}$, $f \stackrel{G}{\to} f$ corresponds to the composition of abstract function-call transitions in a *Tr*-loop. By Corollary 4.2.3, G has an arc of the form $x \stackrel{\downarrow}{\to} x$.

($\Leftarrow$) Suppose for each $f \stackrel{G}{\to} f \in \overline{Tr}$, G has an arc of form $x \stackrel{\downarrow}{\to} x$. Let $\mathcal{M}$ be a *Tr*-loop. By definition of $\overline{Tr}$, the composition of the abstract function-call transitions of $\mathcal{M}$, which has the form $f \stackrel{G^\star}{\to} f$, occurs in $\overline{Tr}$. By assumption, $G^\star$ has an arc of the form $x \stackrel{\downarrow}{\to} x$. By Corollary 4.2.3, *Tr* satisfies ISD.

Corollary 4.2.5 *Tr* satisfies ISD implies that *Tr* satisfies SCT.

Proof By Theorem 4.2.4, *Tr* satisfies ISD implies that *every* $f \stackrel{G}{\to} f \in \overline{Tr}$ has in-situ descent. In particular, every idempotent $f \stackrel{G}{\to} f \in \overline{Tr}$ has in-situ descent. By Theorem 4.1.6, *Tr* satisfies SCT.

4.2.2 Automaton-based approach to decide ISD

Another approach to decide ISD uses automata, by analogy with SCT. However, for ISD, finite state automata are sufficient.

Lemma 4.2.6 Label each element of *Tr* distinctly, and let the set of labels be Σ. Write $tr(a)$ for the abstract function-call transition labelled by $a \in \Sigma$. Define two sets *LOOP* and *DOWN* as follows.

$$\begin{array}{lcl} LOOP & = & \{a_1, \ldots, a_T \mid tr(a_1), \ldots, tr(a_T) \text{ is a } Tr\text{-loop}\} \\ DOWN & = & \{a_1, \ldots, a_T \mid tr(a_1), \ldots, tr(a_n) \\ & & \quad \text{has a complete thread from } x \text{ to } x \text{ with a } \downarrow\text{-labelled arc}\} \end{array}$$

Then, $LOOP \subseteq DOWN$ iff *Tr* satisfies ISD.

Proof is immediate from the definition of ISD.

Lemma 4.2.7 Let *LOOP* and *DOWN* be as above. Both sets are regular languages of Σ.

Proof It is easy to define FSAs to accept *LOOP* and *DOWN*. See appendix A.1 for details.

From FSA theory, given automata for accepting *LOOP* and *DOWN*, it can be decided in PSPACE whether $LOOP \subseteq DOWN$, and hence whether *Tr* satisfies ISD.

Ex.	SCT	ISD	Terminating	Description
1	Yes	Yes	Yes	High-order example with obvious descent.
2	Yes	Yes	Yes	High-order example with less obvious descent.
3	Yes	Yes	Yes	As example 2. Program truncates lazy trees.
4	No	No	No	`f`-closure passes into `f`-environment.
5	No	No	Yes	`f`-closure passes into `f`-environment.
6	Yes	Yes	Yes	Factorial by Y combinator.
7	Yes	Yes	Yes	Simulating let binding.
8	Yes	Yes	Yes	Simple in-situ descent. Program reverses lists.
9	Yes	Yes	Yes	Simple in-situ descent with indirect recursion.
10	Yes	Yes	Yes	Lexical descent. The Ackermann function.
11	*Yes*	*No*	*Yes*	*Descent in sum of parameter sizes.*
12	Yes	Yes	Yes	Descent in maximum of parameter sizes.
13	*Yes*	*No*	*Yes*	*Permuted and discarded parameters.*
14	Yes	Yes	Yes	Insertion sort.
15	Yes	Yes	Yes	Quicksort.

Figure 4.2: Examples handled by ISD

4.2.3 Generality of ISD

Figure 4.2 shows the results of applying ISD to the examples considered previously for SCT. Note that the "permuted argument" example (Ex. 11), which shows a sum of parameter sizes decreasing in every function-call transition, is *not* handled by ISD. To see this, simply observe that the abstract function-call transition for the call `(p m (tl r) n)`, which is $\mathtt{p} \stackrel{G}{\to} \mathtt{p}$ where $G = \{\mathtt{m} \stackrel{\overline{\downarrow}}{\to} \mathtt{m}, \mathtt{n} \stackrel{\overline{\downarrow}}{\to} \mathtt{r}, \mathtt{r} \stackrel{\downarrow}{\to} \mathtt{n}\}$, has no in-situ descent. By Theorem 4.2.4, the graphs for the example do not satisfy ISD. This example highlights a common form of descent missed by Holst's conditions for variable boundedness [Hol91, AH96].

ISD allows the "descent parameter" to vary for different regular subsets of *Tr*-loops. This turns out to be a source of complexity for the criterion. In practice, the ability to consider different descent parameters for different regular subsets of *Tr*-loops is only used to handle lexical descent (as exhibited by Ex. 10).

4.2.4 Complexity of ISD

The closeness of the decision methods for SCT and ISD should have alerted us that the ISD criterion might be intractable. In the next chapter, we will prove that ISD, like SCT, is PSPACE-hard to decide.

All our effort to develop and investigate the ISD criterion has not been in vain. The proof of PSPACE-hardness for ISD leads to a proof of PSPACE-hardness for Holst's variable boundedness analysis [Hol91], refuting a conjecture in [AH96] that the analysis is not "up against intrinsic complexity problems."

4.3 The size-change polytime criterion (SCP)

The search for a polytime approximation to SCT continues! This section investigates a good candidate criterion satisfying our requirements, namely, the criterion should be sufficiently general for practical termination analysis, and it should be decidable efficiently.

Intuitive motivation. Refer to an infinite multipath without infinite descent as *viable*. Thus *Tr* satisfies SCT if and only if every infinite *Tr*-multipath is non-viable. An observation: For *Tr* that contains a loop and satisfies SCT, it is usually easy to find in the set a "progress point"— some abstract function-call transition whose infinite occurrence in a multipath causes infinite descent. For the Ackermann example (Ex. 10), the abstract function-call transition $\mathtt{ack} \stackrel{G}{\to} \mathtt{ack}$, where $G = \{\mathtt{m} \stackrel{\downarrow}{\to} \mathtt{m}\}$, is such a progress point.

By definition, a progress point cannot occur infinitely in a viable multipath. Eliminating a progress point from *Tr* typically unveils another one (corresponding to a nested computation). For Ex. 10, removing $\texttt{ack} \overset{G}{\to} \texttt{ack}$ leaves $\{\texttt{ack} \overset{G'}{\to} \texttt{ack}\}$, where $G' = \{\texttt{m} \overset{\overline{\downarrow}}{\to} \texttt{m}, \texttt{n} \overset{\downarrow}{\to} \texttt{n}\}$. The abstract function-call transition $\texttt{ack} \overset{G'}{\to} \texttt{ack}$ is now a progress point.

A polytime approximation of SCT can be based on finding and removing progress points, as far as possible. If the elements that remain do not contain a loop, we deduce that no element can appear infinitely in a viable multipath. Therefore no viable multipath exists. It follows that *Tr* satisfies SCT.

Different methods can be devised to spot progress points, based on observing the thread behaviour at particular parameters for each function f_t of a multipath $f_0 \overset{G_1}{\to} f_1 \overset{G_2}{\to} f_2 \ldots$. With sufficiently strong methods for deducing thread behaviour, even Ex. 13 can be handled. For this example, every *finite* multipath clearly has a complete thread in `x` and `y`. By König's Lemma, any infinite multipath has an infinite thread in `x` and `y`. However, without the $\downarrow$-labelled arcs, no multipath has any infinite thread. We deduce that every infinite multipath has a thread with infinite descent. For simplicity, we will present an intuitive, but less powerful method for determining progress points, geared towards descent in a sum of parameter sizes (identified as an important form of descent in [Plü90]).

For $S \subseteq \mathit{Tr}$, the set $\mathit{TP}(S)$ of "thread-preserving parameters" consists of pairs $f \cdot x \in \mathit{Fun} \times \mathit{Var}$ such that any thread starting from x at f in an S-multipath is never "lost." For instance, in Ex. 1, the thread between $\texttt{f} \cdot \texttt{y}$ and $\texttt{k} \cdot \texttt{z}$ is never lost, or in Ex. 2, the thread between $\texttt{f} \cdot \texttt{x}$ and $\texttt{k} \cdot \texttt{x}$ is never lost. (Consult Fig. 3.6 and 3.7 on page 44.)

Definition 4.3.1 (TP-parameter) Let $S \subseteq \mathit{Tr}$ be strongly-connected.

1. Define $\mathit{TP}(S)$ as the greatest fixedpoint of the following equation.

$$\mathit{TP}(S) = \{f \cdot x \mid \forall f \overset{G'}{\to} f' \in S : \exists x \overset{r}{\to} y \in G' : f' \cdot y \in \mathit{TP}(S)\}$$

   Each element of $\mathit{TP}(S)$ is called a *TP-parameter* or *TP-pair* of S.

2. For $f \overset{G'}{\to} f' \in S$, $x \overset{r}{\to} y \in G$ such that $f \cdot x, f' \cdot y \in \mathit{TP}(S)$, call $x \overset{r}{\to} y$ a *TP-arc with respect to S*. A (maximal) connected sequence of TP-arcs with respect to S is a *(maximal) TP-thread with respect to S*. A TP-thread extending over the entire length of a multipath is a *complete TP-thread*.

3. Define $\mathit{TP}(S)|_f = \{x \mid f \cdot x \in \mathit{TP}(S)\}$ to be the *focus set for f with respect to S*.

The role of $\mathit{TP}(S)$ for deducing progress points is now briefly explained. Let S be a strongly-connected component of the abstract function-call transitions under consideration. For *fan-in free* S (i.e., the SRGs of S elements are fan-in free), it is easily shown that all $\mathit{TP}(S)|_f$ for $f \in \mathit{Fun}(S)$ have the same size. We deduce that any multipath has a finite number of maximal TP-threads with respect to S. If some $a \in S$ has SRG containing a $\downarrow$-labelled TP-arc with respect to S, then a is clearly a progress point, because in any multipath featuring a infinitely, the TP-arc must appear infinitely among a finite number of threads, which implies that the multipath has a thread with infinite descent.

Important assumption: No fan-ins. It is henceforth assumed that each abstract function-call transition of *Tr* has SRG that is fan-in free. Any SRG produced by the size analysis $\mathcal{S}$ is fan-in free because the result of an expression is either deemed to have size no greater than, *or* strictly less than a particular variable's value. Otherwise, no size information is available for that expression. We do not accurately model, for instance, the result of `(min x y)`, where `min` computes the minimum of two natural number arguments.

Fan-ins are not natural when the size analysis does not take into account the conditions of `if`-constructs. If the conditions are considered, "lateral size relations" (size relations among parameter values in a single environment) can be used to record information like: $\#\sigma(x_1) \geq \#\sigma(x_2)$ holds for any environment σ that gives rise to a function-call transition $\sigma; e \rightsquigarrow \sigma'; e'$ at a particular program point. Then if it is known that $\#\sigma(x_2) \geq \#\sigma'(x_3)$ for any such transition, it can be deduced that $\#\sigma(x_1) \geq \#\sigma'(x_3)$. This induces a fan-in in the SRG of the corresponding abstract function-call transition.

```
fun W(S') // Computes the SCP "witnesses" among S'
    return {f -G'-> f' ∈ S' | G' has a decreasing TP-arc wrt S'}
end fun

fun SCP(S)
    return ⋀_{S'∈SCC(S)} if W(S') is empty then False else SCP(S' − W(S'))
end fun
```

Figure 4.3: Algorithm for deciding SCP

SCP, defined.

Criterion 4.3.2 (SCP) For *Tr* whose abstract function-call transitions have fan-in free SRGs, the SCP criterion is: For every strongly-connected $S \subseteq Tr$, some $a \in S$ has SRG containing a $\downarrow$-labelled TP-arc with respect to S.

Theorem 4.3.3 (Correctness of SCP) Suppose for each $f \xrightarrow{G'} f' \in Tr$, G' is fan-in-free. Then *Tr* satisfies SCP implies that *Tr* satisfies SCT.

Proof Let $S \subseteq Tr$ be strongly-connected. And let $\mathcal{M}'$ be any S-multipath. We first argue that $\mathcal{M}'$ has finitely many maximal TP-threads with respect to S. *Claim*: The cardinality of each $TP(S)|_f$ is the same for each $f \in Fun(S)$. Otherwise, there exists $f, f' \in Fun(S)$ such that $\| TP(S)|_f \| > \| TP(S)|_{f'} \|$. Since S is strongly-connected, there is an S-multipath from f to f'. This has the form $f \xrightarrow{G_1} f_1 \ldots \xrightarrow{G'} f'$. By the definition of $TP(S)$, each $x \in TP(S)|_f$ is connected to some y in G_1 such that $f_1 \cdot y \in TP(S)|_{f_1}$. By the assumption that G_1 is fan-in free, $TP(S)|_{f_1}$ is not smaller than $TP(S)|_f$ in size. Continuing inductively, we have $\| TP(S)|_f \| \leq \| TP(S)|_{f'} \|$, which is a contradiction. So each $TP(S)|_f$ for $f \in Fun(S)$ has the same cardinality K. In a sense, $TP(S)$ has selected, for each $f \in Fun(S)$, a set of K parameters for consideration. We deduce that for each $f \xrightarrow{G'} f' \in S$, there are exactly K TP-arcs in G'. These arcs form K non-overlapping maximal TP-threads in any S-multipath.

To continue with the proof, suppose that *Tr* satisfies SCP. Consider any infinite *Tr*-multipath $\mathcal{M}$. We will argue that $\mathcal{M}$ has infinite descent, which shows that *Tr* satisfies SCT. Now, a tail of $\mathcal{M}$, call it $\mathcal{M}'$, contains only elements from the set

$$S = \{a \in Tr \mid a \text{ occurs infinitely in } \mathcal{M}\}.$$

As S is strongly-connected, by SCP, there exists some $f \xrightarrow{G'} f' \in S$ such that the SRG G' contains a decreasing TP-arc $x \xrightarrow{\downarrow} y$. Since $f \xrightarrow{G'} f'$ appears infinitely often in $\mathcal{M}'$, so does the TP-arc $x \xrightarrow{\downarrow} y$. We have established that there are only finitely many maximal TP-threads in any S-multipath, so some maximal TP-thread of $\mathcal{M}'$ must contain infinitely many occurrences of $x \xrightarrow{\downarrow} y$. That is, $\mathcal{M}'$ has a thread with infinite descent. Since $\mathcal{M}'$ is a tail of $\mathcal{M}$, it follows that $\mathcal{M}$ has infinite descent.

4.3.1 Deciding SCP in polynomial time

Given *Tr* whose abstract function-call transitions have fan-in free SRGs, we can decide whether it satisfies SCP by computing $SCP(Tr)$, using the recursive procedure of Fig. 4.3.1. The auxiliary $W(S')$ computes the set of "witnesses" of $\downarrow$-labelled TP-arc among S'. $SCP(S)$ searches for a strongly-connected subset $S^\star$ of S *not* containing any $f \xrightarrow{G'} f'$ such that G' has a $\downarrow$-labelled TP-arc. Call $S^\star$ *problematic*: $S^\star$ refutes SCP and is potentially (i.e., is not proved not to be) the infinity set of a viable *Tr*-multipath.

For $S \subseteq Tr$, if S has no strongly-connected components, it satisfies SCP vacuously. The SCP procedure (its conjunct $\bigwedge\!\!\!\bigwedge$) returns *True* in this case. If strongly-connected components exist, the SCP procedure considers each one in turn. If a component S' is found containing no element $f \overset{G'}{\to} f'$ such that G' has a $\downarrow$-labelled TP-arc (so $W(S')$ is empty), then *SCP* returns *False*, since a problematic subset of *Tr* has been located. Otherwise, a subset $S^\star$ for refuting SCP may be found by inspecting each strongly-connected component of S (if it is included in S at all). We need only look among strongly-connected components because $S^\star$ is required to be strongly-connected. Later, we show that the TP-parameter set expands as the argument set decreases. It is therefore possible, when considering component S', to exclude any $a \in W(S')$: We know that a cannot appear in any problematic subset $S^\star$, as its occurrence would cause $S^\star$ to inherit a $\downarrow$-labelled TP-arc. In short, problematic subsets may be sought among $S' - W(S')$, for $S' \in SCC(S)$. This explains the recursion in the SCP procedure.

Lemma 4.3.4 Let $S', S'' \subseteq Tr$ be strongly-connected. Then,

$$S'' \subseteq S' \Rightarrow TP(S') \subseteq TP(S'')$$

Proof Any solution of the fixedpoint equation for $TP(S')$ is a solution of the equation for $TP(S'')$. By definition, $TP(S'')$ is the greatest among the solutions of its defining equation.

Theorem 4.3.5 $SCP(S)$ returns *True* if S satisfies SCP, *False* otherwise.

Proof First, note that any invocation of *SCP* terminates, since the argument S is reduced for every recursive invocation. If it returns *False*, a problematic subset $S^\star$ of S has been found, which contains no $f \overset{G'}{\to} f'$ such that G' has a $\downarrow$-labelled TP-arc wrt $S^\star$. By definition, S does not satisfy SCP.

Conversely, suppose S does not satisfy SCP. Then there exists some problematic subset $S^\star$ of S. We prove that the *SCP* procedure returns *False* in this case, by induction on the value of $d = \|S \setminus S^\star\|$.

- If $d = 0$, then $S = S^\star$ is a strongly-connected component. In this case, $SCC(S) = \{S\}$, and the set of witnesses $W(S)$ evaluates to $\{\}$, so $SCP(S)$ evaluates to *False*.

- Let $d > 0$. The induction hypothesis is: For $S'' \subseteq S$ such that $S^\star \subseteq S''$, if $\|S'' \setminus S^\star\| < d$, then $SCP(S'')$ evaluates to *False*. The set $S^\star$ is strongly-connected, so it has to be a subset of a strongly-connected component of S. Let $S^\star \subseteq S'$ for some $S' \in SCC(S)$. By Lemma 4.3.4, $S^\star$ cannot contain any $a \in W(S')$, otherwise it would inherit a $\downarrow$-labelled TP-arc (every $a \in W(S')$ has a $\downarrow$-labelled TP-arc by definition), so in fact $S^\star \subseteq S' \setminus W(S')$. If $W(S')$ is empty, $SCP(S)$ evaluates to *False*. So assume $W(S')$ is non-empty. Let $S'' = S' \setminus W(S')$. Then $S^\star \subseteq S'' \subseteq S$ and $\|S'' \setminus S^\star\| < d$. By the induction hypothesis, $SCP(S'')$ evaluates to *False*. It follows that $SCP(S)$ evaluates to *False*.

4.3.2 Generality of SCP

An example in detail. Let us consider the application of SCP to the Ackermann example.

Recall that the Ackermann function (Ex. 10) has two abstract function-call transitions $\mathtt{ack} \overset{G}{\to} \mathtt{ack}$, where $G = \{\mathtt{m} \overset{\downarrow}{\to} \mathtt{m}\}$, and $\mathtt{ack} \overset{G'}{\to} \mathtt{ack}$, where $G' = \{\mathtt{m} \overset{\mp}{\to} \mathtt{m}, \mathtt{n} \overset{\downarrow}{\to} \mathtt{n}\}$. *Tr* has 3 strongly-connected subsets: the singleton sets containing each abstract function-call transition and the set with both of them.

For the strongly-connected component with both abstract function-call transitions, the TP-set is $\{\mathtt{ack}\cdot\mathtt{m}\}$, and there is indeed a $\downarrow$-labelled TP-arc in G. The SCP procedure ignores the strongly-connected set with just $\mathtt{ack} \overset{G}{\to} \mathtt{ack}$, because it is recognized that $\mathtt{m} \overset{\downarrow}{\to} \mathtt{m}$ arc continues to be a TP-arc for that set. The remaining strongly-connected set has just $\mathtt{ack} \overset{G'}{\to} \mathtt{ack}$. The TP-set expands to $\{\mathtt{ack}\cdot\mathtt{m}, \mathtt{ack}\cdot\mathtt{n}\}$, and $\mathtt{n} \overset{\downarrow}{\to} \mathtt{n} \in G'$ is a $\downarrow$-labelled TP-arc.

Ex.	SCT	ISD	SCP	Terminating	Description
1	Yes	Yes	Yes	Yes	High-order example with obvious descent.
2	Yes	Yes	Yes	Yes	High-order example with less obvious descent.
3	Yes	Yes	Yes	Yes	As example 2. Program truncates lazy trees.
4	No	No	No	No	`f`-closure passes into `f`-environment.
5	No	No	No	Yes	`f`-closure passes into `f`-environment.
6	Yes	Yes	Yes	Yes	Factorial by Y combinator.
7	Yes	Yes	Yes	Yes	Simulating let binding.
8	Yes	Yes	Yes	Yes	Simple in-situ descent. Program reverses lists.
9	Yes	Yes	Yes	Yes	Simple in-situ descent with indirect recursion.
10	Yes	Yes	Yes	Yes	Lexical descent. The Ackermann function.
11	*Yes*	*No*	*Yes*	*Yes*	*Descent in sum of parameter sizes.*
12	*Yes*	*Yes*	*No*	*Yes*	*Descent in maximum of parameter sizes.*
13	*Yes*	*No*	*No*	*Yes*	*Permuted and discarded parameters.*
14	Yes	Yes	Yes	Yes	Insertion sort.
15	Yes	Yes	Yes	Yes	Quicksort.

Figure 4.4: Examples handled by SCP

SCP more generally. SCP has been designed on the premise that an infinitely descending thread usually traverses few parameters, often just a single parameter, as when an inductive variable is decreased in a direct recursive call. SCP handles such situations well. Now it is quite common for an infinitely descending thread to traverse a fixed parameter for each of a number of functions during indirect recursion. SCP handles such situations by "focusing" on different parameters for different functions: For Ex. 1 and 2, SCP deduces that an infinitely descending thread has to be present either between $\mathtt{f}\cdot\mathtt{y}$ and $\mathtt{k}\cdot\mathtt{z}$, or between $\mathtt{f}\cdot\mathtt{x}$ and $\mathtt{k}\cdot\mathtt{x}$.

SCP also handles "permuted argument descent" (as in Ex. 11), where threads do not interfere with one another. SCP gets in trouble if the value of a "descent parameter" may be discarded. For Ex. 12, the entire *Tr* is a strongly-connected component. A multipath with infinity set *Tr* has an infinitely descending thread that uses both $\mathtt{f}\cdot\mathtt{x}$ and $\mathtt{f}\cdot\mathtt{y}$. The SCP criterion fails because the value of `x` or `y` may not be continued at `f`. A similar comment applies to Ex. 13. However, these forms of descent are not commonly observed in practice. The Ackermann example shows that SCP is able to consider different "descent parameters" for different multipaths.

The table in Fig. 4.4 shows the results of applying SCP to the programs considered previously for SCT and ISD. They indicate that SCP is sufficiently general for common forms of descent.

4.3.3 Time-complexity of SCP

In practice, each strongly-connected component S of *Tr* is processed separately. Let $\|S\| = K$, F be the set of source and target functions of S elements, and A be the maximum of arity of functions in F. (For most components, K and A would be small.) Now, S refers to $O(K)$ functions, so the (fan-in-free) SRG of an S element may be represented explicitly with only $O(AK)$ parameters and $O(AK)$ arcs, as the parameters of functions not in F can be ignored. The space requirement for S is therefore $N \sim O(AK^2)$. We work out the worst-case time-complexity of the *SCP* procedure below.

1. **Computation of** $TP(S')$**.**

   We first describe the procedure for computing $TP(S')$ for $S' \subseteq S$. A set of *counts* is maintained for the source parameters of each $f \xrightarrow{G'} f' \in S'$. The count for parameter x in $f \xrightarrow{G'} f' \in S'$ initially records the out-degree of x in G', i.e., $\|\{x' \mid x \xrightarrow{r} x' \in G'\}\|$.

   Initialization: Begin by *marking* every $f \cdot x$ such that for some $f \xrightarrow{G'} f' \in S'$, the source parameter x has count 0. Insert all such $f \cdot x$ into a *worklist*. *Iteration*: When processing some $g \cdot y$ from the worklist, inspect those abstract function-call transitions $f \xrightarrow{G'} g \in S'$ such that G' contains an arc of the form $x \xrightarrow{r} y$. If $f \cdot x$ has *not* been marked, reduce the count of the source parameter x of G'. If the count becomes 0, mark $f \cdot x$ and insert it into the worklist. Terminate when the worklist is empty.

It is easy to prove that upon termination, the set of *unmarked* $f \cdot x$ forms $TP(S')$ (we omit the proof). The initialization considers each source parameter and each size-relation arc at most once, and the iteration considers each size-relation arc and each TP-pair at most once. The algorithm clearly has worst-case time-complexity *linear* in the size of S', i.e., the space required to represent it. However, an initialization phase is needed to index all the functions and parameters, and set up the necessary indexing tables. This is an operation performed on S with time-complexity quadratic in the number of functions referred to in S and the total number of parameters for these functions.

2. **Computation of *SCP*(S).**

   The *SCP* procedure in Fig. 4.3.1 on page 63 considers each element of S in no more than K invocations. Each invocation involves linear operations: computing strongly-connected components [CLR90], computing the TP-set, identifying TP-witnesses. The time complexity is therefore $O(NK) \sim O(AK^3)$. The initialization phase adds another cost of $O((AK)^2) = O(A^2K^2)$.

We believe that the SCP criterion is a practical approximation to SCT for program termination analysis. It should be sufficiently general in practice, and it has a decision procedure with a good time-complexity. Of course, an empirical study is required to confirm the practicality of SCP.

Chapter 5

Complexity of SCT and ISD

5.1 Hard problems

In this chapter, we prove that the SCT and ISD criteria are PSPACE-hard to decide. This establishes that they are complete for PSPACE. We recall some well-known facts about the PSPACE complexity class [Pap94].

1. PSPACE = NPSPACE. Thus non-determinism does not increase the problems solvable in polynomial space.
2. PSPACE = $\overline{\text{PSPACE}}$. The class of PSPACE-complete problems is closed under complementation.

As usual, the PSPACE-hardness proofs in this chapter are by reduction from some known PSPACE-hard problem. One such problem is the termination of boolean programs, which we discuss next.

5.2 Boolean programs

A boolean program b is comprised of a set of variables $\{\mathtt{X}_1, \ldots, \mathtt{X}_k\}$ and a sequence of instructions,

$$\begin{array}{l} 1 : I_1 \\ \vdots \\ n : I_n \end{array}$$

where each I_ℓ has one of the following forms.

The `not` instruction	$\mathtt{X}_i$ `:= not` $\mathtt{X}_i$
The `if` instruction	`if` $\mathtt{X}_i$ `goto` ℓ' `else` ℓ''

Let $i, i', i'' \ldots$ always range over $[1, k]$ and $\ell, \ell', \ell'' \ldots$ always range over $[1, n+1]$. For the `if` instruction, refer to ℓ' and ℓ'' as the positive and negative branch respectively. We assume that $k \leq n$.

Definition 5.2.1 (Semantics of boolean program b) The *computation* of b is a finite or infinite *state transition sequence* of the form $sts = (\ell_0, \sigma_0) \xrightarrow{r_1} (\ell_1, \sigma_1) \ldots$, where each $\sigma_t : \{\mathtt{X}_1, \ldots, \mathtt{X}_k\} \to \{\mathtt{True}, \mathtt{False}\}$. Each ℓ_t and σ_t are called the *control point* and *store* at time t respectively. The symbol $r_{t+1} \in \{\natural, \sharp, \flat\}$ records whether a transition has been due to a `not` instruction, or is the result of following a positive branch, or the result of following a negative branch.

For boolean program b with k variables and n instructions, define its *state transition sequence*, of the form $sts(b) = (\ell_0, \sigma_0) \xrightarrow{r_1} (\ell_1, \sigma_1) \ldots \xrightarrow{r_t} (\ell_t, \sigma_t) \ldots$, as follows.

- Let $\ell_0 = 1$ and for $i = 1, \ldots, k$: $\sigma_0(\mathtt{X}_i) = \mathtt{False}$.
- Given current state (ℓ_t, σ_t), if $\ell_t = n + 1$, then b has terminated, and σ_t is called the *terminating state*; otherwise:
  - Suppose that I_{ℓ_t} has the form $\mathtt{X}_i$ `:= not` $\mathtt{X}_i$.
    In this case, let $\ell_{t+1} = \ell_t + 1$ and $\sigma_{t+1} = \sigma_t[\mathtt{X}_i \mapsto \neg\sigma_t(\mathtt{X}_i)]$.
  - Suppose that I_{ℓ_t} has the form `if` $\mathtt{X}_i$ `goto` ℓ' `else` ℓ''.
    In this case, let $\sigma_{t+1} = \sigma_t$, and if $\sigma_t(\mathtt{X}_i) = \mathtt{True}$, $\ell_{t+1} = \ell'$ else $\ell_{t+1} = \ell''$.

b is *terminating* (i.e., its computation is terminating) if $sts(b)$ if finite.

Definition 5.2.2 Let b be a boolean program with k variables and n instructions, $I_1, \ldots, I_n$. The *normalization* of b is a boolean program b' with k variables and instructions $I'_1, \ldots, I'_{n+2k}$ such that:

- For $\ell = 1, \ldots, n$, I'_ℓ is the same as I_ℓ, except every reference to the program point $n+1$ is replaced with a reference to $n + 2k + 1$.

- For $i = 1, \ldots, k$,
  - I'_{n+2i-1} is `if` $\mathtt{X}_i$ `goto` ℓ' `else` ℓ'', where $\ell' = n + 2i$ and $\ell'' = n + 2i + 1$, and
  - I'_{n+2i} is $\mathtt{X}_i$ `:= not` $\mathtt{X}_i$.

Lemma 5.2.3 For boolean program b with k variables and n instructions, and its normalization b' with $n + 2k$ instructions:

1. b' has no more than $3n \sim O(n)$ instructions.
2. b' terminates iff b terminates.
3. If b' terminates, and σ is its terminating store, then for $i = 1, \ldots, k$, we have $\sigma(\mathtt{X}_i) = \mathtt{False}$.

Theorem 5.2.4 The set $\mathcal{B} = \{b \mid sts(b) \text{ is finite}\}$ is complete for PSPACE.

Proof $\mathcal{B}$ is in PSPACE by direct simulation, using a counter to declare non-termination if the computation for b with k variables and n instructions has exceeded $(n + 1) \cdot 2^k$ steps. For PSPACE-hardness, reduce QBF (truth of quantified boolean formulae) to membership in $\mathcal{B}$ [Jon97].

The set $\mathcal{B}$ above is a subset of all boolean programs. By Lemma 5.2.3, the theorem also holds if it is interpreted as a subset of the normalized boolean programs.

5.3 Complexity of SCT

Theorem 5.3.1 Deciding whether a given Tr satisfies SCT is PSPACE-hard.

The idea. The approach to the proof is this. For any normalized boolean program b with k variables and n instructions, show that a set of abstract function-call transitions Tr_b can be constructed such that Tr_b has size polynomial in n, and Tr_b satisfies SCT just when b does *not* terminate. It follows that SCT is hard for $\overline{\text{PSPACE}} = \text{PSPACE}$.

We will use a single function identifier for Tr_b: $Fun = \{\mathtt{F}\}$. Since every element of Tr_b will have the form $\mathtt{F} \xrightarrow{G} \mathtt{F}$, we omit the source and target functions altogether, and represent an abstract function-call transition as an SRG. The variable set for Tr_b will be

$$Var = \{\mathtt{X}_i \mid 1 \le i \le k\} \cup \{\overline{\mathtt{X}}_i \mid 1 \le i \le k\} \cup \{\mathtt{W}_\ell \mid 1 \le \ell \le n + 1\}.$$

In our reduction proof, the parameters $\mathtt{X}_i$ and $\overline{\mathtt{X}}_i$ are associated with the truth and falsity of boolean variable $\mathtt{X}_i$ respectively. Each $\mathtt{W}_\ell$ corresponds to a possible control point.

Tr_b has a "start graph" G such that every Tr_b-multipath beginning with G represents (in some sense) a possible execution of b. SCT is a property of infinite multipaths. The idea is to design Tr_b so that the only infinite Tr_b-multipath without infinite descent is the simulation of a terminating computation, repeated ad infinitum. Then Tr_b fails to satisfy SCT just when the computation of b is finite, i.e., b is terminating.

For this to work, any infinite Tr_b-multipath $\mathcal{M}$ representing a persistently incorrect simulation of b, e.g., $\mathcal{M}$ repeatedly fails to respect control flow, or correct branching behaviour, should be "punished" with infinite descent. The $\mathtt{X}_i$, $\overline{\mathtt{X}}_i$ and $\mathtt{W}_\ell$ parameters implement this by tracking the store and control point in a simulation. It should also be ensured that any Tr_b-multipath that (eventually) simulates an infinite b computation has infinite descent.

The graphs. For any b, Tr_b has graph $G^\natural_{n+1,1}$, which begins a b-simulation. Tr_b also contains a graph $G^\natural_{\ell,\ell+1}$ for each `not` instruction I_ℓ in the program, and a pair of graphs $G^\sharp_{\ell,\ell'}, G^\flat_{\ell,\ell''}$ for each instruction I_ℓ of the form `if` $\mathtt{X}_i$ `goto` ℓ' `else` ℓ''. We describe the arcs among the $\mathtt{X}_i, \overline{\mathtt{X}}_i$ parameters and the arcs among the $\mathtt{W}_\ell$ parameters for each SRG separately. (It may help to consult Fig. 5.1 on page 73 for examples of Tr_b graphs.)

$\mathtt{X}$-arcs for $G^{\natural}_{n+1,1}$. The start graph $G^{\natural}_{n+1,1}$ has the following arcs.

$$\begin{array}{l} \overline{\mathtt{X}}_i \xrightarrow{\downarrow} \mathtt{X}_i \\ \overline{\mathtt{X}}_i \xrightarrow{\overset{=}{\downarrow}} \overline{\mathtt{X}}_i \end{array}$$

These arcs indicate that the current $\overline{\mathtt{X}}_i$ value is greater or equal in size compared to the $\overline{\mathtt{X}}_i$ value after the function-call transition, and strictly greater in size than the $\mathtt{X}_i$ value after the transition. Intuitively, a non-descending thread starting from the source parameter $\overline{\mathtt{X}}_i$ in this graph is used to express the value of boolean variable $\mathtt{X}_i$ in a correct simulation: Finding such a thread continued to $\mathtt{X}_i$ in a multipath expresses that the boolean variable $\mathtt{X}_i$ has value `True` at that point in the simulation; finding such a thread continued to $\overline{\mathtt{X}}_i$ expresses that the boolean variable $\mathtt{X}_i$ has value `False`. As a correct simulation of b does *not* begin with $\mathtt{X}_i$ having the value `True`, the arc from $\overline{\mathtt{X}}_i$ to $\mathtt{X}_i$ is "punished" with a $\downarrow$-label. (It is "punished" as it expresses incorrect information.)

$\mathtt{X}$-arcs for $G^{\sharp}_{\ell,\ell'}$. The graph $G^{\sharp}_{\ell,\ell'} \in \mathit{Tr}_b$ implies that I_ℓ has the form `if` $\mathtt{X}_{i'}$ `goto` ℓ' `else` ℓ''. The graph has arcs to punish any thread at $\overline{\mathtt{X}}_{i'}$ (a non-descending thread at $\overline{\mathtt{X}}_{i'}$ expresses that the boolean variable $\mathtt{X}_{i'}$ has value `False` in the simulation, in which case following the positive # branch renders the simulation incorrect).

$$\begin{array}{l} \mathtt{X}_{i'} \xrightarrow{\overset{=}{\downarrow}} \mathtt{X}_{i'} \\ \overline{\mathtt{X}}_{i'} \xrightarrow{\downarrow} \overline{\mathtt{X}}_{i'} \\ \mathtt{X}_i \xrightarrow{\overset{=}{\downarrow}} \mathtt{X}_i \text{ for } i \neq i' \\ \overline{\mathtt{X}}_i \xrightarrow{\overset{=}{\downarrow}} \overline{\mathtt{X}}_i \text{ for } i \neq i' \end{array}$$

$\mathtt{X}$-arcs for $G^{\flat}_{\ell,\ell''}$. The graph $G^{\flat}_{\ell,\ell''} \in \mathit{Tr}_b$ implies that I_ℓ has the form `if` $\mathtt{X}_{i'}$ `goto` ℓ' `else` ℓ''. The arcs of this graph are similar to the arcs of $G^{\sharp}_{\ell,\ell'}$, except it is the non-descending thread at $\mathtt{X}_{i'}$, rather than $\overline{\mathtt{X}}_{i'}$, that is punished.

$$\begin{array}{l} \mathtt{X}_{i'} \xrightarrow{\downarrow} \mathtt{X}_{i'} \\ \overline{\mathtt{X}}_{i'} \xrightarrow{\overset{=}{\downarrow}} \overline{\mathtt{X}}_{i'} \\ \mathtt{X}_i \xrightarrow{\overset{=}{\downarrow}} \mathtt{X}_i \text{ for } i \neq i' \\ \overline{\mathtt{X}}_i \xrightarrow{\overset{=}{\downarrow}} \overline{\mathtt{X}}_i \text{ for } i \neq i' \end{array}$$

$\mathtt{X}$-arcs for $G^{\natural}_{\ell,\ell+1}$. The graph $G^{\natural}_{\ell,\ell+1} \in \mathit{Tr}_b$ implies that I_ℓ has the form $\mathtt{X}_{i'}$ `:= not` $\mathtt{X}_{i'}$. The graph has arcs to move any non-descending thread from $\mathtt{X}_{i'}$ to $\overline{\mathtt{X}}_{i'}$ and vice versa.

$$\begin{array}{l} \mathtt{X}_{i'} \xrightarrow{\overset{=}{\downarrow}} \overline{\mathtt{X}}_{i'} \\ \overline{\mathtt{X}}_{i'} \xrightarrow{\overset{=}{\downarrow}} \mathtt{X}_{i'} \\ \mathtt{X}_i \xrightarrow{\overset{=}{\downarrow}} \mathtt{X}_i \text{ for } i \neq i' \\ \overline{\mathtt{X}}_i \xrightarrow{\overset{=}{\downarrow}} \overline{\mathtt{X}}_i \text{ for } i \neq i' \end{array}$$

Lemma 5.3.2 ($\mathtt{X}$-arcs simulate the store) Let Tr_b be the graphs derived for the boolean program b. Let $\mathit{sts}(b) = (\ell_0, \sigma_0) \xrightarrow{r_1} (\ell_1, \sigma_1) \ldots \xrightarrow{r_t} (\ell_t, \sigma_t) \ldots$. Define the graph sequence $G^\star_t$ inductively.

$$\begin{array}{lcl} G^\star_0 & = & G^{\natural}_{n+1,1} \\ G^\star_{t+1} & = & G^\star_t ; G^{r_{t+1}}_{\ell_t,\ell_{t+1}} \end{array}$$

The claims are:

1. If $\sigma_t(\mathtt{X}_i) = \mathtt{False}$, then $G^\star_t$ has arcs $\overline{\mathtt{X}}_i \xrightarrow{\stackrel{=}{\downarrow}} \overline{\mathtt{X}}_i$ and $\overline{\mathtt{X}}_i \xrightarrow{\downarrow} \mathtt{X}_i$ (and no other $\mathtt{X}_i$, $\overline{\mathtt{X}}_i$ arcs).
2. If $\sigma_t(\mathtt{X}_i) = \mathtt{True}$, then $G^\star_t$ has arcs $\overline{\mathtt{X}}_i \xrightarrow{\downarrow} \overline{\mathtt{X}}_i$ and $\overline{\mathtt{X}}_i \xrightarrow{\stackrel{=}{\downarrow}} \mathtt{X}_i$ (and no other $\mathtt{X}_i$, $\overline{\mathtt{X}}_i$ arcs).

Proof is by induction on t.

- $t = 0$: For each i, $\sigma_t(\mathtt{X}_i) = \mathtt{False}$, and $G^\star_0$ has arcs $\overline{\mathtt{X}}_i \xrightarrow{\stackrel{=}{\downarrow}} \overline{\mathtt{X}}_i$ and $\overline{\mathtt{X}}_i \xrightarrow{\downarrow} \mathtt{X}_i$ by definition.
- Assume that the claim is true for t. And perform case analysis on the form of I_{ℓ_t}.
  - I_{ℓ_t} is $\mathtt{X}_{i'}$ `:= not` $\mathtt{X}_{i'}$: By the semantics of b, $\sigma_{t+1}(\mathtt{X}_{i'}) = \neg\sigma_t(\mathtt{X}_{i'})$. Let $\sigma_t(\mathtt{X}_{i'}) = \mathtt{True}$. By the induction hypothesis, $G^\star_t$ has arcs $\overline{\mathtt{X}}_{i'} \xrightarrow{\downarrow} \overline{\mathtt{X}}_{i'}$ and $\overline{\mathtt{X}}_{i'} \xrightarrow{\stackrel{=}{\downarrow}} \mathtt{X}_{i'}$. By definition, $G^{\natural}_{\ell_t,\ell_{t+1}}$ has arcs $\overline{\mathtt{X}}_{i'} \xrightarrow{\stackrel{=}{\downarrow}} \mathtt{X}_{i'}$ and $\mathtt{X}_{i'} \xrightarrow{\stackrel{=}{\downarrow}} \overline{\mathtt{X}}_{i'}$. So $G^\star_{t+1}$ has arcs $\overline{\mathtt{X}}_{i'} \xrightarrow{\stackrel{=}{\downarrow}} \overline{\mathtt{X}}_{i'}$ and $\overline{\mathtt{X}}_{i'} \xrightarrow{\downarrow} \mathtt{X}_{i'}$, and the claim is satisfied for b variable $\mathtt{X}_{i'}$. For b variable $\mathtt{X}_i$ with $i \neq i'$, observe that $\mathtt{X}_i$ and $\overline{\mathtt{X}}_i$ arcs are unaffected by the composition, and $\sigma_{t+1}(\mathtt{X}_i) = \sigma_t(\mathtt{X}_i)$. The case of $\sigma_t(\mathtt{X}_{i'}) = \mathtt{False}$ is proved similarly.
  - I_{ℓ_t} is `if` $\mathtt{X}_{i'}$ `goto` ℓ' `else` ℓ'': Suppose that $(\ell_t, \sigma_t) \xrightarrow{\sharp} (\ell_{t+1}, \sigma_{t+1})$. By the semantics of b, $\sigma_t(\mathtt{X}_{i'}) = \mathtt{True}$. By the induction hypothesis, $G^\star_t$ has arcs $\overline{\mathtt{X}}_{i'} \xrightarrow{\downarrow} \overline{\mathtt{X}}_{i'}$ and $\overline{\mathtt{X}}_{i'} \xrightarrow{\stackrel{=}{\downarrow}} \mathtt{X}_{i'}$. The graph $G^{\sharp}_{\ell_t,\ell_{t+1}}$ has arcs $\overline{\mathtt{X}}_{i'} \xrightarrow{\downarrow} \overline{\mathtt{X}}_{i'}$ and $\mathtt{X}_{i'} \xrightarrow{\stackrel{=}{\downarrow}} \mathtt{X}_{i'}$. So $G^\star_{t+1}$ has the same $\mathtt{X}_{i'}$, $\overline{\mathtt{X}}_{i'}$ arcs as $G^\star_t$. It also has the same $\mathtt{X}_i$, $\overline{\mathtt{X}}_i$ arcs as $G^\star_t$ for $i \neq i'$. Since $\sigma_{t+1} = \sigma_t$, the claim is satisfied. The case for $(\ell_t, \sigma_t) \xrightarrow{\flat} (\ell_{t+1}, \sigma_{t+1})$ is similar.

Corollary 5.3.3 ($\mathtt{X}$-arc behaviour for finite b computation) Let Tr_b be graphs for the normalized boolean program b. Let b be terminating and $sts(b) = (\ell_0, \sigma_0) \xrightarrow{r_1} (\ell_1, \sigma_1) \ldots \xrightarrow{r_T} (\ell_T, \sigma_T)$. Then the composition $G^\star = G^{\natural}_{n+1,1}; G^{r_1}_{\ell_0,\ell_1}; \ldots; G^{r_T}_{\ell_{T-1},\ell_T}$ has $\mathtt{X}$, $\overline{\mathtt{X}}$ arcs $\overline{\mathtt{X}}_i \xrightarrow{\stackrel{=}{\downarrow}} \overline{\mathtt{X}}_i$ and $\overline{\mathtt{X}}_i \xrightarrow{\downarrow} \mathtt{X}_i$ for each i.

Proof follows from Lemma 5.3.2 and the fact that b is normalized, so $\sigma_T(\mathtt{X}_i) = \mathtt{False}$ for each i.

Corollary 5.3.4 ($\mathtt{X}$-arc behaviour for incorrect simulations) Let Tr_b be graphs for the boolean program b. And let $sts(b) = (\ell_0, \sigma_0) \xrightarrow{r_1} (\ell_1, \sigma_1) \ldots \xrightarrow{r_t} (\ell_t, \sigma_t) \xrightarrow{r_{t+1}} (\ell_{t+1}, \sigma_{t+1}) \ldots$. Now suppose that the *finite* multipath $\mathcal{M} = G^{\natural}_{n+1,1}, G^{r_1}_{\ell_0,\ell_1}, \ldots, G^{r_t}_{\ell_{t-1},\ell_t}, G^{r'_{t+1}}_{\ell_t,\ell_{t+1}}, \ldots$ correctly simulates b up to time t. However, let r'_{t+1} be different from r_{t+1}. (That is, the simulation has "gone wrong.") In this case, I_{ℓ_t} must have form `if` $\mathtt{X}_{i'}$ `goto` ℓ' `else` ℓ''. *Claim*: There are complete threads in $\mathcal{M}$ containing $\downarrow$-labelled arcs from $\overline{\mathtt{X}}_{i'}$ to *both* $\mathtt{X}_{i'}$ and $\overline{\mathtt{X}}_{i'}$.

Proof Let $r_{t+1} = \sharp$ and $r'_{t+1} = \flat$. Note that this means $\sigma_t(\mathtt{X}_{i'}) = \mathtt{True}$. The case of $r_{t+1} = \flat$ and $r'_{t+1} = \sharp$ is similar.

By Lemma 5.3.2, composition $G^\star$ of $\mathcal{M}$ graphs up to and including $G^{r_t}_{\ell_{t-1},\ell_t}$ has arcs $\overline{\mathtt{X}}_{i'} \xrightarrow{\downarrow} \overline{\mathtt{X}}_{i'}$ and $\overline{\mathtt{X}}_{i'} \xrightarrow{\stackrel{=}{\downarrow}} \mathtt{X}_{i'}$. By definition, $G^{\flat}_{\ell_t,\ell_{t+1}}$ has arcs $\mathtt{X}_{i'} \xrightarrow{\downarrow} \mathtt{X}_{i'}$ and $\overline{\mathtt{X}}_{i'} \xrightarrow{\stackrel{=}{\downarrow}} \overline{\mathtt{X}}_{i'}$. Composing $G^\star$ with it results in an SRG with arcs $\overline{\mathtt{X}}_{i'} \xrightarrow{\downarrow} \mathtt{X}_{i'}$ and $\overline{\mathtt{X}}_{i'} \xrightarrow{\downarrow} \overline{\mathtt{X}}_{i'}$. These arcs remain upon further composition with *any* graph of Tr_b. Therefore, they are present in the composition of all of $\mathcal{M}$'s graphs. By Lemma 4.1.4, there exist threads with $\downarrow$-labelled arcs connecting $\overline{\mathtt{X}}_{i'}$ to both $\mathtt{X}_{i'}$ and $\overline{\mathtt{X}}_{i'}$ over the length of $\mathcal{M}$.

The $\mathtt{W}$-arcs. Let $\mathcal{M}$ be a Tr_b-multipath. Refer to consecutive graphs $G^{r'}_{\ell,\ell'}, G^{r'''}_{\ell'',\ell'''}$ in $\mathcal{M}$ as a *b-legal transition* if $\ell' = \ell''$. Otherwise, refer to them as a *b-illegal transition*. Call multipath $\mathcal{M}$ *b-legal* if it contains only b-legal transitions. Otherwise, describe it as *b-illegal*. From the Tr_b perspective, any sequence of graphs forms a flow-legal multipath (since the source and target functions are always $\mathtt{F}$). We need a way to distinguish b-illegal multipaths.

For $G^{r'}_{\ell,\ell'} \in Tr_b$, if $\ell' \neq n+1$, let $G^{r'}_{\ell,\ell'}$ have the following arcs.

$$\begin{array}{l} \mathtt{W}_\ell \xrightarrow{\mp} \mathtt{W}_{\ell'} \\ \mathtt{W}_{\ell''} \xrightarrow{\downarrow} \mathtt{W}_{(\ell''+\ell'-\ell) \bmod (n+1)} \text{ for } \ell'' \neq \ell \end{array}$$

where $m \bmod (n+1)$ is the number r in $[1, n+1]$ such that for some $k : k(n+1)+r = m$. We will always use this definition of mod.

The arcs of $G^{r'}_{\ell,\ell'}$ are realized by a "rotation" of the $\mathtt{W}$ arguments by $\|\ell' - \ell\|$, together with some size-decreasing operations. The idea is to capture a change of $\|\ell' - \ell\|$ to the program counter. Only the thread reaching the proper control point $\mathtt{W}_{\ell'}$ is spared from receiving a $\downarrow$-label. Every other thread is continued with one.

For $G^r_{\ell,n+1} \in Tr_b$, let $G^r_{\ell,n+1}$ have the following arcs.

$$\begin{array}{l} \mathtt{W}_\ell \xrightarrow{\mp} \mathtt{W}_{n+1} \\ \mathtt{W}_\ell \xrightarrow{\downarrow} \mathtt{W}_{\ell'} \text{ for } \ell' \neq n+1 \end{array}$$

These arcs mark the end of a b simulation. Any thread at the correct control point $\mathtt{W}_\ell$ is continued. Every other thread is terminated. The thread at $\mathtt{W}_\ell$ is continued without a $\downarrow$-label to the next correct control point $\mathtt{W}_{n+1}$. It is also "broadcast" to every $\mathtt{W}_{\ell'}$ for $\ell' \neq n+1$. This ensures that any incorrect control point decision (b-illegal transition) is recorded in an infinite thread.

Lemma 5.3.5 ($\mathtt{W}$-arc behaviour in a b-legal multipath) Let Tr_b be graphs for the boolean program b. And let $sts(b) = (\ell_0, \sigma_0) \xrightarrow{r_1} (\ell_1, \sigma_1) \ldots \xrightarrow{r_t} (\ell_t, \sigma_t) \ldots$. Define the graph sequence $G^\star_t$ as before.

$$\begin{array}{lcl} G^\star_0 & = & G^\natural_{n+1,1} \\ G^\star_{t+1} & = & G^\star_t ; G^{r_{t+1}}_{\ell_t,\ell_{t+1}} \end{array}$$

The claims:

1. If $\ell_t \neq n+1$ then $G^\star_t$ has arcs $\mathtt{W}_{n+1} \xrightarrow{\mp} \mathtt{W}_{\ell_t}$, and $\mathtt{W}_\ell \xrightarrow{\downarrow} \mathtt{W}_{(\ell+\ell_t) \bmod (n+1)}$ for $\ell \neq n+1$ (and no other $\mathtt{W}$-arcs).

2. If $\ell_t = n+1$ then $G^\star_t$ has arcs $\mathtt{W}_{n+1} \xrightarrow{\mp} \mathtt{W}_{n+1}$, and $\mathtt{W}_{n+1} \xrightarrow{\downarrow} \mathtt{W}_\ell$ for $\ell \neq n+1$ (and no other $\mathtt{W}$-arcs).

Proof of the first item is by induction on t.

- $t = 0$: $\ell_t = 1$ and $G^\star_0 = G^\natural_{n+1,1}$ has arcs $\mathtt{W}_{n+1} \xrightarrow{\mp} \mathtt{W}_1$, and $\mathtt{W}_\ell \xrightarrow{\downarrow} \mathtt{W}_{(\ell+1) \bmod (n+1)}$ for $\ell \neq n+1$. This is by definition.

- $t > 0$: Assume that the claim is true for t. For $\ell_{t+1} \neq n+1$, the graph $G^{r_{t+1}}_{\ell_t,\ell_{t+1}}$ has arcs $\mathtt{W}_{\ell_t} \xrightarrow{\mp} \mathtt{W}_{\ell_{t+1}}$ and $\mathtt{W}_\ell \xrightarrow{\downarrow} \mathtt{W}_{(\ell+\ell_{t+1}-\ell_t) \bmod (n+1)}$ for $\ell \neq \ell_t$. By the induction hypothesis, $G^\star_t$ has arcs $\mathtt{W}_{n+1} \xrightarrow{\mp} \mathtt{W}_{\ell_t}$, and $\mathtt{W}_\ell \xrightarrow{\downarrow} \mathtt{W}_{(\ell+\ell_t) \bmod (n+1)}$ for $\ell \neq n+1$. Thus, $G^\star_{t+1}$ has arcs $\mathtt{W}_{n+1} \xrightarrow{\mp} \mathtt{W}_{\ell_{t+1}}$, and $\mathtt{W}_\ell \xrightarrow{\downarrow} \mathtt{W}_{(\ell+\ell_{t+1}) \bmod (n+1)}$ for $\ell \neq n+1$.

The second item of the claim follows immediately from the following observation. Let $\ell_t = n+1$. The graph $G^\star_{t-1}$ has arcs $\mathtt{W}_{n+1} \xrightarrow{\mp} \mathtt{W}_{\ell_{t-1}}$, and $\mathtt{W}_\ell \xrightarrow{\downarrow} \mathtt{W}_{(\ell+\ell_{t-1}) \bmod (n+1)}$ for $\ell \neq n+1$. And the graph $G^{r_t}_{\ell_{t-1},n+1}$ has arcs $\mathtt{W}_{\ell_{t-1}} \xrightarrow{\mp} \mathtt{W}_{n+1}$, and $\mathtt{W}_{\ell_{t-1}} \xrightarrow{\downarrow} \mathtt{W}_\ell$ for $\ell \neq n+1$. Therefore, $G^\star_t$ has arcs $\mathtt{W}_{n+1} \xrightarrow{\mp} \mathtt{W}_{n+1}$, and $\mathtt{W}_{n+1} \xrightarrow{\downarrow} \mathtt{W}_\ell$ for $\ell \neq n+1$.

Lemma 5.3.6 ($\mathtt{W}$-arc behaviour in a b-illegal multipath.) Suppose $\mathcal{M} = G^r_{\ell,n+1}, G_1, G_2 \ldots, G_T$ is b-illegal. Then the composition of the graphs of $\mathcal{M}$ has arcs $\mathtt{W}_\ell \xrightarrow{\downarrow} \mathtt{W}_{\ell'}$ for all ℓ'.

Proof Define the graph sequence $G^\star_t$ inductively.

$$\begin{array}{rcl} G^\star_0 & = & G^r_{\ell,n+1} \\ G^\star_{t+1} & = & G^\star_t ; G_{t+1} \end{array}$$

Let $G_t = G_{\ldots,\ell'}$. *Claim*: If $\mathcal{M}$ up to and including G_t is b-legal, then $G^\star_t$ has arcs $\mathtt{W}_\ell \xrightarrow{\overset{=}{\downarrow}} \mathtt{W}_{\ell'}$ and $\mathtt{W}_\ell \xrightarrow{\downarrow} \mathtt{W}_{\ell''}$ for $\ell'' \neq \ell'$. If the multipath up to and including G_t is not b-legal, then $G^\star_t$ has arcs $\mathtt{W}_\ell \xrightarrow{\downarrow} \mathtt{W}_{\ell''}$ for every ℓ''. This claim is proved by induction on t and case analysis for the W-arcs of G_t. The proof is straightforward, but tedious. Let us argue less formally.

An equivalent statement of the lemma is that threads containing $\downarrow$-labelled arcs connect $\mathtt{W}_\ell$ to each W variable over the length of $\mathcal{M}$, *if* $\mathcal{M}$ has a b-illegal transition. Observe that the first graph in $\mathcal{M}$, namely $G^r_{\ell,n+1}$, establishes a connection between $\mathtt{W}_\ell$ and each W variable. Every subsequent graph either continues the thread at each W variable, or, for a particular W variable, the thread there is continued to every W variable. (Consult Fig. 5.1.) In either case, there continues to be a thread connecting $\mathtt{W}_\ell$ to each W variable over the multipath. Now, suppose that a thread connects $\mathtt{W}_\ell$ to some $\mathtt{W}_{\ell'}$ over the length of $\mathcal{M}$, but is without a $\downarrow$-labelled arc. Since each graph in $\mathcal{M}$ has only one $\overset{=}{\downarrow}$-labelled W-arc, this means all $\overset{=}{\downarrow}$-labelled arcs have to line up end-to-end between $\mathtt{W}_\ell$ to $\mathtt{W}_{\ell'}$ over the length of $\mathcal{M}$. From the definition of the W-arcs, this in fact implies that $\mathcal{M}$ is b-legal, which contradicts the premise. We conclude that any thread connecting $\mathtt{W}_\ell$ to a W variable over $\mathcal{M}$ contains a $\downarrow$-labelled arc.

The graphs. It should be clear from the proof of Lemma 5.3.6 that diagrams are extremely helpful when reasoning about Tr_b-multipaths. Consider the following boolean program b.

```
1:  X₁ := not X₁
2:  if X₁ goto 1 else 3
3:  X₂ := not X₂
4:  if X₂ goto 1 else 5
```

Let us represent the store as a pair of booleans that indicate the values of $\sigma(\mathtt{X}_1)$ and $\sigma(\mathtt{X}_2)$, writing T for `True` and F for `False`. Then b has the following state transition sequence.

$$\begin{array}{rcl} sts(b) & = & (1,\mathtt{FF}) \xrightarrow{\natural} (2,\mathtt{TF}) \xrightarrow{\sharp} (1,\mathtt{TF}) \xrightarrow{\natural} (2,\mathtt{FF}) \xrightarrow{\flat} (3,\mathtt{FF}) \xrightarrow{\natural} (4,\mathtt{FT}) \xrightarrow{\sharp} \\ & & (1,FT) \xrightarrow{\natural} (2,\mathtt{TT}) \xrightarrow{\sharp} (1,\mathtt{TT}) \xrightarrow{\natural} (2,\mathtt{FT}) \xrightarrow{\flat} (3,\mathtt{FT}) \xrightarrow{\natural} (4,\mathtt{FF}) \xrightarrow{\flat} (5,\mathtt{FF}) \end{array}$$

The PSPACE-hardness proof encodes termination of programs like b as an SCT problem. It is no wonder that SCT is difficult.

Figure 5.1 shows the multipath simulating a b computation. To avoid clutter, all labels are omitted, but $\downarrow$-labelled arcs are drawn in heavy lines. *Claim*: The represented multipath, repeated ad infinitum, has no infinite descent. Observe how descent is only just avoided everywhere.

The proof. We now formally prove that deciding SCT is PSPACE-hard.

Proof Given normalized boolean program b, work out Tr_b as described. Suppose b is terminating. And let $sts(b) = (\ell_0, \sigma_0) \xrightarrow{r_1} \ldots \xrightarrow{r_T} (\ell_T, \sigma_T)$. By Corollary 5.3.3, $G^\star = G^\natural_{n+1,1}; G^{r_1}_{\ell_0,\ell_1}; \ldots; G^{r_T}_{\ell_{T-1},\ell_T}$ has the following X, $\overline{\mathtt{X}}$-arcs.

$$\begin{array}{l} \overline{\mathtt{X}}_i \xrightarrow{\overset{=}{\downarrow}} \overline{\mathtt{X}}_i \\ \overline{\mathtt{X}}_i \xrightarrow{\downarrow} \mathtt{X}_i \end{array}$$

Since b is terminating, $\ell_T = n+1$, so by Lemma 5.3.5, $G^\star$ has the following W-arcs.

$$\begin{array}{l} \mathtt{W}_{n+1} \xrightarrow{\overset{=}{\downarrow}} \mathtt{W}_{n+1} \\ \mathtt{W}_{n+1} \xrightarrow{\downarrow} \mathtt{W}_\ell \text{ for } \ell \neq n+1 \end{array}$$

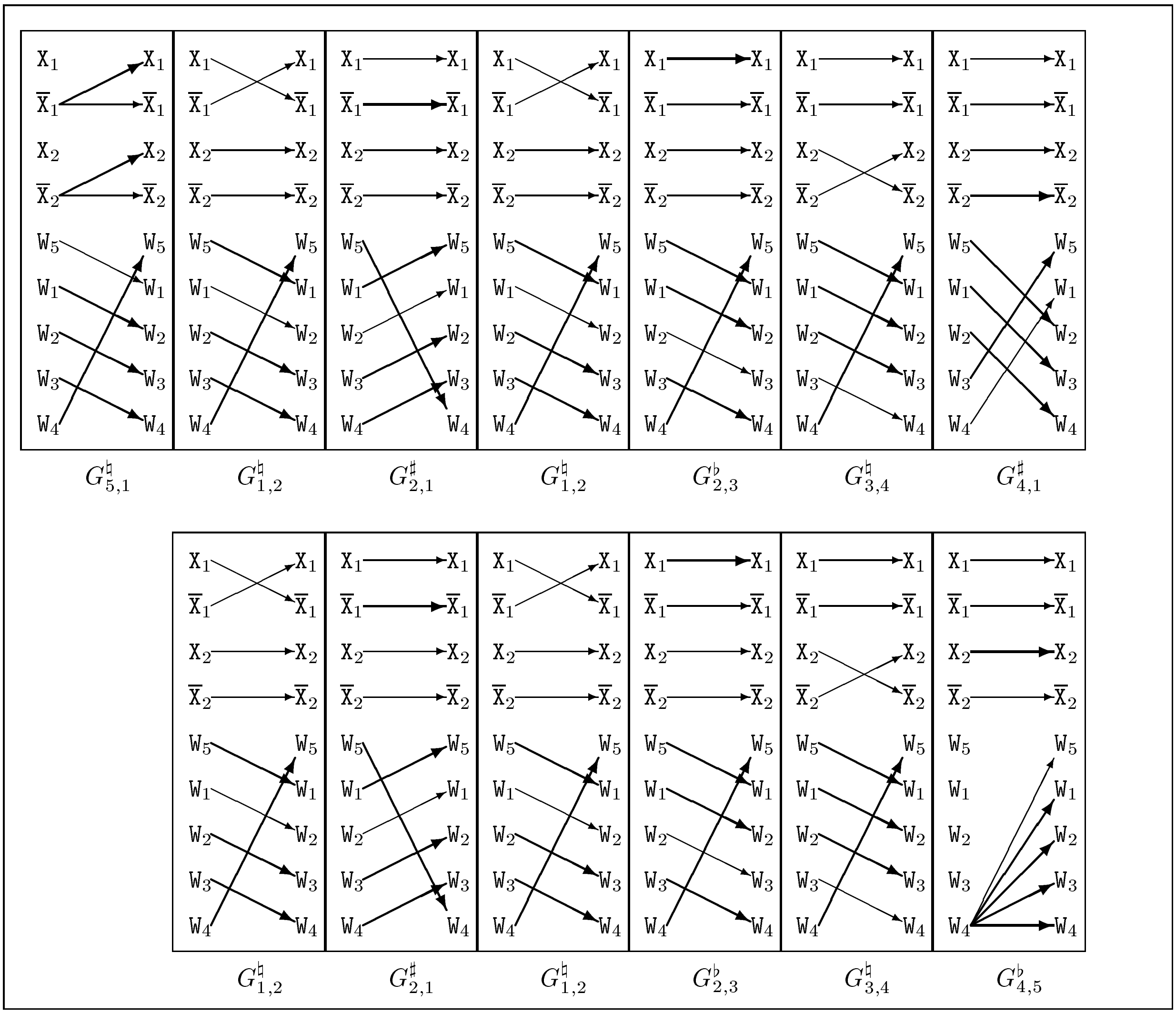


Figure 5.1: Simulation of a b computation

Clearly, $G^\star$ is an idempotent graph of $\overline{Tr_b}$ without any arc of the form $x \xrightarrow{\downarrow} x$. By Theorem 4.1.6, Tr_b does not satisfy SCT.

Conversely, suppose that b does not terminate. And let $\mathcal{M} = G_0 G_1 \ldots$ be an infinite Tr_b-multipath. We must show that $\mathcal{M}$ has infinite descent, for then Tr_b satisfies SCT.

- Scenario 1: $\mathcal{M}$ contains finitely many graphs of the form $G^r_{\ell,n+1}$. Then $\mathcal{M}$ has a tail $\mathcal{M}'$ completely free from such graphs. Multipath $\mathcal{M}'$ has exactly $n+1$ maximal threads formed from the W-arcs. And infinitely many $\downarrow$-labels occur among these threads. Thus $\mathcal{M}'$ has at least one thread with infinite descent. Hence, $\mathcal{M}$ has infinite descent.

- Scenario 2: $\mathcal{M}$ contains infinitely many graphs of the form $G^r_{\ell,n+1}$ and infinitely many b-illegal transitions. Decompose a tail of $\mathcal{M}$ as $G^{r_1}_{\ell_1,n+1}, \mathcal{M}_1, G^{r_2}_{\ell_2,n+1}, \mathcal{M}_2, \ldots$ where each $G^{r_t}_{\ell_t,n+1}, \mathcal{M}_t$ contains a b-illegal transition. It follows from Lemma 5.3.6 that each $G^{r_t}_{\ell_t,n+1}, \mathcal{M}_t$ has a thread from $\mathtt{W}_{\ell_t}$ to $\mathtt{W}_{\ell_{t+1}}$ containing a $\downarrow$-labelled arc. Therefore $\mathcal{M}$ has infinite descent.

- Scenario 3: $\mathcal{M}$ contains infinitely many graphs of the form $G^r_{\ell,n+1}$ and finitely many b-illegal transitions. In this case, $\mathcal{M}$ has a tail $\mathcal{M}'$ that is b-legal and contains infinite occurrences of $G^\natural_{n+1,1}$ (which must follow every graph of form $G^r_{\ell,n+1}$ in a b-legal multipath).

Divide $\mathcal{M}'$ into consecutive finite sections $G^{\natural}_{n+1,1}, \mathcal{M}_1, G^{\natural}_{n+1,1}, \mathcal{M}_2, \ldots$ such that each $\mathcal{M}_t$ is free from $G^{\natural}_{n+1,1}$. Since each section $G^{\natural}_{n+1,1}, \mathcal{M}_t$ is b-legal, it must "go wrong" somewhere by following an incorrect conditional branch. By Corollary 5.3.4, each section $G^{\natural}_{n+1,1}, \mathcal{M}_t$ has a thread with a $\downarrow$-labelled arc from some $\overline{\mathtt{X}}_{i'}$ to itself over the length of that section. The set of all threads from any $\overline{\mathtt{X}}_i$ to itself over the length of each $G^{\natural}_{n+1,1}, \mathcal{M}_t$ form exactly k maximal threads in $\mathcal{M}'$. Since each $G^{\natural}_{n+1,1}, \mathcal{M}_t$ section contributes at least one $\downarrow$-label to these threads, one such thread has to have infinite descent.

Since under every scenario, an infinite Tr_b-multipath has infinite descent, we conclude that Tr_b satisfies SCT. This completes the proof of the theorem stated at the start of the chapter.

5.4 Complexity of ISD

Theorem 5.4.1 Deciding whether a given Tr satisfies ISD is PSPACE-hard.

Proof The proof of this requires only a small adaptation to the construction of Tr_b. Add three parameters to *Var*. Call them $\mathtt{Z}_1, \mathtt{Z}_2$ and $\mathtt{Z}_3$. Replace $G^{\natural}_{n+1,1}$ with $G^{\sharp}_{n+1,1}$ and $G^{\flat}_{n+1,1}$, which have the same $\mathtt{X}$ and $\mathtt{W}$ arcs as $G^{\natural}_{n+1,1}$. Additionally, $G^{\sharp}_{n+1,1}$ has $\mathtt{Z}_1 \xrightarrow{\downarrow} \mathtt{Z}_1$, and $G^{\flat}_{n+1,1}$ has $\mathtt{Z}_2 \xrightarrow{\downarrow} \mathtt{Z}_2$. Every other graph in Tr_b has both $\mathtt{Z}_1 \xrightarrow{\downarrow} \mathtt{Z}_1$ and $\mathtt{Z}_2 \xrightarrow{\downarrow} \mathtt{Z}_2$. Every graph *not* of the form $G^{r'}_{\ell,\ell'}$ where $\ell' = n+1$ is given the arc $\mathtt{Z}_3 \xrightarrow{\downarrow} \mathtt{Z}_3$. We next prove that the new Tr_b satisfies ISD just when b is not terminating.

For the forward direction, let the normalized boolean program b be terminating, and let its state transition sequence be $sts(b) = (\ell_0, \sigma_0) \xrightarrow{r_1} (\ell_1, \sigma_1) \ldots \xrightarrow{r_T} (\ell_T, \sigma_T)$. Consider the graph

$$G^{\star} = (G^{\sharp}_{n+1,1}; G^{r_1}_{\ell_0,\ell_1}; \ldots; G^{r_T}_{\ell_{T-1},\ell_T}); (G^{\flat}_{n+1,1}; G^{r_1}_{\ell_0,\ell_1}; \ldots; G^{r_T}_{\ell_{T-1},\ell_T}).$$

From the hardness proof of SCT, we already know that $G^{\star}$ has $\mathtt{X}, \overline{\mathtt{X}}$-arcs: $\overline{\mathtt{X}}_i \xrightarrow{\mp} \overline{\mathtt{X}}_i$ and $\overline{\mathtt{X}}_i \xrightarrow{\downarrow} \mathtt{X}_i$, and $\mathtt{W}$-arcs: $\mathtt{W}_{n+1} \xrightarrow{\mp} \mathtt{W}_{n+1}$ and $\mathtt{W}_{n+1} \xrightarrow{\downarrow} \mathtt{W}_{\ell}$ for $\ell \neq n+1$. Clearly, $G^{\star}$ has no arcs involving the $\mathtt{Z}$ variables: the $\mathtt{Z}_1$, $\mathtt{Z}_2$, $\mathtt{Z}_3$ threads are broken at $G^{\flat}_{n+1,1}$, $G^{\sharp}_{n+1,1}$, $G^{r_T}_{\ell_{T-1},\ell_T}$ respectively (recall that $\ell_T = n+1$). Therefore $G^{\star}$ is a graph in $\overline{Tr_b}$ without any arc of the form $x \xrightarrow{\downarrow} x$. By Theorem 4.2.4, Tr_b does not satisfy ISD.

For the backward direction, let b be non-terminating, and consider a Tr_b-loop $\mathcal{M}$. We want to prove that $\mathcal{M}$ has a complete thread with a $\downarrow$-labelled arc from some variable x to itself. Call such a thread an *ISD-thread in x*. For $\mathcal{M}$ not to have an ISD thread in the $\mathtt{Z}$ variables, it has to contain at least one $G^{\sharp}_{n+1,1}$, one $G^{\flat}_{n+1,1}$ and some graph of form $G^{r}_{\ell,n+1}$. So this is the only situation that need concern us. Now, having both $G^{\sharp}_{n+1,1}$ and $G^{\flat}_{n+1,1}$ means that $\mathcal{M}$ contains at least one complete b simulation. Since b is non-terminating, the simulation is in error. Specifically, $\mathcal{M}$ either contains a b-illegal transition, or follows an incorrect conditional branch. We consider each possibility in turn.

Let $\mathcal{M}$ contain a b-illegal transition. Write $\mathcal{M}$ as $\mathcal{M}_1, G^{r}_{\ell,n+1}, \mathcal{M}_2$ (where each $\mathcal{M}_t$ may be empty). Consider the unique thread *th* ending at $\mathtt{W}_{\ell}$ constructed as follows. Let $x = \mathtt{W}_{\ell}$ initially and inspect the graphs of $\mathcal{M}_1$ from right to left. For each graph, select the unique arc that ends at x, then set x to the source parameter of this arc before looking at the next graph. By design of the $\mathtt{W}$-arcs, it is possible to construct *th*. Further, *th* is free from $\downarrow$-labelled arcs just when $\mathcal{M}_1, G^{r}_{\ell,n+1}$ is b-legal. Suppose that $\mathcal{M}_1, G^{r}_{\ell,n+1}$ contains a b-illegal transition. Then *th* is a complete thread in $\mathcal{M}_1$ with a $\downarrow$-labelled arc connecting some $\mathtt{W}$ variable to $\mathtt{W}_{\ell}$. It is easy to see that $G^{r}_{\ell,n+1}, \mathcal{M}_2$ has a complete thread connecting $\mathtt{W}_{\ell}$ to each $\mathtt{W}$ variable. Therefore, $\mathcal{M}$ has an ISD-thread in some $\mathtt{W}$ variable. Next, suppose that the b-illegal transition occurs in $G^{r}_{\ell,n+1}, \mathcal{M}_2$. It follows from Lemma 5.3.6 that a complete thread containing a $\downarrow$-labelled arc connects $\mathtt{W}_{\ell}$ to each $\mathtt{W}$ variable over $G^{r}_{\ell,n+1}, \mathcal{M}_2$. The thread *th* connects some $\mathtt{W}$ variable to $\mathtt{W}_{\ell}$ over $\mathcal{M}_1$. Therefore, $\mathcal{M}$ still has an ISD-thread in some $\mathtt{W}$ variable.

Finally, suppose that $\mathcal{M}$ is b-legal. Write $\mathcal{M}$ as $\mathcal{M}_1, G^{r}_{n+1,1}, \mathcal{M}_2, G^{r'}_{n+1,1}, \mathcal{M}_3$ (where $\mathcal{M}_2$ is free from graphs of the form $G^{r''}_{n+1,1}$ and each $\mathcal{M}_t$ may be empty). Consider the section $G^{r}_{n+1,1}, \mathcal{M}_2$. Since it is b-legal and completes with a graph of the form $G^{r''}_{\ell,n+1}$, it must "go wrong" somewhere by following an incorrect branch. By Corollary 5.3.4, this multipath section has an ISD-thread in some $\overline{\mathtt{X}}_{i'}$. It is easy to show that one of $\mathtt{X}_{i'}$ or $\overline{\mathtt{X}}_{i'}$ is connected to $\overline{\mathtt{X}}_{i'}$ over $\mathcal{M}_1$. It is also easy to show that complete threads connect $\overline{\mathtt{X}}_{i'}$ to both $\mathtt{X}_{i'}$ and $\overline{\mathtt{X}}_{i'}$ over $G^{r'}_{n+1,1}, \mathcal{M}_3$. It follows that there is an ISD-thread in either $\mathtt{X}_{i'}$ or $\overline{\mathtt{X}}_{i'}$. This completes the proof that any Tr_b-loop has an ISD-thread for b non-terminating.

5.5 About the complexity results

Tr_b is realizable. Tr_b is easily realized by a first-order tail-recursive L program. The program has a single function `F`, and the definition of `F` contains a tail recursive call for each graph $G \in Tr_b$. Assume an ordering of the parameter set: $x_1, \ldots, x_K$, and let the formal parameter list of `F` be in this order. Given graph G, construct a tail recursive call to be included in the body of `F`. The i-th argument of this call is determined as follows: If there is an arc of the form $x_j \xrightarrow{\overset{=}{\downarrow}} x_i \in G$, then the i-th argument is just x_j; if there is an arc of the form $x_j \xrightarrow{\downarrow} x_i \in G$, then the i-th argument is (`tl` x_j); if there does not exist any arc of the form $x_j \xrightarrow{r} x_i$ in G, then the i-th argument is `nil`. This is well-defined because graphs of Tr_b are fan-in-free. The recursive function calls are pieced together in a nested conditional expression to form the body of `F`. The graphs of Tr_b are then recoverable precisely from the definition of `F` using SRG analysis.

Consequence of the complexity results. The complexity results mean that it is not possible to inspect short Tr-multipaths and make rigorous conclusions about whether Tr satisfies SCT or ISD, at least not generally. It is possible to construct examples of Tr (e.g., Tr_b of Fig. 5.1) such that all short loops, when repeated infinitely, exhibit infinite descent, and only exponentially long loops betray that SCT is violated.

Other PSPACE-hard analyses. The first-order L-program realizing a given Tr_b can be coded for Prolog. It can be shown that Termilog [LSS97] and Terminweb [CT97] are able to give precise answers to SCT queries posed indirectly this way. These analyses are therefore also PSPACE-hard.

We may regard SCT a clean formulation of the termination reasoning behind these approaches to program termination analysis. It is owing to the cleanness of the formulation that the complexity results of this chapter have been achieved.

Chapter 6

Termination of Offline Partial Evaluation

6.1 Introduction to the variable boundedness problem

A program specializer performs some form of evaluation of the subject program, based on a partially specified input. The idea is to symbolically interpret as much of the program as possible, to avoid the same computation at runtime. However, as the input is underspecified, certain computations have to be suspended. To represent such computations, the evaluation environment maps variables to expressions (*dynamic* data), as well as values (*static* data). For higher-order programs, variables may also be mapped to closures, whose arguments are a mix of expressions, values and closures. Such closures are called *partially static*.

Given an expression of the subject program and an evaluation environment, the specializer takes different actions depending on the type of data (static or dynamic) contained in the relevant variables. This in turn determines the type of the result. If an operation is attempted on the wrong type of data, the specializer is said to experience a *type error*.

To avoid type errors during mixed computation, one approach is tag every piece of data with its type. The specializer then decides what actions to perform by consulting these tags during specialization. Alternatively, by static analysis, type information is collected for each variable and expression of the subject program. Consistent typing is ensured by suitable insertion of "casting operators," to indicate coersion of static data into dynamic data during specialization. With the casting operators, it is possible to annotate every program construct, indicating the type of action to be taken during specialization, in a way that guarantees no type errors will be committed. Such a scheme leads to an efficient, syntax-directed specialization phase. This approach is called *offline partial evaluation*.

In offline partial evaluation, every variable is classified as *static* ("always available") or *dynamic*, before any specialization occurs. The classification of a variable is called its *binding-time*. A *binding-time division* of variables induces a *binding-time type* for each program expression, and a *binding-time annotation* for each program construct.

An expression is classified dynamic if its evaluation will result in dynamic data, and static if its evaluation will result in static data. A program construct (conditional, operation, function reference or application) is *annotated* static if it is to remain in the residual code, and dynamic if it is to be reduced away. Generally, a static construct is treated as in ordinary evaluation, and a dynamic construct is handled by evaluating its (dynamic) subexpressions, and piecing together a residual expression from their results. We use a third binding-time annotation *memo* for marking function references and applications. It indicates the generation of a residual function application. (Note that the use of explicit *memo* annotations deviates from standard presentations.)

To ensure consistent binding-time types (consider a static conditional with branches of different types), and annotatability (consider an operation with some dynamic arguments), the operator `lift` is used to cast certain expressions from static to dynamic. The *memo* annotation also has a casting effect, since *memo*-marked expressions are dynamic. In fact, it will be ensured, through the use of *memo* annotations, that `lift` is never applied to non base values.

The need to prevent the lifting of closures interferes with the binding-time division. For instance, consider a conditional with a dynamic branch, and a static branch consisting of just a variable reference. Suppose that according to analysis, a (partially) static closure may be assigned to the variable. As the variable cannot be lifted, the binding-time division has to be adjusted instead. On the other hand, we will be increasing the *memo* annotations and dynamic binding-times in any case, as a means to control the amount of reduction, and thereby ensure the termination of the specialization process. In this work, any mechanism that decreases the amount of reduction (and hence increases the chances of successful specialization) is called a *generalization*.

Consistent binding-time typing and annotatability are enforced using *well-annotatedness* rules. These rules subsume the *congruence requirements*:

- Any expression whose result may be assigned to program variable x during specialization must have the same binding-time type as x.
- Any expression that may cause a (static) function-call transition to f during specialization must have the same binding-time type as the body of f.

Apart from congruence, other requirements on the binding-time division and annotations are needed to guarantee freedom from type errors, and ensure other desirable properties. These requirements include:

- The entry function variables must have their specified binding-times.
- The function body of `goal` must be dynamic.
- If a function f is *memo*-annotated in the subject program, then a specialization of f may be required for which the values of the f parameters are unknown. Thus the f parameters must be dynamic, and the body of f must be dynamic.
- If $\langle f, \zeta_1, \ldots, \zeta_n \rangle$ is a possible value for the rator e_1 of a *memo*-annotated application $(e_1\ e_2)$, then a specialization of f may be required for which the values of $f^{(n+2)}, \ldots, f^{(ar(f))}$ are unknown. Thus these parameters must be dynamic, and the body of f must be dynamic.
- An application must *not* be annotated static if this may give rise to a (static) function-call transition that leads to computation duplication, or the discarding of errors or non-termination.

The penultimate item, in particular, means it has to be determined before specialization, which closures may occur in a *dynamic context*, so that the necessary generalizations are performed. Further generalizations can ensure finite unfolding and finite function specialization. This leads us to a discussion of the role of variable boundedness analysis.

The purpose of *variable boundedness analysis* (also called *finiteness analysis* [Hol91]) is to determine whether the variables of the subject program assume only finitely many values during any specialization. This property, together with sufficient *memo*-annotations in the program, ensure that specialization terminates. The idea is to perform controlled generalization so that variable boundedness and other requirements: congruence, annotatability, etc. are satisfied simultaneously. The goal is to generate for any legal input a residual program in finite time. Naturally, the amount of generalization should be minimized, as the trivial solution to lift every static input variable, and make every program expression dynamic is always viable, albeit at the expense of all specialization!

Road map to this chapter. In this chapter, we review the limited literature on variable boundedness analysis. These works have been developed over the past decade, and all use essentially the same *boundedness condition*, with a few differences in the presentation. We compare and contrast the various approaches, and indicate those aspects that will be incorporated in our formulation.

We then define an *annotated syntax* and a *specialization semantics* for L programs, which is proved to generate residual programs preserving the semantics of the subject program in some sense. The requirements on the binding-time division and the annotations are then specified. Among these are generalizations for ensuring finite specialization. It turns out that termination analysis is useful in at least two ways:

- It allows unused but terminating computations to be safely discarded, which may lead to better residual code.
- It is readily adapted to prevent infinite unfolding during specialization.

We show how a binding-time division and well-annotations can be derived satisfying all the requirements for finite and safe specialization, *given* a suitably formulated variable boundedness analysis. This chapter culminates in the statement of the variable boundedness problem, whose solution would complete the development, and give a correct and terminating specializer for L programs. The study of the variable boundedness problem begins in earnest from the next chapter.

6.2 A brief review

The scope of variable boundedness analysis is perhaps narrower than termination analysis. In this respect, it is unlike termination analysis, which has been well researched in a variety of contexts, including logic programming, functional programming, and term-rewriting systems. In development spanning a decade [Hol91, JGS93, GJ96, AH96, Gle99], the criterion used to assess the boundedness of variables has remained essentially the same. No generalization or approximation of this condition has been suggested. And its complexity has never been studied. Instead, effort has been directed at sharpening the supporting analyses needed for its application, and obtaining better formulations.

Andersen and Holst [AH96] have discussed interesting techniques to treat higher-order programs. For example, they observe that single-threadedness information is useful for handling programs in CPS (although we point out later that decreasing continuation values admit a natural size-descent argument). These techniques are applied orthogonally to the main variable boundedness criterion. In this work, our goal will be to study a general *principle* for deciding variable boundedness, which (strictly) subsumes the existing works on variable boundedness analysis. In particular, we will be interested in developing a *practical* criterion based on this principle. It is for this reason that our choice of supporting analyses will be based on simplicity, rather than strength.

6.2.1 Holst's formulation

Holst [Hol91] first stated and proved the correctness of an in-situ descent (ISD) condition for variable boundedness, which he formulated for first-order functional programs. Assume we are operating in a downward-closed data domain, i.e., for any value v, the set of values less than or equal to v is finite. (This assumption continues to hold for the remainder of this literature review.) Suppose we are given a congruent binding-time division for the subject program. The following is essentially what was proved in [Hol91].

> Suppose that in some specialization loop (not necessarily a simple one), a variable x is assigned the result u of a computation that has used an old value v of x, and u is not less than or equal to v. Call such a variable *auto-constructive* in the loop.
>
> If whenever a variable is auto-constructive, the value of some bounded static variable is decreased, then every variable is bounded.

A clarification. Let us refer to a decreasing bounded static variable as an *anchor*, following Glenstrup [Gle99]. The auto-construction in some variable x may be accompanied by different anchors over different loops. For instance, consider the following program, where every parameter is assumed to be static.

```
f x y dp = if ... (f (tl x) y (dp + 1))
                  (f x (tl y) (dp + 1))
```

The parameter `dp` tracks the "recursion depth" in an execution of the above (essentially) first-order program. Its boundedness implies the termination of `f`, as observed by Glenstrup [Gle99]. Any loop from `f` to itself sees an auto-construction in `dp`. For the simple loop comprising of a single function call due to `(f (tl x) y (dp + 1))`, the bounded static variable `x` is decreased, while in the simple loop comprising of a single function call due to `(f x (tl y) (dp + 1))`, the bounded static variable `y` is decreased. The crucial observation is that *every* possible loop (including the non-simple ones, e.g., two function calls due to `(f x (tl y) (dp + 1))`, followed by three due to `(f (tl x) y (dp + 1))` is accompanied by a decrease in `x` or `y` or both. By Holst's ISD criterion, it can be concluded that `dp` is bounded.

The ISD variable boundedness condition places a limit on the number of observed auto-constructions in `dp`. (Note that the same limit applies regardless of the starting state in the auto-constructive sequence. Contrast this with our termination criteria, which seek to limit the length of a function-call transition sequence from *each* starting state.) For the example above, arbitrarily large growth in `dp` would be accompanied by arbitrarily large descent in $sx + sy$, where sx and sy are the sizes of `x` and `y`'s values.

Since the value of $sx + sy$ is bounded (because `x` and `y` are bounded variables), the boundedness of `dp` follows. The same reasoning applies to the example below.

```
f x y dp = if ... (f y (tl x) (dp + 1))
                  (f x (tl y) (dp + 1))
```

However, this example is *not* handled by the ISD variable boundedness condition. This demonstrates that the condition does not capture size-descent reasoning in a completely natural way.

The correctness proof. The correctness proof of the ISD variable boundedness condition in [Hol91] is highly combinatorial. A close inspection of the proof reveals the *principle* behind the condition.

> Unbounded sequences of increases (constructions) in a variable are *impossible*, if they would give rise to unbounded sequences of size decreases for some bounded-variable values.

The interpretation of the above statement is not so straightforward. By *unbounded sequences of increases in x*, we mean that it is possible to produce for any given N, a *finite* sequence of function-call transitions that exhibit more than N increases in x. (The starting state of the call sequence is irrelevant.) The claim is: This is impossible, if it allows us to exhibit unbounded sequences of size decreases for some bounded-variable values.

For the example program on the previous page, unbounded sequences of increases in `dp` either entail state transition sequences that feature arbitrarily many repeats of the top call, in which case unbounded sequences of size decreases would be observed for `x` values; or they entail state transition sequences that feature arbitrarily many repeats of the bottom call, in which case unbounded sequences of size decreases would be observed for `y` values. Both situations are impossible, since `x` and `y` are (clearly) bounded variables. In [Hol91], a finitary description is proposed for the argument dependencies and data size relations observed in possible finite state transition sequences, in order to realize such reasoning.

It should come as little surprise that the Finite Ramsey's Theorem can be used to simplify the correctness proof of the ISD variable boundedness condition in [Hol91]. In fact, as we shall see, the application of Ramsey's Theorem suggests an immediate generalization of the condition, in the same way that SCT generalizes the ISD *termination* criterion.

To iterate or not to iterate? The ISD variable boundedness condition invites iteration (with the suitable modifications). However, Holst never suggested this. Instead, anchors are taken to be those static variables that are *clearly* bounded. (For instance, a static variable is clearly bounded if it is always in-situ decreasing or assigned constant values.) By not iterating, the resulting approach to variable boundedness does not handle lexical descent. Consider the following example, where as before, every variable is assumed to be static.

```
ack m n dp = if (m = 0) (n + 1)
                (if (n = 0) (ack (m ⊖ 1) 1 (dp + 1))
                            (ack (m ⊖ 1)
                                 (ack m (n ⊖ 1) (dp + 1))
                                 (dp + 1)))
```

The above is the Ackermann function, augmented with a recursion depth parameter `dp`, whose boundedness implies that `ack` is terminating. The variable `m` is clearly bounded. Variable `n` is bounded because whenever it is auto-constructive, `m` is decreased. Finally, `dp` is bounded because whenever it is auto-constructive, either `m` or `n` is decreased (and both `m` and `n` are bounded). Observe that `n` may even increase over some loops where `m` is the anchor. Such secondary effects are missed if the ISD variable boundedness condition is not iterated. Note that Holst appealed directly to lexical descent for the termination of `ack` in [Hol91]. The variable boundedness condition is used there only to deduce the boundedness of `n` from the boundedness of `m`.

The iteration has arisen because the variable boundedness condition mentions *known* bounded variables. Andersen and Holst [AH96] appeared to have confused this iteration with the alternation of binding-time analysis and variable boundedness analysis, when they described their formulation as "top-down," as opposed to Glenstrup and Jones' approach in [GJ96], which was described as "bottom-up." In fact, Glenstrup [Gle99] has suggested both iterating a suitable formulation of the ISD variable boundedness condition, *and* alternating the variable boundedness analysis with binding-time analysis.

An explicit principle for bounded variables? Holst's formulation does not account for the dependence of an auto-constructive variable x on unbounded variables not involved in its auto-construction. It does not need to: If a limit has been deduced for the number of auto-constructions in x, but x is unbounded, then x depends on some x' such that there is unbounded auto-construction in x'. In Holst's formulation, x' is made dynamic. By congruence, the dependence of x on x' means that x is made dynamic too. In [Hol91], Holst derives a binding-time division that is congruent, and such that the static variables are bounded. We will see that this is sufficient for finite function specialization. However, for an explicit condition to deduce bounded variables from known bounded variables, one suitable for iteration, the dependence of a variable on possibly unbounded variables not involved in its auto-construction has to be considered.

To alternate or not to alternate? The procedure suggested by Holst for deriving a congruent division such that the static variables are bounded is a simple process: Obtain a congruent division, determine all those variables whose auto-construction have anchors in clearly bounded static variables, generalize every other variable as dynamic, and re-establish congruence. (The final step is necessary– for instance, a non auto-constructive variable may depend on one of the generalized variables.)

Holst reasons that the re-establishing of congruence does not affect the anchors (which do not depend on the generalized variables in his set-up). Therefore there is no need to alternate binding-time analysis and variable boundedness analysis. In a more general setting, it is possible for other safety requirements to be compromised by the generalization of possibly unbounded variables. In that case, variable boundedness analysis, and the generalization of possibly unbounded variables, can be alternated with other generalization procedures that ensure additional safety conditions, in order to satisfy all the requirements simultaneously.

Abstract interpretation. The abstract interpretation of [Hol91] consists of a straightforward dependence analysis and a size analysis. The dependent variables of an argument are those variables whose values are used in its computation. If an expression is not definitely decreasing, or non-increasing on some variable occurring in it, it is taken to be possibly constructive over all of its dependent variables. This yields a safe (combined) description of may-depend and definite size-relation information.

6.2.2 Jones, Gomard and Sestoft's formulation

Chapter 14 of Jones, Gomard and Sestoft's textbook [JGS93] presents a variable boundedness analysis for a flow-chart language. The ISD variable boundedness condition given in Ex. 14.8 of Ch. 14 (generalized from the one presented in the body of the chapter) is the basis for the following proposition.

> Describe x as *dependent* on x' if some expression whose value is assigned to x uses x'. Describe x as having an *input dependence* on x' if x is transitively dependent on x', but x' is *not* transitively dependent on x. Describe x as having a *mutual dependence* on x'if x and x' are transitively dependent on each other.
>
> Consider a variable x. Suppose that
>
> - whenever x has an input dependence on x', x' is bounded, and
> - for x' mutually dependent with x, whenever x' is auto-constructive, the value of some bounded static variable z is decreased.
>
> Then, x is bounded, along with any x' with whom it is mutually dependent.

The formulation. The conditions seen above are suitable for iteration. They allow bounded variables to be deduced from known bounded static variables. It is important to be able to collect a set of mutually dependent variables in a single application of the conditions, otherwise indirect recursion would not be handled. Consider the following example, where every variable is assumed to be static.

```
f x dp1 = g (tl x) (dp1 + 1)
g y dp2 = f y (dp2 + 1)
```

Parameters `dp1` and `dp2` are mutually dependent, and have no input dependence on any variable. Whenever one of them is auto-constructive, either `x` or `y` is an anchor. By the above proposition, both recursion depth parameters are bounded.

The application. Jones et al. suggest the following approach to apply the conditions in the proposition. Obtain a congruent division. Iteratively classify variables as bounded based on the proposition until no more bounded variables can be deduced as such. Generalize any remaining static variables not certain to be bounded as dynamic. It is argued that the generalization does not destroy congruence, since each bounded static variable is only dependent on other bounded static variables. However, in a more general setting, other safety requirements may be compromised. In that case, variable boundedness analysis can be used as the basis of a generalization procedure that derives from a congruent division a more general congruent division for which the static variables are bounded. Such a procedure can be alternated with other generalization procedures to ensure that all the safety requirements are satisfied.

6.2.3 Glenstrup's formulation

Glenstrup and Jones [GJ96] and Glenstrup [Gle99] use essentially the same (iterative) formulation of the ISD variable boundedness condition as [JGS93]. As before, a congruent division is obtained, and a subset of the variables are determined to be bounded. Static variables not certain to be bounded are then generalized as dynamic. Additionally, in [Gle99], further generalizations are carried out to prevent infinite unfolding, a step that possibly destroys congruence. For this reason, a generalization procedure based on variable boundedness analysis is alternated with generalizations to ensure finite unfolding and congruence.

The abstract interpretation. An important contribution of [GJ96, Gle99] is the decoupling of may-depend and must-decrease information in the abstract interpretation, leading to sharper results. The dependency analysis regards $e = \texttt{if} \ \ldots \ \texttt{x} \ (\texttt{y} + \texttt{z})$ as having a constructive dependence on `y` and `z`, but a non-constructive dependence on `x`. The goal is to determine whether the evaluation of a given expression can give rise to a value v not bounded by the values of variables used to construct v, which is a natural notion of "constructiveness." For e, the analysis result expresses that if the values of `y` and `z` are fixed, and every other value is allowed to vary, any return value that exceeds the value of `x` (and thus cannot be due to a dependence on `x`) is the same (more generally: is drawn from a finite set).

Glenstrup divides dependencies further into material dependencies, and immaterial dependencies. For instance, in the example below,

```
lenplus x = if (x = nil) (0 : nil)
               (0 : (lenplus (tl x)))
```

the result of `lenplus` only appears to have a *control dependence* on the (initial) input value of `x`. However, it nevertheless has a *size dependence* on `x`, due to recursive inspection of `x`. Such an induced dependency is termed an *immaterial dependency*, to distinguish it from material dependencies, as those seen for expression e above. This development leads to a sharp characterization of an argument expression's dependencies, and seems difficult to extend.

Size analysis, which gives information about size decreases and non-increases among source and target argument values in possible state transitions for the specializer, clearly has a great impact on the success of applying the ISD variable boundedness condition, as it determines which variables are viable anchors. However, Glenstrup [GJ96, Gle99] has only suggested a simple analysis for size relation information, based on the same downward-closed data ordering used to characterize constructiveness. Like other authors, Glenstrup uses a single notion of size for constructiveness and argument descent. In general this is not necessary, nor is it desirable. For constructiveness, what is needed is a size function that is *downward-closed*, i.e., a size function such that for every natural number N, the set of data values with sizes less than or equal to N is a finite set; for argument descent, *any* size function would suffice. In particular, a size function for characterizing argument descent should not be compelled to assign non-trivial sizes for all data types, and it should be allowed to assign data sizes in a way that captures the appropriate notion of size descent, even if downward-closure is violated.

6.2.4 Andersen and Holst's formulation

Andersen and Holst [AH96] have suggested taking size analysis one step further by accounting for disjunctive information. For example, they argue that the result of `f x y = if ... x y` should be recorded as non-decreasing on `x` *or* `y`, then the evaluation of `(f (tl z) (hd z))`, for instance, is definitely decreasing on `z`. However, nothing short of a fully-fledged size analysis, based on a more generous notion of size descent, will handle the boundedness of the recursion depth for quicksort or minsort.

The treatment of variable boundedness analysis by Andersen and Holst [AH96] is based on the original formulation of [Hol91] (although it is indicated that variable boundedness analysis should be alternated with binding-time analysis). The contribution of [AH96] is the handling of higher-order constructs.

Modelling closures. In our terminology, Andersen and Holst model closures and embedded closures using grammars, and define essentially an extension of closure analysis to collect these recursive structures. Their work amply demonstrates the power of abstract analysis to collect very complicated runtime information. Compared to our shallow, "single-level" SRGs, grammar-based analysis will fare better for programs whose termination depends crucially on the *context* of an embedded closure. Consider for instance the situation where the critical parameter-descent is captured by the embedded closure *or* occurs in some parameter captured by the enclosing closure. On the other hand, such reasoning is not expected to be necessary in practice.

Important further research. At the same time, Andersen and Holst's approach raises concerns that are important areas for further research. This work is a study in some of these directions.

1. Their approach possesses a heavy engineering flavour. For instance, it is pointed out that single-threadedness information is important for the termination (and variable boundedness) of CPS programs. Indeed, consider the `append` program written in continuation style.

   ```
   append a b = app a b id

   app x y c = if (x = nil) (c y)
                  (app (tl x) y (cont c x))

   cont k u v = k (hd u : v)

   id z = z
   ```

   Informally, argue that any unfolding of `append` terminates as follows: Only a finite number of closures are generated for `c` in any specialization, since otherwise the list constructors in the value of `x` would decrease infinitely. Each of these `cont` closures is completed at most once, so no termination problems can be caused by unfolding `(c y)`. (On the other hand, each subsequent value of `c` is a smaller closure than the previous one, so the unfolding of `(c y)` also terminates by a descent argument.)

   Discounting those techniques aimed at particular types of programs, it seems that a reasonable number of test examples in [AH96] are handled by the simple *k-limited* approach. It would be ambitious, but important, future research to design an extensible framework, in which specialized techniques can be incorporated. However, it would be premature to consider this when a suitable formulation of the basic variable boundedness condition may handle some of the test examples in [AH96]. It is probably more important to clarify and examine the *principle* behind the variable boundedness condition first.

2. It is conjectured by Andersen and Holst [AH96] that the ISD variable boundedness condition is not intrinsically hard. In fact, a simple adaptation of our earlier hardness proof for the ISD *termination* criterion shows that it is, in fact, PSPACE-hard. On the other hand, authors who have carried out empirical studies (e.g. [LSS97, CT97, Gle99]) have suggested that descriptions of argument behaviour in possible finite state transition sequences have manageable size in practice.

   Nonetheless, our complexity result should motivate the search for effective approximations to the variable boundedness condition that are decidable efficiently. In this work, we will study one such approximation.

3. During the study of termination analysis, we have raised the concern that closure analysis, extended with information about argument behaviour, may produce exponentially large descriptions. We have argued that combinatorial blow-up is probably not common in practice. However, possible widenings is an area for future research.

In view of these hurdles to a practical implementation of the ISD variable boundedness condition, it must be considered very carefully whether the cost of implementing an analysis manipulating grammar-based representations is justified. Clearly, this is intimately connected with the "programs of interest." For general programs, we conjecture that good results are obtainable with the ISD or comparable variable boundedness conditions using "shallow" structures such as the SRG to describe the arguments captured by runtime closures. (Recall that we do approximate embedded closures, though not their contexts.)

6.2.5 This work

The various approaches surveyed all use some variation of the same variable boundedness condition, although the formulation of Jones et al., which is suitable for iteration, may be considered in some ways superior to the original formulation of [Hol91]. Differences among the approaches include abstract interpretation of varying precision, whether application of the variable boundedness condition is iterated, and whether variable boundedness analysis is alternated with generalization procedures that ensure other safety requirements.

Our variable boundedness conditions will be modelled after the formulation of Jones et al. [JGS93]. This means explicit conditions will be given for deducing bounded variables from known bounded static variables. These criteria lend themselves naturally to iteration. As for abstract interpretation, like Glenstrup [GJ96, Gle99], we keep may-dependence information distinct from definite size-change information. The *essence* of the variable boundedness condition is always that unbounded sequences of auto-constructions would give rise to unbounded sequences of size decreases for some bounded-variable values, which is impossible. As for termination analysis, there is free rein as to the choice of data-size function for characterizing argument descent. There is also an independent choice of a downward-closed data ordering for characterizing data-construction.

In earlier chapters, we have defined a size analysis based on ILP techniques. Its results provide access to viable anchors. For simplicity, dependency analysis will be a naive one, although we will distinguish between an expression whose result is always "contained" in the value of some variable, from one that is "possibly constructive." Closure representation is "shallow" in that embedded closures are not modelled directly, as for termination analysis in Part 1 of this work. The structure of a runtime closure can be approximated by inspecting the abstract closures associated with the formal parameters corresponding to the modelled closure's arguments.

We wish to demonstrate a reasonable approach for handling general higher-order programs. However, the primary goal is to carry out a thorough study of possible variable boundedness criteria. In the following chapters, we will propose a generalization of the ISD variable boundedness condition, prove that both the ISD variable boundedness condition and its generalization are PSPACE-hard, and suggest a polynomial-time approximation that seems promising in practice. Clearly, an empirical study is required before a suitable variable boundedness analysis, with a good balance of efficiency and power, can be implemented for the popular offline partial evaluator Similix [Bon91b]. This work is a step towards that goal.

6.2.6 Connection with work on program termination

The variable boundedness principle vs the program termination principle. The variable boundedness principle on which the ISD variable boundedness condition is based seeks to show a limit on the auto-constructions for a variable, by showing that unbounded auto-constructions would be accompanied by unbounded sequences of size decreases for some bounded-variable values. In other words, by considering sufficiently long (finite) sequences of auto-constructions, we can deduce arbitrarily long (finite) sequences of size decreases for some bounded-variable values, which is impossible. These sequences of auto-constructions may occur anywhere in a state transition sequence, and they may have different starting states. Contrast this with the size-change termination principle, which seeks to show a limit on the state transitions from *each* starting state, by showing that unbounded state transitions from any state would be accompanied by unbounded size decreases for some variable values.

Adapting the variable boundedness principle for program termination, we may attempt to establish that unbounded sequences of state transitions would be accompanied by unbounded sequences of size decreases for some bounded-variable values. We will see that this leads to a weaker approach to program termination. From a practical point of view, the principle for variable boundedness is more difficult to apply because it establishes variable boundedness based on anchors that are known to be bounded. It is important that the anchors are bounded. The following is adapted from an example in [AH96]. Assume that every variable is static.

```
g z = f 1 z

f x y = if (y = 0) (f 1 x)
                   (f (x + 1) (y ⊖ 1))
```

Apparently, y is decreased whenever x is auto-constructive. However, as y cannot be determined to be bounded, it is not a viable anchor. This is just as well, since x is actually unbounded: f repeatedly computes the value of its second argument plus 1 in x, and then copies this value back to its second argument y. Therefore, both x and y are unbounded. Observe that each state transition sequence in which the value of x increases infinitely would be accompanied by infinite descent in the sizes of y values, yet this is insufficient for the boundedness of x: The boundedness of the anchor y is crucial.

Bottom-up versus top-down. The variable boundedness principle is based on the observation that unbounded construction is impossible, if it would give rise to unbounded descent in the sizes of bounded-variable values. This leads to an iterative process that collects bounded variables "bottom-up." In contrast, the application of the termination criteria of chapter 4 was "top-down": for these criteria, an infinite function-call transition sequence is sought that casts doubt on termination, because it is not certain to exhibit infinite descent. The absence of such function-call transition sequences confirms program termination. The following example shows that for termination analysis, a top-down approach may be better.

```
f x y z w = if ... (f (x + 1) (y ⊖ 1) z (z ⊖ 1))
                   (f (x ⊖ 1) (y + 1) (w ⊖ 1) w)
```

This example leads to the abstract function-call transitions shown in Fig. 6.1. (As usual, $\downarrow$-arcs are darkened.) Any composition containing both transitions and starting with the one on the left of Fig. 6.1 results in the transition depicted on the left of Fig. 6.2; if the starting transition is the one shown on the right of Fig. 6.1, then the composition is the one depicted on the right of Fig. 6.2. In each case, there is in-situ descent, in z and w respectively. It is clear that any composition containing only the transition on the left of Fig.6.1 has in-situ descent in y, and any composition containing only the transition on the right of Fig. 6.1 has in-situ descent in `x`. By the ISD criterion, the termination analogue of the ISD variable boundedness condition, the example program is terminating.

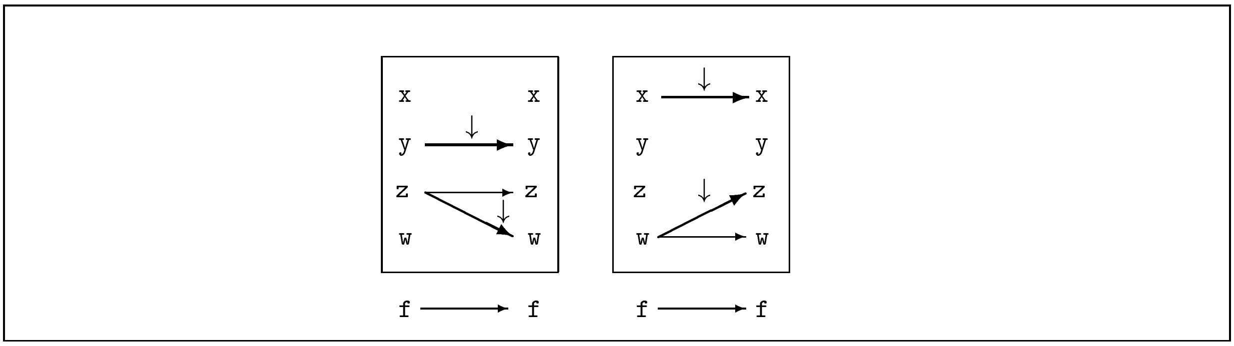


Figure 6.1: Abstract function-call transitions for example

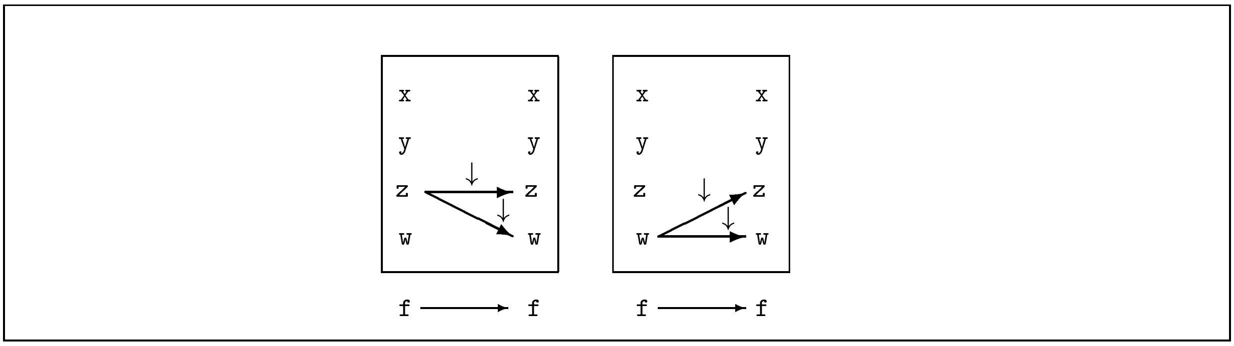


Figure 6.2: Some compositions of the transitions of Fig. 6.1

Now, augment the example with a recursion-depth parameter, whose boundedness implies the termination of the program.

```
f x y z w dp = if ... (f (x + 1) (y ⊖ 1) z (z ⊖ 1) (dp + 1))
                      (f (x ⊖ 1) (y + 1) (w ⊖ 1) w (dp + 1))
```

Assume that every variable is static. And try to prove that `dp` is bounded, to establish that the program is terminating.

Variables `z` and `w` are clearly bounded. However, to prove that `dp` is bounded, an anchor is needed for the loop consisting of a single state transition due to the top function call. The only potential anchor is `y`. So it is necessary to prove that `y` is bounded. To prove that `y` is bounded, the loop consisting of a single state transition due the bottom function call shows that `x` has to be proved bounded first. Now the loop consisting of a single state transition due to the top function call shows that to prove `x` bounded, `y` has to be proved bounded first.

Infinite construction is not unbounded construction. It is tempting to try to define a top-down principle for variable boundedness, for example, based on showing that infinite auto-constructions from *each* starting state would be accompanied by infinite size descent. So here is a useful reminder that this does not exclude unbounded (finite) sequences of auto-constructions from different starting states.

In the example below (seen earlier),

```
f x y = if (y = 0) (f 1 x)
                   (f (x + 1) (y ⊖ 1))
```

there cannot be infinite auto-constructions in `x`, since any sequence of auto-constructions in `x` eventually ends with a copy to `y`. However, there do exist unbounded (finite) sequences of auto-constructions in `x` with *different* starting states, during the specialization of `f` with `x` and `y` equal to `1`. The example demonstrates that the absence of infinite auto-constructions from each starting state is insufficient to establish variable boundedness.

6.2.7 Connection with work on termination of other transformations

For completeness, we briefly review some of the works in related areas.

Online partial evaluation. Generally, much more aggressive propagation of static information is possible with online than with offline partial evaluation. For instance, the branches of a conditional are not required to evaluate to results of the same binding-time type in online partial evaluation, so there is no need for some of the casting of static data to dynamic data used to achieve consistent binding-time typing and syntax-directed specialization in offline partial evaluation. Moreover, there exist techniques that ensure the termination of online partial evaluation, while permitting the specialization that is desired. (We will discuss some very liberal strategies later in the section.) It is therefore not out-of-place to remind ourselves some of the *advantages* that offline partial evaluation has over its online counterpart.

- While specialization stages the program computations, offline partial evaluation stages the specialization process. Just as calling a specialized function multiple times during execution increases the savings in execution costs relative to calling the unspecialized version, specializing an annotated function multiple times increases the savings in *transformation costs* relative to specializing the unannotated version. In general, offline partial evaluation achieves a good level of specialization via a fast, syntax-directed specialization phase.
- The program annotations can be used to understand the output of specialization, even though they would not normally be consulted.
- The specialization of an offline specializer to an interpreter (and its annotations) is a stand-alone compiler for the interpreted object language [JGS93]. Neil Jones first demonstrated that the approach can produce an *effective* compiler, which removes the interpretative overhead (due to syntactic dispatch and variable dereference) when applied the object program. Achieving compilation by self-application of an offline specializer is elegant: Only the interpreter (which is, in some sense, the *definition* of the object language) has to be proved correct. Binding-time information for the interpreter is crucial to get effective self-application [Glü91, JGS93].

On the other hand, for termination, an online partial evaluator can in principle always have the same information at its disposal as an offline partial evaluator. In fact, a lot of information that has to be approximated by abstract interpretation for controlling offline specialization can be tracked precisely online. Generally, online partial evaluators make use of online information to achieve much more aggressive specialization than offline partial evaluators. An apparent downside is that online partial evaluators are usually slower than offline ones. A less obvious problem with online partial evaluators has to do with the reason for using them in the first place. This is because generous specialization is *not* guaranteed to improve the subject program: Liberal unfolding can lead to computation duplication, and too much specialization can lead to bloated residual code, which can degrade performance in practice. The issue for online partial evaluation is really how to *control* the amount of specialization. This is difficult. The heuristics that have been proposed vary greatly in complexity, effectiveness and efficiency [JLM96]. The performance of each heuristics also varies across different benchmarks. Selecting the appropriate one to apply can itself be an issue.

Online termination for partial deduction. *Partial deduction*, based on Llyod and Sheperdson's foundational work [LS91], is a well-accepted framework for the (online) specialization of Prolog programs. In partial deduction, a set of incomplete SLDNF-trees is constructed for each atom of a finite set $\mathcal{A}$, based on the semantics of the subject program. The trees are closed, in the following sense: For each atom at a leaf node, a more general atom is found in $\mathcal{A}$. If $\mathcal{A}$ includes the given goal atom, then a specialized program can be extracted from the SLDNF-trees.

To ensure termination for partial deduction, local and global control are imposed. They are explained below.

- Local termination: Initially, the only SLDNF-tree corresponding to some atom of $\mathcal{A}$ is the tree consisting of just that atom at its root. During partial deduction, a leaf of an SLDNF-tree is selected. The tree is then replaced by the result of unfolding that leaf node in every possible way according to the subject program.

  An *unfolding strategy* is used to decide which leaf node (if any) to unfold. Local termination is achieved by ensuring that any branch of an SLDNF-tree permitted under the unfolding strategy is finite.

- Global termination: The atoms of $\mathcal{A}$ can be structured into a global tree. A *whistle* then ensures that the tree (and consequently, the set $\mathcal{A}$) is finite, by directing generalizations to be performed upon the insertion of certain atoms.

  Generalizations are needed to guarantee the finiteness of the global tree, while maintaining closedness. If the comparison between an atom in the global tree and the atom to be inserted triggers the whistle, a usual way to proceed is to specialize the most specific generalizer of the two atoms in question instead.

Usually finiteness of an SLDNF-tree or the global tree is ensured by requiring each branch of atoms to form an *admissible* sequence in a well-founded ordering (wfo), or a well binary relation (wbr). Recall that a sequence of terms $t_0, t_1, t_2, \ldots$ is admissible in a wfo $<$ if for each i, it holds that $t_{i+1} < t_i$; the sequence is admissible in a wbr $\leq$ if for each $i < j$, it holds that $t_i \not\leq t_j$. By definition of wfo and wbr, admissible sequences are finite.

In [Leu98], Leuschel presents a convincing case that wbrs are more appropriate for online termination than wfos. First, there is a wbr with the same admissible sequences as a given wfo. Second, there is a particular wbr, the *homeomorphic embedding*, whose admissible sequences include those of *every* monotonic wfo and simplification ordering. Generally, for terms t and t', t is homeomorphically embedded in t', written $t \trianglelefteq t'$, if t and t' have the same constructor symbol, and each argument of t is homeomorphically embedded in the corresponding argument of t', *or* t is homeomorphically embedded in an argument of t'. As for variables, a simple treatment is to let $t \trianglelefteq t'$ if both t and t' are variables, but more complicated schemes are possible. Informally, t is homeomorphically embedded in t' if t is obtained by "erasing" parts of t', e.g., $[[X],[Y]] \trianglelefteq [[A],[],[B]]$.

In practice, unfolding an SLDNF-tree until the atom at a leaf node is homeomorphically embedded in an ancestor term is probably too aggressive. There is a high risk for the tree to explode, and more seriously, for source-program computations to be duplicated in the residual code [JLM96]. Unfolding strategies that limit the amount of non-determinate unfolding have been demonstrated to work well in experiments [JLM96].

For global control, the atoms in the global tree do not provide important information about the resolution steps leading to the specialization of a particular atom. This information may be crucial to prevent generalizations that suppress the desired specialization. To overcome the problem, *characteristic atoms* can be used in place of atoms in the global tree. A characteristic atom is a pair, consisting of an atom and a *characteristic tree*, which is an abstraction of SLDNF-trees. The whistle can then be triggered when the characteristic atom to be inserted in the global tree is embedded in an ancestor. Less explosive strategies are also possible. For example, the whistle can be triggered when the term size of the atom part is not decreased compared to the atom in the parent node, *and* the characteristic-tree part is embedded in the characteristic tree of an ancestor node [JLM96].

Despite the aggressive specialization that is permitted by various heuristics for online termination, the fact that argument-dependency information is not tracked means that variable boundedness analysis may permit certain specialization suppressed by these methods. For a concrete example, consider the following Prolog program.

```
goal(X) :- p(0,0,0,0,0,X).

p(s(s(s(s(0)))),s(s(s(0))),s(s(0)),s(0),0,_).
p(_,s(s(s(0),s(s(0)),s(0),0,X) :- q(X).
p(A,B,C,D,E,X) :- p(s(B),s(C),s(D),s(E),E,X).

q(X) :- ... //something recursive
```

Suppose that `X` is dynamic. Binding-time analysis determines that the argument positions of `p`, except the final one, are static. Further, variable boundedness analysis determines that `A,B,C,D,E` are bounded, as no auto-construction is possible for them. Consequently, any specialized version of `p` will have one argument, corresponding to the final position of `p` in the source program. ECCE is a system implemented by Leuschel to experiment with different heuristics for the online termination of (conjunctive) partial deduction [Leu96]. With all the heuristics we have tested, the residual code produced by ECCE when `goal(X)` is specialized has the following form.

```
goal1(X) :- p2(0,X).

p2(s(0),_).
p2(A,X) :- q3(X).
p2(A,X).

q3(X) :- ... //essentially q(X)
```

On a second application of partial deduction, the fully specialized program is obtained. The little experiment highlights a couple of points.

1. Online termination does not *automatically* imply more specialization than an offline approach (although we have mentioned that in general, the problem with online termination is not under-specialization, but over-specialization).

2. More seriously, for a non-trivial subject program, and a complicated heuristics for ensuring termination online, it can be mysterious what the quality of partial deduction will be, and how long specialization will take. Can the residual code be improved by repeated specialization?

Conjunctive partial deduction. This extends partial deduction by considering conjunctions of atoms in $\mathcal{A}$, rather than single atoms. The resulting transformation subsumes many others based on the unfold-fold paradigm [BD77], including deforestation [Wad88] and supercompilation [Tur86]. (Note that even plain partial deduction is not simply program specialization. When information is not available for a variable, partial deduction sometimes pursues different bindings for it.)

The methods described before for ensuring the termination of partial deduction can be adapted to ensure the termination of conjunctive partial deduction. First, characteristic atoms are replaced by *characteristic conjunctions*. Generalization for conjunctive partial deduction usually involves more than taking the most specific generalizer of two terms. Various heuristics have been described for *splitting* a conjunction as part of generalization [JLM96].

Controlling the amount of transformation is even more of a challenge for conjunctive partial deduction than for plain partial deduction. Experiments in [JLM96] show that different options for ECCE can drastically affect the effectiveness and efficiency of the transformation (and in different ways for different benchmark programs). For non-trivial programs, it is difficult to gauge the transformation time by inspecting the source program, or the quality of the specialization by inspecting the residual program.

Deforestation, positive supercompilation, supercompilation. Wadler's deforestation [Wad88] aims to remove the use of intermediate data structures automatically from lazy functional programs with pattern-matching variables. The idea is that data constructors that are *produced* while evaluating one part of the program, and subsequently *consumed* (matched) by another part of the program are redundant, and such constructions and deconstructions should be optimized away from the program. Deforestation is typically illustrated using the double-append example: `(append (append xs ys) zs)` traverses `xs` twice; the transformed program, essentially the following L program, traverses `xs` once.

```
dappend xs ys zs = if (xs = nil) (append ys zs)
                      (hd xs) : (dappend (tl xs) ys zs)

append = ...
```

Sørensen et al. [SGJ94] argue that deforestation can partial evaluate, in a way, by consuming constructors of static data at transformation time. Given an expression, the transformation acts on the largest non-constructor subexpressions. Each subexpression with a non-trivial redex (a function call that would be unfolded during ordinary lazy evaluation) is replaced by the call to a new function in the transformed program. The body of this function is the subexpression with the redex unfolded one step. Any constructor of a pattern-matching argument to the redex is used to select among definitions (cases) of the called function. In this way, constructors are "consumed." If some pattern-matching argument is a variable x, then a unique definition cannot be selected. In this case, the transformation is pursued with respect to each possible constructor for the pattern-matching parameter. This accounts for the greater amount of information propagation achieved by deforestation compared to partial evaluation.

However, deforestation does not propagate the information that x is bound to a certain constructor term within each generated definition. Positive supercompilation [SGJ94] extends deforestation by allowing such information propagation. For languages with "else" or "otherwise" constructs, negative information, i.e., x is *not* bound to certain constructor terms, can also be propagated. Turchin's supercompilation [Tur86] propagates both positive and negative information. Among the transformations that achieve linear speed-ups, the more aggressive ones are positive supercompilation, supercompilation and conjunctive partial deduction. For these transformations, termination is usually ensured by online techniques. We only briefly discuss the termination of (positive) supercompilation below, as the ideas are already familiar from the earlier review of partial deduction.

For the above transformations, the notion of *local control* is not applicable, as unfolding is not performed locally. Each unfolding step is accompanied by the generation of a new definition. In [Sør98], the terms acted on by the transformation are structured into a *process tree*, analogous to the global tree for partial deduction. In that work, supercompilation is regarded abstractly as a map from process tree to process tree. It is proved to be terminating by showing that repeated applications of the map, starting with any singleton tree, eventually stabilizes.

The supercompilation uses a whistle, which is triggered when a term to be inserted in the process tree homeomorphically embeds an ancestor term, *and* the two terms agree as to whether a constructor is available for consumption at the redex. The latter requirement corresponds to the use of (very simple) characteristic trees. This is particularly important in the present context, because except at generalizations, a new term in the process tree has been derived from its parent term by unfolding the parent term at the redex. The new term is therefore likely to embed the parent term. Note that the methods for guaranteeing termination *online*, whether for (positive) supercompilation or (conjunctive) partial deduction, are based on the same ideas.

Static analysis for termination of deforestation. The simplest way to ensure the termination of deforestation is by syntactic means. Consult [Chi91] for a treatment of first-order programs, and [Ham96] for a treatment of Hindley-Milner–typeable higher-order programs.

For first-order programs, Sørensen proposed using grammars to approximate *statically* the terms that may be encountered by deforestation [Sør94]. From the computed grammar, *dangerous subterms* are automatically identified: These are the subterms whose results may be assigned to possibly-unbounded parameters (so-called *accumulating parameters*), and subterms that may become embedded in unboundedly deep contexts. By generalizing the program at the dangerous subterms, termination of deforestation can be ensured while allowing more transformation than the syntactic methods mentioned earlier.

In Seidl's reformulation of Sørensen's approach, grammars are dispensed with altogether [Sei96]. Instead, a set of integer constraints is collected. These include constraints that connect a set of integer variables, each representing the depth of context in which some program subterm may occur during deforestation. Other constraints connect integer variables that represent the depth of terms assigned to formal parameters, and integer variables that represent the depth of subterms taking into account the binding of parameters. These constraints involve the operators + and *max*. By expressing them as constraints of the form $Y \geq c + X$ where c is an integer, and using a directed arc $X \rightarrow Y$ to denote the inequality, a solution can be computed by graph techniques. The approach involves computing the strongly-connected components of the digraph due to the constraint arcs, identifying the "increasing" components, and collecting the integer variables reachable from such components. Now the integer variables indicate a set of dangerous subterms. Generalizing the program at these subterms ensures that deforestation will terminate. Such an approach has recently been extended to higher-order programs [SS97].

For ensuring the termination of offline partial evaluation, we will present an approach that is more precise than generalizing *all* accumulating parameters. For example, if it can be deduced that the accumulation in a parameter x is bounded (because for x dependent on x' or $x' = x$, unbounded sequences of auto-constructions in x' would give rise to unbounded sequences of size decreases for some bounded-variable values), then x need not be generalized on account of variable boundedness considerations. It is the purpose of variable boundedness analysis, the object of our study, to determine that certain parameters are bounded, based on known bounded static parameters.

This completes our informal introduction to variable boundedness analysis and broadly related areas. In the following sections, we build a correct specializer for L programs, which is guaranteed to terminate given a procedure for ensuring variable boundedness. The chapter closes with a formal statement of the variable boundedness problem, which is investigated in subsequent chapters. There is nothing theoretically exciting in what follows. The approach to program specialization is roughly based on the offline partial evaluation of Scheme presented in [JGS93]. However, for the specialization of L, whose semantics allows closures to be extended one argument at a time, the details are rather challenging to formalize.

6.3 Annotated syntax

Before an L program is specialized, it is first annotated. These annotations direct the specializer in its choice of evaluation rules. Formally, the process of annotation is a mapping of the subject program p to an *annotated program* $tp \in 2L$. Programs in $2L$ have the following syntax, where $f \in Fun$, $x_i \in Var$ and $op \in Op$ are as for L.

$$\begin{array}{lll}
tp \in \mathit{2Prog} & ::== & td_1 \ \ldots \ td_M \\
td_i \in \mathit{2Def} & ::== & f \ x_1 \ \ldots \ x_N = te^f \\
te, te_i, te^f \in \mathit{2Exp} & ::== & x \mid f^{bt+} \mid (\texttt{if}\ te_1\ te_2\ te_3)^{bt} \mid (op\ te_1\ \ldots\ te_n)^{bt} \\
 & & \mid (te_1\ te_2)^{bt+} \mid (\texttt{lift}\ te) \\
bt+ \in BT+ & ::== & bt \mid \texttt{M} \\
bt \in BT & ::== & \texttt{S} \mid \texttt{D}
\end{array}$$

An annotation is understood as directive for the specializer, indicating whether a program construct (function reference, conditional, operation or application) should be reduced away or residualized. (It is distinct from the *binding-time type* of the program expression.) An annotation of `S` directs the specializer to handle the construct as in ordinary evaluation; an annotation of `D` directs the specializer to act on each subexpression separately; an annotation of `M` directs the specializer to generate a residual-program application. The special directive `lift` converts a *Baseval* value v to a piece of code that evaluates to v at runtime. We use $Bt(x)$ to denote the *binding time (BT) type* of formal parameter x, and $bt(te)$ to denote the BT type of expression te in the subject $2L$ program. The $Bt(x)$ and $bt(e)$ are required to satisfy certain conditions with respect to the annotations in tp (to be discussed later).

Definition 6.3.1 (Annotation-forgetting function ϕ) For $te \in \mathit{2Exp}$, $\phi(te) \in Expr$ is the L expression obtained by dropping all the annotations and `lifts` from *te*. We omit the obvious recursive definition.

For $tp \in 2Prog$, we overload ϕ so that $\phi(tp) \in Prog$ is the L program obtained by dropping all the annotations and `lifts` from *tp*.

Later, *well-annotatedness* requirements will be introduced to ensure that expressions are annotated such that there is always an evaluation rule that applies. A semantic property, *congruence*, is needed to make sure that whenever a static function-call transition or an `M`-annotated application is encountered during specialization, the BT types of the arguments match the BT division. This in turn ensures that expressions always evaluate to data of the expected BT type during specialization, which guards against type errors. The well-annotatedness of *tp* with respect to $Bt(x)$ for $x \in Var$ ensures that the specialization terminates successfully, if it terminates at all. The problem becomes one of finding the well-annotations and BT division that somehow guarantee termination.

Congruence provides one direction of the link between the BT division and the annotations. In the other direction, we have to ensure that the annotations of variables respect the BT division: the following syntactic requirement is the first in a series of *safety requirements* to be specified for the BT division and annotations.

[S1] (Respect of BT division) For all $te \equiv x$, $bt(te) = Bt(x)$.

6.4 Specialization semantics

The specialization semantics for *2L* programs uses *2Val*, a category of *2-level values.*

$$\begin{array}{lll} \zeta_i \in \mathit{2Val} & ::== & bv \mid e \mid k \\ k \in \mathit{2Clo} & ::== & \langle f, \zeta_1, \ldots, \zeta_n \rangle \\ \tau \in \mathit{2Env} & ::== & \varepsilon \mid \tau[x \mapsto \zeta] \end{array}$$

where $f \in Fun$, $e \in Expr$ and bv is in *Baseval*, the set of base values for ordinary evaluation of L programs, which includes the distinguished elements `Err` and `False`. Members of *2Clo* are called *partially static closures* (or simply *static closures*). They have the form $\langle f, \zeta_1, \ldots, \zeta_n \rangle$, where $f \in Fun$ and each $\zeta_i \in 2Val$. The syntactic *specialization environment* τ is treated like a partial function from *Var* to *2Val.*

Definition 6.4.1 (Specialization environment)

1. For $\tau = \varepsilon[x_1 \mapsto \zeta_1] \ldots [x_n \mapsto \zeta_n] \in 2Env$,

$$\tau(x) = \begin{cases} \zeta_i & \text{if } i \text{ is the largest index such that } x_i = x, \\ \texttt{Err} & \text{otherwise.} \end{cases}$$

2. Let $\tau \in 2Env$. Define $\tau[f^{(i)} \mapsto \zeta_i]_{i=1,n}$ to be $\tau[f^{(1)} \mapsto \zeta_1] \ldots [f^{(n)} \mapsto \zeta_n]$.

Definition 6.4.2 (Auxiliary specialization-time functions)

1. The shorthand $error_{bt}$ denotes `Err` if $bt = S$, and an expression whose evaluation always results in a runtime error if $bt = D$.

2. The functions *makeIf*, $makeOp_{op}$, *makeApp* : $Expr^* \to Expr$ make the respective constructs, piecing together the given subexpressions.

3. Function *makeMem* constructs a residual application. It will be known whether at runtime, this application will result in a function-call transition to a residual-program function, or return a closure of the residual-program function.

   A companion function *nextMem* returns all the necessary information, including the specialization environment, for creating the corresponding residual-program function. The definitions of *make-Mem* and *nextMem* are rather involved, and will be treated separately.

4. Function *nextFun* takes a function identifier f of the subject program, and an specialization environment τ, and generates a unique function identifier f' for the residual program, different from all the identifiers in the subject program.

   We assume that the values of f and τ are recoverable from f'. Thus, we will write $f' = \mathit{nextFun}(f, \tau)$ to "extract" the values of f and τ. In practice, f' should be sensitive only to the set of $\tau(x)$ such that $x \in \mathit{Param}(f)$ and x is static, to avoid specialization with respect to irrelevant information.

5. Function *makeDef* takes a function identifier, a formal parameter sequence, and an expression, and composes a residual-program definition.

Generating the residual program.

Definition 6.4.3 An *initial environment* τ_0 satisfies:

$$\begin{array}{ll} \tau_0(x) \in \mathit{Baseval}, & \text{if } x \in \mathit{Param}(f_{\mathit{initial}}) \text{ and } Bt(x) = S, \\ \tau_0(x) = x, & \text{if } x \in \mathit{Param}(f_{\mathit{initial}}) \text{ and } Bt(x) = D, \\ \tau_0(x) = \mathit{error}_{Bt(x)}, & \text{for } x \notin \mathit{Param}(f_{\mathit{initial}}). \end{array}$$

Definition 6.4.4 By "$\mathit{Sp}; \tau; \mathit{te} \downarrow \zeta$" (read "$\mathit{Sp}; \tau; \mathit{te}$ *specializes to* ζ"), we mean that $\mathit{Sp}; \tau; \mathit{te} \downarrow \zeta$ is provable using the evaluation rules of Fig. 6.3 and 6.4.

Note that $\downarrow$ is dependent on the subject *2L* program *tp*, so strictly, it should be written as $\downarrow_{tp}$. For succinctness, we sometimes write "$\mathit{Sp}; \tau; \mathit{te} \downarrow \zeta = \langle f, \xi_1, \ldots, \xi_n \rangle$" to mean "$\mathit{Sp}; \tau; \mathit{te} \downarrow \zeta$ where $\zeta = \langle f, \xi_1, \ldots, \xi_n \rangle$," or "$\mathit{Sp}; \tau; \mathit{te} \downarrow \zeta \neq \mathtt{Err}$" to mean "$\mathit{Sp}; \tau; \mathit{te} \downarrow \zeta$ where $\zeta \neq \mathtt{Err}$."

Figure 6.3 specifies the specialization of constructs other than those resulting in function applications. Figure 6.4 handles the remaining constructs and the `lift` operator. The side conditions given in brackets are *not* checked during specialization. However, using the results of semantic analysis, it will be arranged so that the program annotations allow an evaluation rule to be applied *only if* the side conditions hold. The reader may worry that the specialization semantics is careless with respect to non-termination and error behaviour in the subject program. These concerns will be addressed by semantic analysis. The proof rules appear complicated, but they specify the intuitive actions to take when coming across an expression annotated in a certain way. For static expressions, these actions coincide with ordinary L evaluation.

Definition 6.4.5 (Specialization of *tp* with τ_0) Let τ_0 be a given initial environment.

1. Elements of *Sp* are known as *specialization configurations.*

   Let $x_1, \ldots, x_n$ be the parameters of f_{initial}, and $x_{j_1}, \ldots, x_{j_d}$ be those classified dynamic by Bt. Assume that $j_1, \ldots, j_d$ are in increasing order. Let $f'_0 = \mathit{nextFun}(f_{\mathit{initial}}, \tau_0)$, and $\mathit{argv}'_0 = x_{j_1} \ldots x_{j_d}$. Then the specialization configuration $(f'_0, \mathit{argv}'_0, f_{\mathit{initial}}, \tau_0)$ is required to be in *Sp*. Other configurations in *Sp* are determined by the specialization of $\mathit{Sp}; \tau; \mathit{te}^f \downarrow \zeta$ for $(f', \mathit{argv}', f, \tau)$ in *Sp*, carried out according to the evaluation rules of Fig. 6.3 and 6.4. The pair $\tau; \mathit{te}^f$ is the *specializer state* corresponding to the configuration $(f', \mathit{argv}', f, \tau)$.

2. The *specialization of tp with* τ_0 is:

$$p' = \{\mathit{makeDef}(f', \mathit{argv}', \zeta) \mid (f', \mathit{argv}', f, \tau) \in \mathit{Sp} \text{ and } \mathit{Sp}; \tau; \mathit{te}^f \downarrow \zeta\},$$

The enumerability of $\downarrow$ rules, and of p' and *Sp* implies a semi-decidable procedure for obtaining a specialization of *tp* with τ_0, if one exists. Of course, one would not appeal to exhaustive enumeration in practice.

Lemma 6.4.6 $\mathit{Sp}; \tau; \mathit{te} \downarrow \zeta$ and $\mathit{Sp}; \tau; \mathit{te} \downarrow \zeta'$ implies that $\zeta = \zeta'$. Further, the proof tree for $\mathit{Sp}; \tau; \mathit{te}$ is unique, if it exists.

$$\frac{\zeta = \tau(x)}{Sp;\tau;x \downarrow \zeta} \quad (1)$$

$$\frac{}{Sp;\tau;f^{\mathtt{S}} \downarrow \langle f \rangle} \quad (2)$$

$$\frac{Sp;\tau;te_1 \downarrow \mathtt{Err} \quad te \equiv (\mathtt{if}\ te_1\ te_2\ te_3)^{\mathtt{S}}}{Sp;\tau;te \downarrow error_{bt(te)}} \quad (3)$$

$$\frac{Sp;\tau;te_1 \downarrow \mathtt{False} \quad Sp;\tau;te_3 \downarrow \zeta}{Sp;\tau;(\mathtt{if}\ te_1\ te_2\ te_3)^{\mathtt{S}} \downarrow \zeta} \quad (4)$$

$$\frac{Sp;\tau;te_1 \downarrow \zeta_1 \quad \zeta_1 \notin \{\mathtt{Err},\mathtt{False}\} \quad Sp;\tau;te_2 \downarrow \zeta}{Sp;\tau;(\mathtt{if}\ te_1\ te_2\ te_3)^{\mathtt{S}} \downarrow \zeta} \quad (5)\ [\zeta_1 \notin \mathit{2Clo}]$$

$$\frac{Sp;\tau;te_1 \downarrow \zeta_1 \quad Sp;\tau;te_2 \downarrow \zeta_2 \quad Sp;\tau;te_3 \downarrow \zeta_3 \quad e = \mathit{makeIf}(\zeta_1,\zeta_2,\zeta_3)}{Sp;\tau;(\mathtt{if}\ te_1\ te_2\ te_3)^{\mathtt{D}} \downarrow e} \quad (6)$$

$$\frac{Sp;\tau;te_1 \downarrow \zeta_1 \quad \ldots \quad Sp;\tau;te_n \downarrow \zeta_n \quad \nexists v : v = \mathcal{O}[\![op]\!](\zeta_1,\ldots,\zeta_n)}{Sp;\tau;(op\ te_1\ \ldots\ te_n)^{\mathtt{S}} \downarrow \mathtt{Err}} \quad (7)\ [\zeta_i \notin \mathit{2Clo}]$$

$$\frac{Sp;\tau;te_1 \downarrow \zeta_1 \quad \ldots \quad Sp;\tau;te_n \downarrow \zeta_n \quad v = \mathcal{O}[\![op]\!](\zeta_1,\ldots,\zeta_n)}{Sp;\tau;(op\ te_1\ \ldots\ te_n)^{\mathtt{S}} \downarrow v} \quad (8)\ [\zeta_i \notin \mathit{2Clo}]$$

$$\frac{Sp;\tau;te_1 \downarrow \zeta_1 \quad \ldots \quad Sp;\tau;te_n \downarrow \zeta_n \quad e = \mathit{makeOp}_{op}(\zeta_1,\ldots,\zeta_n)}{Sp;\tau;(op\ te_1\ \ldots\ te_n)^{\mathtt{D}} \downarrow e} \quad (9)$$

Figure 6.3: Specialization rules for *2L*– Part 1

$$\frac{Sp;\tau;te_1 \downarrow \zeta_1 \quad \zeta_1 \notin \mathit{2Clo} \quad te \equiv (te_1\ \ te_2)^{\mathtt{S}}}{Sp;\tau;te \downarrow error_{bt(te)}} \quad (10)$$

$$\frac{Sp;\tau;te_1 \downarrow \langle f,\xi_1,\ldots,\xi_n\rangle \quad Sp;\tau;te_2 \downarrow \mathtt{Err} \quad te \equiv (te_1\ \ te_2)^{\mathtt{S}}}{Sp;\tau;te \downarrow error_{bt(te)}} \quad (11)$$

$$\frac{Sp;\tau;te_1 \downarrow \langle f,\xi_1,\ldots,\xi_n\rangle \quad Sp;\tau;te_2 \downarrow \xi_{n+1} \quad \xi_{n+1} \neq \mathtt{Err} \quad n \neq ar(f)-1}{Sp;\tau;(te_1\ \ te_2)^{\mathtt{S}} \downarrow \langle f,\xi_1,\ldots,\xi_{n+1}\rangle} \quad (12)$$

$$\frac{\begin{array}{r}Sp;\tau;te_1 \downarrow \langle f,\xi_1,\ldots,\xi_n\rangle \quad Sp;\tau;te_2 \downarrow \xi_{n+1} \quad \xi_{n+1} \neq \mathtt{Err} \quad n = ar(f)-1\\ \tau_1 = \tau[f^{(i)} \mapsto \xi_i]_{i=1,ar(f)} \quad Sp;\tau_1;te^f \downarrow \zeta\end{array}}{Sp;\tau;(te_1\ \ te_2)^{\mathtt{S}} \downarrow \zeta} \quad (13)$$

$$\frac{Sp;\tau;te_1 \downarrow \zeta_1 \quad Sp;\tau;te_2 \downarrow \zeta_2 \quad e = makeApp(\zeta_1,\zeta_2)}{Sp;\tau;(te_1\ \ te_2)^{\mathtt{D}} \downarrow e} \quad (14)$$

$$\frac{Sp;\tau;te \downarrow \zeta \quad e \text{ always evaluates to } \zeta}{Sp;\tau;(\mathtt{lift}\ te) \downarrow e} \quad (15)$$

$$\frac{Sp;\tau;te_1 \downarrow \zeta_1 \quad \zeta_1 \notin \mathit{2Clo}}{Sp;\tau;(te_1\ \ te_2)^{\mathtt{M}} \downarrow error_{\mathtt{D}}} \quad (16)$$

$$\frac{Sp;\tau;te_1 \downarrow \langle \xi_1,\ldots,\xi_n\rangle \quad Sp;\tau;te_2 \downarrow \mathtt{Err}}{Sp;\tau;(te_1\ \ te_2)^{\mathtt{M}} \downarrow error_{\mathtt{D}}} \quad (17)$$

$$\frac{\begin{array}{r}Sp;\tau;te_1 \downarrow \langle f,\xi_1,\ldots,\xi_n\rangle \quad Sp;\tau;te_2 \downarrow \xi_{n+1} \quad \xi_{n+1} \neq \mathtt{Err} \quad \tau_1 = \tau[f^{(i)} \mapsto \xi_i]_{i=1,n+1}\\ nextMem(f,\tau_1) \in Sp \quad e = makeMem(f,\tau_1,n+1)\end{array}}{Sp;\tau;(te_1\ \ te_2)^{\mathtt{M}} \downarrow e} \quad (18)$$

$$\frac{nextMem(f,\tau) \in Sp \quad e = makeMem(f,\tau,0)}{Sp;\tau;f^{\mathtt{M}} \downarrow e} \quad (19)$$

Figure 6.4: Specialization rules for *2L*– Part 2

Proof By induction on the size of the proof tree. We omit the details, and simply note that given the uniqueness of subcomputation trees, and subcomputation results, at most one rule is applicable for $Sp; \tau; te$, so its proof tree and result are unique.

The lemma means that the following definition makes sense.

Definition 6.4.7 (*Sp* set)

1. Let $nextSp(f, \tau)$ be the minimal set of specialization configurations that must be included in Sp to evaluate $Sp; \tau; te^f$ successfully (when possible at all); otherwise, leave $nextSp(f, \tau)$ undefined.

2. Define Sp_i inductively as follows.

$$
\begin{array}{lcl}
Sp_0 & = & \{(f'_0, argv'_0, f_{initial}, \tau_0)\} \quad \text{(as described earlier)} \\
Sp_{i+1} & = & Sp_i \cup \{sp \mid sp \in nextSp(f, \tau) \text{ and } (f', argv', f, \tau) \in Sp_i\}
\end{array}
$$

Thus if a specialization of *tp* with τ_0 exists, we may take it to be the residual program corresponding to the minimal specialization configurations $Sp^* = \bigcup_i Sp_i$; otherwise Sp^* is infinite or undefined.

One way to specialize *tp* with τ_0 is to begin with the evaluation of $\tau_0; te^{f_{initial}}$. For deterministic rules such as those of Fig. 6.3 and 6.4, evaluation can be carried out by using placeholders to represent uncomputed results, and extending the evaluation tree depth-first, left-to-right, instantiating the placeholders by unification when a leaf node is discharged [Jon99].

For leaf nodes specifying specialization configurations for *Sp*, we check whether the residual function name already exists in a configuration of (the current) *Sp*. If not, the configuration is inserted and marked as unfollowed. Note that expanding *Sp* does not disturb the proof tree in any way, except to allow the discharge of "$\in Sp$" predicates. (This justifies excluding the *Sp* component when specifying an evaluation state.) Upon completing an evaluation and forming a definition, other unfollowed configurations in *Sp*, if they exist, are considered, leading to further evaluation. If the process terminates successfully, we have a specialization of *tp* with τ_0 corresponding to the minimal configurations Sp^*. As *Sp* is not significant for the computation corresponding to an evaluation state, we henceforth write "$\tau; te \downarrow \zeta$" to mean "$Sp; \tau; te \downarrow \zeta$" *for some appropriate Sp* (that discharges all "$\in Sp$" requirements for a successful computation).

Arity raising. Apart from the initial configuration, every specialization configuration *sp* involves the generation of a residual function application. At runtime, this application either results in a call to some residual function f' corresponding to a function f of the subject program, or evaluates to an f' closure. The complete treatment of *sp* entails generating a definition for f'.

In the simple case, all the static arguments for f are base values. Then the formal arguments for f' are simply the dynamic parameters of f, and the residual function application and definition for f' can be generated in a straightforward way. On the other hand, if (partially static) closure arguments are involved, the situation is considerably more complicated. Such closures contain expressions, which may need to appear in the body of f'. These expressions may refer to dynamic variables of the current function being specialized. Any possibly referenced variable has to be incorporated in the argument list of the residual function application, so that its runtime value is accessible when an expression referring to it is finally evaluated.

The definition of f' must use fresh identifiers to capture the values of these variables to avoid name clashes in its formal arguments. And any expression embedded in a *2Clo* argument must be updated to refer to the new parameters. The entire process is described as *arity raising.* The idea is to ensure that embedded expressions evaluate to the expected results in the *runtime environment* of f'. Only then are the static closures valid for the evaluation generating the body of f'. Finally, we must be careful that the variable renaming scheme does not lead to unboundedly many alpha-equivalent definitions in the residual program. We require a few definitions to proceed.

Definition 6.4.8 (Auxiliary specialization-time functions)

1. Let $\mathit{newvar}_1, \mathit{newvar}_2, \ldots$ be a supply of variable identifiers distinct from any identifier in the subject program, and any identifier generated by *nextFun*.

2. $FVE(e)$ is the set of free variables of the expression e, and $FVV(\zeta)$ is the set of free variables of those expressions embedded in $\zeta \notin \mathit{Expr}$.

$$\begin{array}{lcl} FVE(x) & = & \{x\} \\ FVE(f) & = & \{\} \\ FVE(e) & = & \bigcup_{e' \text{ subexpr of } e} FVE(e'), \text{ for any other } e \end{array}$$

$$\begin{array}{lcl} FVV(\zeta) & = & \{\} \text{ if } \zeta \notin \mathit{2Clo} \\ FVV(\langle f, \xi_1, \ldots, \xi_n \rangle) & = & (\bigcup_{Bt(f^{(i)})=S} FVV(\xi_i)) \cup (\bigcup_{Bt(f^{(i)})=D} FVE(\xi_i)) \end{array}$$

Functions FVE and FVV are not defined outside their respective domains. In practical terms, since *2Val* elements cannot be tested for membership in *Expr* (due to the offline nature of our partial evaluation), the behaviour of FVE and FVV on unexpected inputs is unpredictable. It has to be *proved* that unexpected inputs to these functions do not occur during any specialization. On success, FVE and FVV return a set of parameter identifiers.

3. Function $\mathit{SubE}(e, \vec{x}, \vec{y})$ performs parallel substitution of $(\vec{x})_i$ with $(\vec{y})_i$ in expression e. The function $\mathit{SubV}(\zeta, \vec{x}, \vec{y})$ does the same for $\zeta \notin \mathit{Expr}$. They are not defined outside their respective domains.

$$\begin{array}{lcl} \mathit{SubE}(z, \vec{x}, \vec{y}) & = & (\vec{y})_i \text{ if } z = (\vec{x})_i \\ \mathit{SubE}(f, \vec{x}, \vec{y}) & = & f \\ \mathit{SubE}((\texttt{if}\ e_1\ e_2\ e_3), \vec{x}, \vec{y}) & = & \mathit{makeIf}(\mathit{SubE}(e_1, \vec{x}, \vec{y}), \mathit{SubE}(e_2, \vec{x}, \vec{y}), \mathit{SubE}(e_3, \vec{x}, \vec{y})) \\ \mathit{SubE}((op\ e_1\ \ldots\ e_n), \vec{x}, \vec{y}) & = & \mathit{makeOp}_{op}(\mathit{SubE}(e_1, \vec{x}, \vec{y}), \ldots, \mathit{SubE}(e_n, \vec{x}, \vec{y})) \\ \mathit{SubE}((e_1\ e_2), \vec{x}, \vec{y}) & = & \mathit{makeApp}(\mathit{SubE}(e_1, \vec{x}, \vec{y}), \mathit{SubE}(e_2, \vec{x}, \vec{y})) \end{array}$$

$$\begin{array}{lcl} \mathit{SubV}(\zeta, \vec{x}, \vec{y}) & = & \zeta \text{ if } \zeta \notin \mathit{2Clo} \\ \mathit{SubV}(\langle f, \xi_1, \ldots, \xi_n \rangle, \vec{x}, \vec{y}) & = & \langle f, \xi'_1, \ldots, \xi'_n \rangle \\ & & \text{where } \xi'_i = \begin{cases} \mathit{SubV}(\xi_i, \vec{x}, \vec{y}), & \text{if } Bt(f^{(i)}) = S, \\ \mathit{SubE}(\xi_i, \vec{x}, \vec{y}), & \text{if } Bt(f^{(i)}) = D. \end{cases} \end{array}$$

4. The following is the procedure to compute the residual application $\mathit{makeMem}(f, \tau, n)$.

   Let $f^{(i_1)}, \ldots, f^{(i_s)}$ be the static arguments of f with $i_1 < \ldots < i_s \leq n$. We refer to the i_k as *static indices*, and denote them by I_S when f and n are clear from context. Let $f^{(j_1)}, \ldots, f^{(j_d)}$ be the dynamic arguments of f with $j_1 < \ldots < j_d \leq n$. We refer to the j_k as *dynamic indices*, and denote them by I_D.

   Let $\bigcup_k FVV(\tau(f^{(i_k)})) = \{f^{(fv_1)}, \ldots, f^{(fv_m)}\}$ where $fv_1 < \ldots < fv_m$ be the free dynamic variables within the partially static f arguments of τ, and let $f' = \mathit{nextFun}(f, \tau)$. (In practice, *nextFun* should be sensitive only to $\tau(f^{(i)})$ for $i \in I_S$.) Then $\mathit{makeMem}(f, \tau, n)$ is the following application: $(f'\ f^{(fv_1)}\ \ldots\ f^{(fv_m)}\ \tau(f^{(j_1)})\ \ldots\ \tau(f^{(j_d)}))$.

   Special case: $m = d = 0$ and $n = ar(f)$. In this case, the residual function would have no parameters. Since this is not allowed in L, a dummy argument will be created, to which is bound an arbitrary value in the residual application: e.g., $(f'\ \texttt{nil})$.

5. The companion function $\mathit{nextMem}(f, \tau)$ is described next. Let $f^{(i_1)}, \ldots, f^{(i_s)}$ be the static arguments of f with $i_1 < \ldots < i_s \leq ar(f)$. Let $f^{(j_1)}, \ldots, f^{(j_d)}$ be the dynamic arguments of f with $j_1 < \ldots < j_d \leq ar(f)$.

   Let $\bigcup_k FVV(\tau(f^{(i_k)})) = \{f^{(fv_1)}, \ldots, f^{(fv_m)}\}$ where $fv_1 < \ldots < fv_m$ be the free dynamic variables occurring within the partially static f arguments of τ. Define the sequences $\vec{fv} = f^{(fv_1)} \ldots f^{(fv_m)}$

and $\vec{nv} = newvar_1 \ldots newvar_m$. Then $nextMem(f, \tau)$ returns the configuration $(f', argv', f, \overline{\tau})$ where $f' = nextFun(f, \tau)$, $argv'$ is the parameter list $newvar_1 \ \ldots \ newvar_m \ f^{(j_1)} \ \ldots f^{(j_d)}$, and $\overline{\tau}$ is extended from τ with new bindings for the parameters of f. The effect of these bindings is:

$$\begin{array}{lcl} \overline{\tau}(f^{(i_k)}) & = & SubV(\tau(f^{(i_k)}), \vec{fv}, \vec{nv}), \text{ for } k = 1, \ldots, s, \\ \overline{\tau}(f^{(j_k)}) & = & f^{(j_k)}, \text{ for } k = 1, \ldots, d, \\ \overline{\tau}(x) & = & \tau(x), \text{ for } x \notin Param(f). \end{array}$$

Naturally, if a definition has been generated for f', then $(f', argv', f, \overline{\tau})$ does not have to be entered into *Sp*; the definition of f' can be "shared." *Special consideration*: If $argv'$ turns out to be empty, use the list with a single parameter $newvar_1$. (In proofs, we write $\vec{fv}$, $\vec{nv}$ and $\overline{\tau}$ without further qualification when τ and f are clear from context.)

Further remarks about the representation of 2Clo: Consistent representation of partially static closures is important. Closures are used in a textual way to generate residual function names, which are compared against one another. The renaming of dynamic variables occurring in a closure's arguments should not lead to the creation of unboundedly many alpha-equivalent definitions during specialization. To ensure this, it is only necessary that the fresh variables used for treating closures of bounded size form a bounded set. Our renaming scheme is certainly viable.

This completes the presentation of the specialization of *2L* programs. Next, we consider properties of the specialization. Specifically, we are interested in its termination, type-safety (this goes to whether a residual program is generated whenever the process terminates), and correctness with respect to the semantics of the source program.

6.5 Termination of specialization

We define two generally non-computable relations $\overset{S}{\hookrightarrow}$ and $\overset{D}{\hookrightarrow}$, to help us reason about termination and variable boundedness. A $\overset{S}{\hookrightarrow}$ *transition* $\tau_0; te_0 \overset{S}{\hookrightarrow} \tau_1; te_1$ signifies that the evaluation state $\tau_1; te_1$ is a *subcomputation* following from $\tau_0; te_0$. Every subcomputation is represented in the $\overset{S}{\hookrightarrow}$ relation. In a finite proof tree, the states at the roots of subtrees in the proof of $\tau_0; te_0$ are precisely those $\tau_1; te_1$ such that $\tau_0; te_0 \overset{S}{\hookrightarrow} \tau_1; te_1$. We frequently use the term "subcomputation" in this sense.

The $\overset{D}{\hookrightarrow}$ relation connects evaluation states that follow from `M`-annotated expressions, i.e., if $\tau_0; te_0$ is successful and leads to the specialization configuration $(f', argv', f_1, \tau_1)$, then the transition from $\tau_0; te_0$ to τ_1, te^{f_1} appears in the $\overset{D}{\hookrightarrow}$ relation. We will define $\hookrightarrow$ to be the union of $\overset{S}{\hookrightarrow}$ and $\overset{D}{\hookrightarrow}$. The relation $\overset{S}{\hookrightarrow}$ contains information about individual evaluations, and is suitable for discussing unfolding, while $\hookrightarrow$ contains information about all the evaluation states that follow from some initial state, and is suitable for discussing variable boundedness.

Definition 6.5.1 ($\overset{S}{\hookrightarrow}, \overset{D}{\hookrightarrow}, \hookrightarrow$) We write $st \overset{S}{\hookrightarrow} st_1, \ st_2, \ \ldots$ as shorthand for $st \overset{S}{\hookrightarrow} st_1$ and $st \overset{S}{\hookrightarrow} st_2 \ldots$.

1. $\overset{S}{\hookrightarrow}$ is defined as follows.

   - $\tau; (\texttt{if}\ te_1\ te_2\ te_3)^{\texttt{S}} \overset{S}{\hookrightarrow} \tau; te_1,\ \tau; te_2$ if $\tau; te_1 \downarrow \zeta_1$, and $\zeta_1 \notin \{\texttt{Err}, \texttt{False}\}$.
   - $\tau; (\texttt{if}\ te_1\ te_2\ te_3)^{\texttt{S}} \overset{S}{\hookrightarrow} \tau; te_1,\ \tau; te_3$ if $\tau; te_1 \downarrow \texttt{False}$.
   - $\tau; (\texttt{if}\ te_1\ te_2\ te_3)^{\texttt{S}} \overset{S}{\hookrightarrow} \tau; te_1$ if the previous cases do not apply.
   - $\tau; (\texttt{if}\ te_1\ te_2\ te_3)^{\texttt{D}} \overset{S}{\hookrightarrow} \tau; te_1,\ \tau; te_2,\ \tau; te_3$.
   - $\tau; (op\ te_1\ \ldots\ te_n)^{bt} \overset{S}{\hookrightarrow} \tau; te_i$ for each $i = 1, \ldots, n$ and $bt \in \{\texttt{S}, \texttt{D}\}$.

- $\tau; (te_1\ \ te_2)^{\mathtt{S}} \overset{S}{\hookrightarrow} \tau; te_1,\ \tau; te_2,\ \tau_1; te^f$, where $\tau_1 = \tau[f^{(i)} \mapsto \xi_i]_{i=1,ar(f)}$ if $\tau; te_1 \downarrow \langle f, \xi_1, \ldots, \xi_{ar(f)-1}\rangle$, and $\tau; te_2 \downarrow \xi_{ar(f)} \neq \mathtt{Err}$.

  Call $\tau; (te_1\ \ te_2)^{\mathtt{S}} \overset{S}{\hookrightarrow} \tau_1; te^f$ an *unfolding transition* to f.
- $\tau; (te_1\ \ te_2)^{\mathtt{S}} \overset{S}{\hookrightarrow} \tau; te_1,\ \tau; te_2$ if $\tau; te_1 \downarrow \zeta_1 \in \mathit{2Clo}$, but the previous case does not apply.
- $\tau; (te_1\ \ te_2)^{\mathtt{S}} \overset{S}{\hookrightarrow} \tau; te_1$ if the previous cases do not apply.
- $\tau; (te_1\ \ te_2)^{\mathtt{D}} \overset{S}{\hookrightarrow} \tau; te_1,\ \tau; te_2$.
- $\tau; (\mathtt{lift}\ te) \overset{S}{\hookrightarrow} \tau; te$.
- $\tau; (te_1\ \ te_2)^{\mathtt{M}} \overset{S}{\hookrightarrow} \tau; te_1,\ \tau; te_2$ if $\tau; te_1 \downarrow \zeta_1 \in \mathit{2Clo}$.
- $\tau; (te_1\ \ te_2)^{\mathtt{M}} \overset{S}{\hookrightarrow} \tau; te_1$ if the previous case does not apply.
- $\tau; x$ and $\tau; f^{\mathtt{S}}$ and $\tau; f^{\mathtt{M}}$ have no subcomputations.

2. $\overset{D}{\hookrightarrow}$ contains the following *memoization transitions* to f.

   - $\tau; (te_1\ \ te_2)^{\mathtt{M}} \overset{D}{\hookrightarrow} \tau_1; te^f$, for $\tau; te_1 \downarrow \langle f, \xi_1, \ldots, \xi_n\rangle$ and $\tau; te_2 \downarrow \xi_{n+1} \neq \mathtt{Err}$, if function $\mathit{nextMem}(f, \tau')$ returns $(f', \mathit{argv}', f, \tau_1)$, where $\tau' = \tau[f^{(i)} \mapsto \xi_i]_{i=1,n+1}$.
   - $\tau; f^{\mathtt{M}} \overset{D}{\hookrightarrow} \tau_1; te^f$, if $\mathit{nextMem}(f, \tau)$ returns $(f', \mathit{argv}', f, \tau_1)$.

3. Let $\hookrightarrow\ =\ \overset{S}{\hookrightarrow} \cup \overset{D}{\hookrightarrow}$.

4. The reflexive transitive closures of $\overset{S}{\hookrightarrow}$ and $\hookrightarrow$ are denoted $\overset{S}{\hookrightarrow}^*$ and $\hookrightarrow^*$ respectively. The transitive closures of $\overset{S}{\hookrightarrow}$ and $\hookrightarrow$ are denoted $\overset{S}{\hookrightarrow}^+$ and $\hookrightarrow^+$ respectively.

5. $\tau_1; te_1$ is *reachable* from $\tau_0; te_0$ if $\tau_0; te_0 \hookrightarrow^* \tau_1; te_1$. An evaluation state is reachable if it is reachable from some state $\tau_0; te^{f_{initial}}$, where τ_0 is an initial environment. We also call $\tau_0; te^{f_{initial}}$ an *initial state*.

The variable boundedness proposition.

Definition 6.5.2 The *variable boundedness proposition* is:

> For any initial environment τ_0, $\{\tau; te \mid \tau; te \text{ is reachable from } \tau_0; te^{f_{initial}}\}$ is finite.

In words, the set of all evaluation states encountered during any specialization is finite. In particular, the set of states corresponding to configurations in Sp^* (if it exists) is finite, so the only way for specialization to diverge is for the evaluation of some state in Sp^* to fail to terminate.

The finite unfolding proposition.

Definition 6.5.3 The *finite unfolding proposition* is:

> For any reachable state $\tau; te$, there does not exist any infinite chain of the following form: $\tau; te \overset{S}{\hookrightarrow} \tau_1; te_1 \overset{S}{\hookrightarrow} \ldots \overset{S}{\hookrightarrow} \tau_{t+1}; te_{t+1} \overset{S}{\hookrightarrow} \ldots$.

As we shall see, this forces the computation of any reachable evaluation state to terminate. The purpose of our study is to consider how the above propositions can be satisfied. Before embarking on this task, we investigate the question of correctness, *assuming* that the propositions hold.

6.6 Well-annotatedness requirements

We have already noted a syntactic requirement linking the BT division and the annotations.

[S1] (Respect of BT division) For *2L* expression $te \equiv x$, $bt(te) = Bt(x)$.

Below are other natural requirements on the BT division and annotations. Generally, they are needed to ensure that specialization does not experience type errors, or get stuck in any other way (e.g., a state is reached such that no rule applies). Ignore for now how these requirements are satisfied.

[S2] (Consistent annotations) In order for specialization not to become stuck, expressions must have *consistent annotations*. One of the following must hold.

- $te \equiv x$ and $bt(te) \in \{S, D\}$.
- $te \equiv f^{\mathtt{S}}$ and $bt(te) = S$.
- $te \equiv (\mathtt{if}\ te_1\ te_2\ te_3)^{\mathtt{S}}$, $bt(te_1) = S$ and $bt(te) = bt(te_2) = bt(te_3)$.
- $te \equiv (\mathtt{if}\ te_1\ te_2\ te_3)^{\mathtt{D}}$ and $bt(te) = bt(te_1) = bt(te_2) = bt(te_3) = D$.
- $te \equiv (op\ te_1\ \ldots\ te_n)^{\mathtt{S}}$ and $bt(te) = bt(te_1) = \ldots = bt(te_n) = S$.
- $te \equiv (op\ te_1\ \ldots\ te_n)^{\mathtt{D}}$ and $bt(te) = bt(te_1) = \ldots = bt(te_n) = D$.
- $te \equiv (te_1\ te_2)^{\mathtt{S}}$ and $bt(te_1) = S$.
- $te \equiv (te_1\ te_2)^{\mathtt{D}}$ and $bt(te) = bt(te_1) = bt(te_2) = D$.
- $te \equiv (\mathtt{lift}\ te_1)$, $bt(te_1) = S$ and $bt(te) = D$.
- $te \equiv (te_1\ te_2)^{\mathtt{M}}$, $bt(te_1) = S$ and $bt(te) = D$.
- $te \equiv f^{\mathtt{M}}$ and $bt(te) = D$.

[S3] (Delaying conditionals) Referring to the rules of Fig. 6.3, a side condition would be violated if the condition of a static conditional evaluates to an element of *2Clo*. This must be prevented.

Let $\tau; (\mathtt{if}\ te_1\ te_2\ te_3)^{\mathtt{S}}$ be reachable. If $\tau; te_1 \downarrow \zeta_1$ then $\zeta_1 \notin \mathit{2Clo}$.

Semantic requirements of this form are enforced using semantic analysis to anticipate whether te_1 may evaluate to a closure.

[S4] (Delaying operations) Referring to the rules of Fig. 6.3, a side condition would be violated if an argument of a static operation evaluates to an element of *2Clo*. This must be prevented.

Let $\tau; (op\ te_1 \ldots\ te_n)^{\mathtt{S}}$ be reachable. For all i, if $\tau; te_i \downarrow \zeta_i$ then $\zeta_i \notin \mathit{2Clo}$.

[S5] (Closures in dynamic context) Inspecting the rules of Fig. 6.4, the result of a static application is static *unless* the application always results in an unfolding transition. This consideration leads to the following semantic requirement.

Let $\tau; (te_1\ te_2)^{\mathtt{S}}$ be reachable. If $\tau; te_1 \downarrow \langle f, \xi_1, \ldots, \xi_n \rangle$, and $bt(te) = D$ for $te \equiv (te_1\ te_2)^{\mathtt{S}}$, then $n = ar(f) - 1$.

[S6] (Lift) By the evaluation rule for `lift` expressions, and the rule for `lift`, specialization cannot proceed if the item lifted is anything but a base value. This leads to the following semantic requirement.

Let $\tau; (\mathtt{lift}\ te)$ be reachable. If $\tau; te \downarrow \zeta$ then $\zeta \notin \mathit{2Clo}$.

[S7] (Congruence) This semantic requirement ensures that for any reachable state $\tau; te$, if τ respects the BT division, then any subsequent update to the environment (of the form $\tau[f^{(i)} \mapsto \xi_i]_i$) does not destroy this property. This prevents specialization from becoming stuck due to type errors.

Let $\tau; (te_1\ \ te_2)^{\mathtt{S}}$ be reachable. If $\tau; te_1 \downarrow \langle f, \xi_1, \ldots, \xi_n \rangle$ where $0 \leq n < ar(f)$, then $bt(te_2) = Bt(f^{(n+1)})$. And if $n = ar(f) - 1$ then $bt(te) = bt(te^f)$ where $te \equiv (te_1\ \ te_2)^{\mathtt{S}}$.

[S8] (Dynamic closures) This semantic requirement ensures that for a reachable state of the form $\tau; (te_1\ \ te_2)^{\mathtt{M}}$, if te_1 evaluates to an f closure with n arguments, then $f^{(n+2)}, \ldots, f^{(ar(f))}$ are dynamic. This makes intuitive sense: For $n + 1 < ar(f)$, the residual function application evaluates to a closure at runtime, so the arguments for completing it must be dynamic. In the same vein, if $\tau; f^{\mathtt{M}}$ is reachable, then all f parameters must be dynamic.

Let $\tau; (te_1\ \ te_2)^{\mathtt{M}}$ be reachable. If $\tau; te_1 \downarrow \langle f, \xi_1, \ldots, \xi_n \rangle$, then $Bt(f^{(n+2)}) = \ldots = Bt(f^{(ar(f))}) = D$. Let $\tau; f^{\mathtt{M}}$ be reachable. Then $Bt(f^{(1)}) = \ldots = Bt(f^{(ar(f))}) = D$.

[S9] (Dynamic definitions) The final semantic requirement is to make sure that *makeDef* is given an expression, not a base value or closure, to compose a new definition. This is expected by the definition of the residual program p'.

First, $bt(te^{f_{initial}}) = D$. For $\tau; (te_1\ \ te_2)^{\mathtt{M}}$ reachable, if $\tau; te_1 \downarrow \langle f, \xi_1, \ldots, \xi_n \rangle$, then we have $bt(te^f) = D$. For $\tau; f^{\mathtt{M}}$ reachable, we have $bt(te^f) = D$.

The safety conditions [S1]–[S9] are enforced by requiring the subject *2L* program *tp* to be well-annotated, according to the following definition.

Definition 6.6.1 (Well-annotatedness)

1. An expression *te* of the subject *2L* program *tp* is *consistently annotated (c.a.)* with respect to the set of $Bt(x)$ and $bt(e)$ if one of the following holds.
   - $te \equiv x$ and $bt(te) = Bt(x)$.
   - $te \equiv f^{\mathtt{S}}$ and $bt(te) = S$.
   - $te \equiv (\mathtt{if}\ te_1\ te_2\ te_3)^{\mathtt{S}}$, each of te_1, te_2, te_3 is c.a., $bt(te_1) = S$, and $bt(te) = bt(te_2) = bt(te_3)$.
   - $te \equiv (\mathtt{if}\ te_1\ te_2\ te_3)^{\mathtt{D}}$, each of te_1, te_2, te_3 is c.a., and $bt(te) = bt(te_1) = bt(te_2) = bt(te_3) = D$.
   - $te \equiv (op\ te_1\ \ldots\ te_n)^{\mathtt{S}}$, each te_i is c.a., and $bt(te) = bt(te_1) = \ldots = bt(te_n) = S$.
   - $te \equiv (op\ te_1\ \ldots\ te_n)^{\mathtt{D}}$, each te_i is c.a., and $bt(te) = bt(te_1) = \ldots = bt(te_n) = D$.
   - $te \equiv (te_1\ \ te_2)^{\mathtt{S}}$, each of te_1 and te_2 is c.a., and $bt(te_1) = S$.
   - $te \equiv (te_1\ \ te_2)^{\mathtt{D}}$, each of te_1 and te_2 is c.a., and $bt(te) = bt(te_1) = bt(te_2) = D$.
   - $te \equiv (te_1\ \ te_2)^{\mathtt{M}}$, each of te_1 and te_2 is c.a., $bt(te_1) = S$, and $bt(te) = D$.
   - $te \equiv f^{\mathtt{M}}$, and $bt(te) = D$.
   - $te \equiv (\mathtt{lift}\ te_1)$, te_1 is c.a., $bt(te_1) = S$, and $bt(te) = D$.
2. Expression *te* is *well-annotated* if it is consistently annotated and the following are guaranteed.
   - $\tau; (\mathtt{if}\ te_1\ te_2\ te_3)^{\mathtt{S}}$ is reachable and $Sp; \tau; te_1 \downarrow \zeta_1$ imply that $\zeta_1 \notin \mathit{2Clo}$.
   - $\tau; (op\ te_1\ \ldots\ te_n)^{\mathtt{S}}$ is reachable and $Sp; \tau; te_i \downarrow \zeta_i$ imply that $\zeta_i \notin \mathit{2Clo}$.
   - $\tau; (te_1\ \ te_2)^{\mathtt{S}}$ is reachable and $Sp; \tau; te_1 \downarrow \langle f, \xi_1, \ldots \xi_n \rangle$ imply the following.
     - $bt(te_2) = Bt(f^{(n+1)})$.
     - Let *te* be $(te_1\ \ te_2)^{\mathtt{S}}$. Then either $n < ar(f) - 1$ and $bt(te) = S$, or $n = ar(f) - 1$ and $bt(te) = bt(te^f)$.
   - $\tau; (te_1\ \ te_2)^{\mathtt{M}}$ is reachable and $Sp; \tau; te_1 \downarrow \langle f, \xi_1, \ldots, \xi_n \rangle$ imply the following.
     - $bt(te_2) = Bt(f^{(n+1)})$.
     - $Bt(f^{(n+1)}) = \ldots = Bt(f^{(ar(f))}) = D$.

  - $bt(te^f) = D$.
- $\tau; f^{\mathtt{M}}$ is reachable implies the following.
  - $Bt(f^{(1)}) = \ldots = Bt(f^{(ar(f))}) = D$.
  - $bt(te^f) = D$.
- $\tau; (\mathtt{lift}\ te_1)$ is reachable and $Sp; \tau; te_1 \downarrow \zeta_1$ imply that $\zeta_1 \notin \mathit{2Clo}$.

3. Program tp is *well-annotated* if each expression te in tp is well-annotated.

Safety of specialization. The development below shows that, barring termination problems, specialization successfully produces a residual program each time.

Definition 6.6.2 (Respecting BT information)

1. $\zeta \in \mathit{2Val}$ *respects the annotation bt and the BT division* if one of the following holds.
   - $bt = S$ and $\zeta \in \mathit{Baseval}$.
   - $bt = S$, $\zeta = \langle f, \xi_1, \ldots, \xi_n \rangle$, $0 \le n < ar(f)$, and for $0 < i \le n$, ξ_i respects $Bt(f^{(i)})$.
   - $bt = D$ and $\zeta \in \mathit{Expr}$.

   For brevity, we sometimes write "ζ respects bt" for "ζ respects bt and the BT division."

2. Evaluation environment τ *respects the BT division* if for $x \in \mathit{Var}$, $\tau(x)$ respects $Bt(x)$.

Lemma 6.6.3 Suppose that the subject *2L* program tp is well-annotated. For τ respecting the BT division, if $\tau; te$ is reachable and $Sp; \tau; te \downarrow \zeta$, then ζ respects $bt(te)$ and the BT division.

Proof Let $\tau; te$ be reachable and $Sp; \tau; te \downarrow \zeta$. The proof that ζ respects $bt(te)$ is by induction on the size of the evaluation tree of $Sp; \tau; te$, with case analysis of the applicable rule at the root of the tree. We may assume, by the induction hypothesis, that the result of any subcomputation $\tau_i; te_i$ respects $bt(te_i)$, provided the updated environment τ_i continues to respect the BT division.

- If the applicable proof rule is (1), then $te \equiv x$. By assumption, τ respects the BT division, so $\zeta = \tau(x)$ respects $Bt(x)$, which is equal to $bt(te)$ by well-annotatedness ([S1]).

- If the rule is (2), $te \equiv f^{\mathtt{S}}$ and $\zeta = \langle f \rangle$. By definition, ζ respects $bt(te) = S$.

- If the rule is (3), (10) or (11), $\zeta = \mathit{error}_{bt(te)}$, which respects $bt(te)$.

- If the rule is (4) or (5), then $te = (\mathtt{if}\ te_1\ te_2\ te_3)^{bt}$, where $bt(te) = bt(te_2) = bt(te_3)$ by well-annotatedness ([S2]). Since ζ is either the evaluation of te_2 or te_3, by the induction hypothesis, it respects $bt(te_2)$ or $bt(te_3)$, and both are equal to $bt(te)$.

- If the rule is one of (6), (9), (14)–(19) then $bt(te) = D$, and by definition, the result of any *makeIf*, *makeOp*, *makeApp*, *lift* and *makeMem* is an expression, which respects D.

- If the rule is (7) or (8), then $\zeta = \mathtt{Err}$ or $\zeta = \mathcal{O}[\![op]\!](\zeta_1, \ldots, \zeta_n)$. In either case, $\zeta \in \mathit{Baseval}$, so ζ respects $bt(te) = S$.

- If the rule is (12) (for partial application), then $te \equiv (te_1\ te_2)^{\mathtt{S}}$. By the induction hypothesis, $\tau; te_1 \downarrow \langle f, \xi_1, \ldots, \xi_n\rangle$, which respects S and the BT division, so $n < ar(f)$, and for $i = 1, \ldots, n$, ξ_i respects $Bt(f^{(i)})$. By a condition for the proof rule, $n \neq ar(f) - 1$, so $n + 1 < ar(f)$. Again by the induction hypothesis, $\tau; te_2 \downarrow \xi_{n+1}$, which respects $bt(te_2)$. By well-annotatedness ([S7]), $bt(te_2) = Bt(f^{(n+1)})$, so ξ_{n+1} respects $Bt(f^{(n+1)})$. It follows that $\xi = \langle f, \xi_1, \ldots, \xi_{n+1}\rangle$ respects $bt(te) = S$ and the BT division.

- If the applicable rule is (13) (for the unfolding transition), then $te \equiv (te_1\ te_2)^{\mathtt{S}}$. By the inductive hypothesis, $\tau; te_1 \downarrow \langle f, \xi_1, \ldots, \xi_{ar(f)-1}\rangle$, which respects S and the BT division, and $\tau; te_2 \downarrow \xi_{ar(f)}$, which respects $bt(te)$. It follows from these facts and from well-annotatedness ([S7]) that for $i = 1, \ldots, ar(f)$, ξ_i respects $Bt(f^{(i)})$. Let $\tau_1 = \tau[f^{(i)} \mapsto \xi_i]_{i=1,ar(f)}$. Then τ_1 respects the BT division. Now, $\tau_1; te^f \downarrow \zeta$, so by the induction hypothesis, ζ respects $bt(te^f)$. By well-annotatedness ([S7]), $bt(te^f) = bt(te)$, so ζ respects $bt(te)$, as desired.

Lemma 6.6.4 (Well-definedness of specialization)

1. Let ζ respect S and the BT division. Then $FVV(\zeta)$ and $SubV(\zeta, \vec{fv}, \vec{nv})$ are well-defined, and $SubV(\zeta, \vec{fv}, \vec{nv})$ respects S and the BT division.
2. Let $\tau; (te_1\ te_2)^{\mathtt{M}}$ be reachable where τ respects the BT division. Suppose that $\tau; te_1 \downarrow \langle f, \xi_1, \ldots, \xi_n\rangle$ and $\tau; te_2 \downarrow \xi_{n+1} \neq \mathtt{Err}$. And let $\tau' = \tau[f^{(i)} \mapsto \xi_i]_{i=1,n+1}$. Then $makeMem(f, \tau', n+1)$ and $nextMem(f, \tau')$ are well-defined. Further, $nextMem(f, \tau') = (f', argv', f, \tau_1)$ where τ_1 respects the BT division.
3. Let $\tau; (te_1\ te_2)^{\mathtt{M}}$ be reachable where τ respects the BT division. Then $makeMem(f, \tau, 0)$ and $nextMem(f, \tau)$ are well-defined. Further, $nextMem(f, \tau) = (f', argv', f, \tau_1)$ where τ_1 respects the BT division.

Proof The first claim is proved by induction on the form of ζ. The other claims follow from the relevant definitions using Lemma 6.6.3 and well-annotatedness (the safety requirements). We omit the details.

Corollary 6.6.5 For $\tau; te$ reachable, τ respects the BT division.

Proof The initial environment τ_0 respects the BT division. Assume that τ respects the BT division, and prove that for $\tau; te$ reachable and $\tau; te \hookrightarrow \tau_1; te_1$, τ_1 respects the BT division. The corollary follows by the principle of mathematical induction.

If $\tau; te \overset{S}{\hookrightarrow} \tau_1; te_1$, then either $\tau_1 = \tau$, and there is nothing to prove; or the next environment has the form $\tau_1 = \tau[f^{(i)} \mapsto \xi_i]_{i=1,ar(f)}$, and the applicable rule is (13) for the unfolding transition. In this case, $te^{bt} \equiv (te_1\ te_2)^{\mathtt{S}}$, $\tau; te_1 \downarrow \langle f, \xi_1, \ldots, \xi_n\rangle$ and $\tau; te_2 \downarrow \xi_{n+1} \neq \mathtt{Err}$ with $n = ar(f) - 1$. By Lemma 6.6.3, $\langle f, \xi_1, \ldots, \xi_n\rangle$ respects S and the BT division, implying that for $0 < i \leq n$, ξ_i respects $Bt(f^{(i)})$. As for ξ_{n+1}, it respects $bt(te_2)$ by Lemma 6.6.3, and $bt(te_2) = Bt(f^{(ar(f))})$ by well-annotatedness ([S7]). Thus τ_1 continues to respect the BT division.

For memoization transitions, $te \equiv f^{\mathtt{M}}$ or $te \equiv (te_1\ te_2)^{\mathtt{M}}$. By Lemma 6.6.4, in each case, $\tau; te \overset{D}{\hookrightarrow} \tau_1; te_1$ such that τ_1 respects the BT division. This completes the proof.

Theorem 6.6.6 Assume that the variable boundedness proposition and finite unfolding proposition hold. Suppose that the subject *2L* program *tp* is well-annotated. Then Sp^* (the minimal set of specialization configurations) is defined and finite, and for each configuration, the corresponding evaluation state $\tau; te$ is such that $Sp^*; \tau; te \downarrow \zeta$ for some $\zeta \in Expr$. Therefore, a set of residual definitions is successfully generated.

Proof (Sketch)

1. Argue that for $\tau; te$ reachable, if there does not exist ζ such that $\tau; te \downarrow \zeta$, there is a subcomputation $\tau; te \overset{S}{\hookrightarrow} \tau_1; te_1$ such that $\tau_1; te_1$ is non-terminating. This implies an infinite $\overset{S}{\hookrightarrow}$ chain starting with a reachable state, which violates the finite unfolding proposition. The proof of this claim is not difficult, albeit somewhat tedious.

   Informally, the cases in the definition of $\overset{S}{\hookrightarrow}$ are exhaustive, and in each case, if all the subcomputations terminate, so does the computation. The actual proof uses Lemma 6.6.5, which implies that τ respects the BT division; Lemma 6.6.3, which implies that any the result of any terminating subcomputation respects the binding-time of the subexpression; Lemma 6.6.4, which implies that any `M`-annotated expression can be evaluated for the appropriate *Sp*; well-annotatedness (the safety requirements); and the totality of the *make* functions.

2. Fix the initial state. In the following, by "reachable," we mean reachable from this state. Sp_0 is certainly well-defined, and the state corresponding to the only configuration in Sp_0 is (trivially) reachable. For Sp_i whose configurations correspond to reachable states, it follows from the last claim that such a state $\tau; te$ is terminating for the appropriate *Sp*. So Sp_{i+1} is well-defined. By definition of $\hookrightarrow$, the states corresponding to configurations of Sp_{i+1} are reachable from $\tau; te$, thus reachable from the initial state. By the principle of mathematical induction, each Sp_i, and hence Sp^*, is well-defined, and consists of reachable states.

3. By the variable boundedness proposition, Sp^* is finite. For each configuration in Sp^*, the corresponding reachable state $\tau; te^f$ is terminating by the first claim, so there exists ζ such that $\tau; te^f \downarrow \zeta$. It follows from well-annotatedness ([S9]) that $bt(te^f) = D$, so we deduce from Lemma 6.6.3 that $\zeta \in \mathit{Expr}$. In other words, a definition is successfully generated for each configuration in Sp^*. The residual program p' is therefore defined.

6.7 Correctness of partial evaluation

So given the variable boundedness proposition, the finite unfolding proposition, and well-annotatedness, a residual program p' is guaranteed to be generated. Is p' correct?

Proposition 6.7.1 Let p' be the specialization of *tp* with τ_0. Suppose that p' has initial function f_0' (i.e., $\tau_0; te^{f_{initial}} \downarrow \zeta_0 = e^{f_0'}$), with parameters corresponding to the dynamic arguments of $f_{initial}$. Let σ_0 be an evaluation environment for $p = \phi(tp)$ such that

$$\begin{array}{ll} \sigma_0(x) = \tau_0(x) \in \mathit{Baseval}, & \text{for } x \in \mathit{Param}(f_{initial}) \text{ and } Bt(x) = S, \\ \sigma_0(x) \in \mathit{Baseval}, & \text{for } x \in \mathit{Param}(f_{initial}) \text{ and } Bt(x) = D, \\ \sigma_0(x) = \mathtt{Err}, & \text{otherwise.} \end{array}$$

And let σ_0' be the evaluation environment for p' such that $\sigma_0'(x) = \sigma_0(x)$ for $x \in \mathit{Param}(f_{initial})$ and $Bt(x) = D$ (these are the parameters of f_0'). The proposition is: $\sigma_0; e^{f_{initial}} \Downarrow_p v \in \mathit{Baseval} \setminus \{\mathtt{Err}\}$ just when $\sigma_0'; e^{f_0'} \Downarrow_{p'} v \in \mathit{Baseval} \setminus \{\mathtt{Err}\}$.

6.7.1 Preserving the results of successful computations

In this section, we work towards a proof for the forward implication of the correctness proposition, which establishes the preservation of the results of *successful* computations. The proof must handle values from the specialization, the evaluation of the subject program p, and the evaluation of residual program p'. To avoid confusion, we standardize the variables for referring to these values: $\zeta, \zeta_i, \zeta_{(i)}, \xi, \xi_i, \ldots$ will refer to values of *2Val*, handled by the specializer; $v, v_i, v_{(i)}, u, u_i, \ldots$ will refer to the values handled by the subject program p during its evaluation; and $v', v_i', v_{(i)}', u', u_i', \ldots$ will refer to the values handled by the residual program p' during *its* evaluation.

The difficulty in *formulating* a statement of correctness that lends itself to induction is in relating the residual program's runtime closures to the subject program's runtime closures. The next definition addresses this.

Definition 6.7.2 (Relating residual-program and subject-program closures)

1. The specialization of a static expression te ($bt(te) = S$) yields a base value or a partially static closure. In the latter case, the static parts of the closure should *match* the corresponding parts of the actual evaluation of $\phi(te)$, in some sense, given *matching* specialization and evaluation environments.

   For $\zeta \in \mathit{2Val}$ and $v \in \mathit{Val}$, ζ *matches* v if one of the following holds.

   - $\zeta \in \mathit{Baseval}$ and $\zeta = v$;
   - $\zeta = \langle f, \xi_1, \ldots, \xi_n \rangle$, $v = \langle f, u_1, \ldots, u_n \rangle$, and for $Bt(f^{(i)}) = S$: ξ_i matches u_i. For $Bt(f^{(i)}) = D$, call (ξ_i, u_i) an *unmatched pair* of ζ and v. Further unmatched pairs of ζ and v include those of ξ_i and u_i for $Bt(f^{(i)}) = S$;
   - $\zeta \in \mathit{Expr}$. In this case, (ζ, v) is an unmatched pair.

   Thus ζ either does not match v, or it matches v, possibly with unmatched pairs (t_k, u_k).

2. Let te be a subexpression of te^f, and $\tau; te \downarrow \zeta$. For (τ, σ') *corresponding to* σ on f, in the sense that the dynamic parts of τ evaluate with σ' in the way expected by σ, *and* the static parts of τ match σ: if $\sigma'; \zeta \Downarrow_{p'} v'$ and $\sigma; \phi(te) \Downarrow_p v$, then v and v' *agree.* (We cannot insist that $v' = v$, since v' may be the closure of some residual function.) Formally, let p' be a specialization of tp, a well-annotated version of p. In this context, v and v' *agree* if

   - $v, v' \in \mathit{Baseval}$ and $v = v'$; or
   - $v = \langle f, v_1, \ldots, v_n \rangle$, $v' = \langle f', v'_1, \ldots, v'_m \rangle$, $f' = \mathit{nextFun}(f, \tau)$, and the following hold.
     - For $i = 1, \ldots, n$: $\overline{\tau}(f^{(i)})$ (i.e., the value of $f^{(i)}$ used to specialize f') matches v_i, and for every unmatched pair (t, u) and σ' such that $\sigma'(f'^{(i)}) = v'_i$, $i = 1, \ldots, m$: $\sigma'; t \Downarrow_{p'} u'$ for some u' agreeing with u.
       (*Point*: In the evaluation environment σ' for the residual program, any dynamic part t evaluates "as expected.")
     - $ar(f) - n = ar(f') - m$, i.e., both closures are at the same level of instantiation.

3. The pair (ζ, σ') *corresponds to* v if ζ matches v and for every unmatched pair (t, u): $\sigma'; t \Downarrow_{p'} u'$ where u' agrees with u.

4. The pair (τ, σ') *corresponds to* σ *on* f if τ respects the BT division, and for $x \in \mathit{Param}(f)$, $(\tau(x), \sigma')$ corresponds to $\sigma(x)$.

Theorem 6.7.3 (Preservation of the results of successful computations) Let p' be a specialization of tp, a well-annotated version of p. Let te be a subexpression of te^g, and $e = \phi(te)$. Suppose that (τ, σ') corresponds to σ on g, and $\sigma; e \Downarrow v \neq \texttt{Err}$. If $\tau; te \downarrow \zeta$, then (ζ, σ') corresponds to v.

Proof The proof is by induction on the size of the proof tree for $\sigma; e$. The details, which are rather involved, are given in the appendix. Refer to the proof of B.1.1

Corollary 6.7.4 The forward direction of Prop. 6.7.1 holds.

Proof In the proposition, p' is a specialization of tp, a well-annotated version of p. Since $bt(te^{f_{initial}}) = D$ by well-annotatedness (which is one of the safety requirements), and (τ_0, σ'_0) corresponds to σ_0 on $f_{initial}$, if $\sigma_0; e^{f_{initial}} \Downarrow v \in \mathit{Baseval} \setminus \{\texttt{Err}\}$ and $\tau_0; te^{f_{initial}} \downarrow \zeta_0$, then $\sigma'_0; \zeta_0 \Downarrow v'$ such that v' agrees with v, i.e., $v' = v$, since $v \in \mathit{Baseval}$. This establishes the preservation of the results of successful computations.
Remark: The condition that $v \neq \texttt{Err}$ in Prop. 6.7.1 is important. As it stands, the specialization procedure could well discard erroneous subcomputations during unfolding, allowing the residual program to succeed where the source program would have aborted.

6.7.2 Preserving non-termination and errors

The reverse implication of the correctness proposition does not hold! The problem is that the specializer, as defined, may inadvertently discard a divergent computation during unfolding, causing the residual program to terminate where the source program, executing on a corresponding environment, would have diverged. This is the same problem as discarding erroneous subcomputations, mentioned above. A related problem resulting from careless unfolding is the duplication of code, leading to the duplication of computation at runtime. This can cause the transformation to degrade, rather than improve performance.

Part of Similix's strategy [Bon91b] for overcoming these problems involves inserting let-bindings of the form `(let ((x x)) ...)`, for every dynamic variable `x`. It is then decided which `lets` should be unfolded, and which should remain in the residual code. This allows decisions about the inlining of dynamic arguments to be separated from other aspects of the specialization. Our subject language L does not possess a let-construct. If let-construction is available, it would still be difficult to adapt Similix's approach. Unlike Scheme, where closure formation via the lambda is guaranteed to succeed, an L application has to evaluate its argument (which may diverge) before forming a closure. This means that *2L* specialization is violating the evaluation order when it suspends the computation of a dynamic argument during the formation of a partially static closure. If the dynamic argument is potentially divergent or erroneous, the closure should be prevented from being formed in the first place (to prevent the erroneous runtime behaviours from being inadvertently discarded). We achieve this by inserting *memoization points.*

More about discarding computations. Generally, it is unsafe to discard dynamic computations. For example, it is not valid to transform e `* 0` as `0`, unless e can be guaranteed to terminate non-erroneously whenever it is evaluated, during execution of the subject program.

A semantic analysis to determine whether an expression is definitely error-free during ordinary evaluation is possible (a naive and conservative analysis comes to mind, which checks for dynamic errors such as taking the head or tail of `[]`, inappropriate closures etc. in an elementary way, although more careful study of the problem is needed for a practical solution. In any case, we point out below the possibility of using termination analysis in conjunction with such an analysis to detect expressions that are both error-free and terminating. These expressions may be safely discarded during unfolding, to give more compact and efficient residual code. The purpose of this development is to illustrate another possible application of termination analysis in offline partial evaluation, other than to ensure finite unfolding (discussed later).

Querying the termination of expressions. The following is a naive approach to determine whether a program expression e always terminates at runtime.

1. Let $F = \{f \mid (e'\ e'')$ is a subexpression of $e, \langle\!\langle f, ar(f){-}1\rangle\!\rangle \in \mathcal{C}l[\![e']\!]\}$. F contains the target functions of potential function-call transitions due to e, according to closure analysis.

2. Let $f \to f'$ if $f \stackrel{G'}{\to} f'$ is an abstract function-call transition of $\mathit{Tr}(p)$. Determine the set of functions reachable from functions of F as $F' = \{f' \mid f \in F, f \to^* f'\}$. The abstract function-call transitions from these functions are: $\mathit{Tr}' = \{f' \stackrel{G''}{\to} f'' \in \mathit{Tr}(p) \mid f' \in F'\}$.

3. Decide whether Tr' is terminating by a termination criterion.

There exist better schemes for managing multiple queries about the termination of program expressions. However, the above approach is intuitive. Informally, any infinite $\leadsto$ chain starting with a reachable state of the form $\sigma; e$ (where e is the expression being considered) has a subsequence corresponding to function-call transitions (see the proof of Theorem 3.5.24). As each function-call transition is modelled by some element of Tr', this implies an infinite Tr'-multipath. Since Tr' satisfies a termination criterion, the Tr'-multipath has infinite descent, which induces a sequence of values whose sizes descend infinitely. As this is impossible, we conclude that any infinite $\leadsto$ chain starting with a reachable state of the form $\sigma; e$ is impossible, i.e., the evaluation of e always terminates in any execution of p.

Occurrence counting. *Occurrence counting* is the approach used by Similix to determine an upper bound for the number of references to a variable during ordinary evaluation.

Definition 6.7.5 (oc) For parameter x and expression e, define $oc(x, e)$ as follows.

$$
\begin{array}{lcl}
oc(x, x) & = & 1 \\
oc(x, y) & = & 0 \text{ if } y \not\equiv x \\
oc(x, f) & = & 0 \\
oc(x, (\texttt{if}\ e_1\ e_2\ e_3)) & = & oc(x, e_1) + max(oc(x, e_2), oc(x, e_3)) \\
oc(x, (op\ e_1\ \ldots\ e_n)) & = & \Sigma_{i=1,n}\, oc(x, e_i) \\
oc(x, (e_1\ e_2)) & = & oc(x, e_1) + oc(x, e_2)
\end{array}
$$

$oc(x, e)$ is the maximum number of times x is referenced when e is evaluated. This information is used to determine which dynamic arguments must not be inlined, to avoid duplication of runtime computation. The only interesting case in the definition of oc is the one for the conditional: During evaluation, only one of the branches can be executed, thus the occurrence count is the sum of the count for the condition, and the maximum of the counts for the branches.

Inserting memoization points.

Definition 6.7.6 (Memoization points)

1. Let $te \equiv (te_1\ te_2)^{\mathtt{S}}$ be in the subject *2L* program tp. It is *dangerous to unfold te more than n arguments* if $\langle\!\langle f, ar(f) - n - 1\rangle\!\rangle \in \mathcal{C}l[\![\phi(te_1)]\!]$ for some f, $bt(te_2) = D$, and $e_2 = \phi(te_2)$ is possibly non-terminating or erroneous in p (as e_2 may disappear after unfolding). It is dangerous to unfold te more than n arguments if it is dangerous to unfold te_1 more than $n + 1$ arguments.
2. Function f is *dangerous to unfold* if for some n such that $Bt(f^{(n)}) = D$, it holds that $oc(f^{(n)}, e^f) > 1$ (as any computation bound to $f^{(n)}$ may be duplicated).
3. We will be specifying various conditions for a tp application $te \equiv (te_1\ te_2)^{\mathtt{S}}$ to be considered a *memoization point*. We begin with the following. The application te is a memoization point if one of the conditions below is satisfied.
   - It is dangerous to unfold te more than n arguments for some n, and the expression containing te as an immediate subexpression is *not* an application.
   - It is dangerous to unfold te more than 0 argument.
   - For some f that is dangerous to unfold, $\langle\!\langle f, ar(f) - 1\rangle\!\rangle \in \mathcal{C}l[\![\phi(te_1)]\!]$.

   Note that a memoization point is an application annotated with `S`.

An application that is dangerous to unfold must not be allowed to cause an unfolding transition, or to form a static closure that is completed eventually. The result of such an application should be kept "local." The memoization points are designed to occur as late as possible, to maximize the amount of specialization. The intention is convert each memoization point $(te_1\ te_2)^{\mathtt{S}}$ in tp to $(te_1\ te_2)^{\mathtt{M}}$, unless te_1 has become dynamic. This is treated later.

The introduction of memoization points means that static computations that could have proceeded may be prevented from doing so by the existence of irrelevant (but possibly non-terminating or erroneous) computations. The effect seems severe if a suspended static computation is recursive. In Similix, a continuation-based reduction strategy is used to allow any static context surrounding a dynamic let-construct to interact with the construct's static body [JGS93], but a discussion of these matters goes too far beyond the scope of the present study.

Our approach for preserving divergent or erroneous computations is based on inspecting the dynamic expressions that may appear as arguments of static closures. Generalization is minimized when such arguments are kept simple in the subject *2L* program. Whenever possible, it should be arranged so that dynamic variables, rather than larger expressions, participate in the formation of static closures.

Preserving source-program semantics, revisited. The proposed strategy is still not completely faithful with respect to the divergent behaviour of the subject program. By delaying memoization points, we allow static errors to cause (potentially divergent) dynamic arguments to be discarded. On the other hand, generalizing *every* static application for which it is dangerous some number of arguments introduces sensitivity on the ordering of function parameters (since it would then be preferable to arrange static parameters before dynamic ones, to mitigate the effect of memoization points). Our strategy only guarantees that if the residual program terminates *non-erroneously*, then so does the subject program. (We already know that if the subject program terminates non-erroneously, then so does the residual program.)

Theorem 6.7.7 Let p' be a specialization of tp, a well-annotated version of p *without memoization points*. Suppose that (τ_0, σ_0') corresponds to σ_0 on $f_{initial}$, and $\tau_0; te^{f_{initial}} \downarrow \zeta_0$. If $\sigma_0; e^{f_{initial}}$ diverges, i.e., there is an infinite $\rightsquigarrow$ chain starting with $\sigma_0; e^{f_{initial}}$, then either $\sigma_0'; \zeta_0$ diverges, or $\sigma_0'; \zeta_0 \Downarrow \mathtt{Err}$.

Proof Refer to the proof of B.1.2 in the appendix.

Theorem 6.7.8 Let p' be a specialization of tp, a well-annotated version of p *without memoization points*. Consider a subexpression te in te^g such that either te does *not* have the form $(te_1\ te_2)^{\mathtt{S}}$, or it is *not* dangerous to unfold te any number of arguments. In particular, te may be $te^{f_{initial}}$.

Suppose that (τ, σ') corresponds to σ on g, and for $e = \phi(te)$: $\sigma; e$ is reachable and $\sigma; e \Downarrow \mathtt{Err}$. If $\tau; te \downarrow \zeta$, then ζ is *erroneous with* σ': If $bt(te) = S$ then $\zeta = \mathtt{Err}$, and if $bt(te) = D$ then $\sigma'; \zeta \Downarrow \mathtt{Err}$.

Proof The proof is by induction on the size of the proof tree for $\sigma; e$. Refer to the proof of B.1.3 in the appendix for details.

The last two theorems establish some form of correctness for our specializer. As for the non-duplication of runtime computation, we observe informally that for computation to be duplicated, a specialization-time call to some function f must be unfolded, where f has a dynamic formal argument $f^{(i)}$ having an occurrence count greater than 1 in e^f. This implies the reduction of an application of the form $te \equiv (te_1\ te_2)^{\mathtt{S}}$, where $\tau; te_1 \downarrow \langle f, \xi_1, \ldots, \xi_{ar(f)-1} \rangle$. Furthermore, the result of this reduction is executed during the evaluation of the residual program. This implies that $\phi(te)$ causes a function-call transition to f in the related source program computation. As such, $\langle\!\langle f, ar(f)-1 \rangle\!\rangle \in \mathcal{C}l[\![\phi(te_1)]\!]$. Now, by definition, f is dangerous to unfold. Therefore te is a memoization point, and could not have been annotated $\mathtt{S}$.

6.8 Finite unfolding

This section contains a proposal for ensuring finite unfolding by further insertion of memoization points. We are given the subject *2L* program tp, a well-annotated version of L program p with respect to BT information given by $bt(te)$ and $Bt(x)$.

Definition 6.8.1 (Static analyses for *2L* programs)

1. Define closure analysis $2\mathcal{C}l[\![te]\!] = \mathcal{C}l[\![\phi(te)]\!]$.
2. Define size analysis $2\mathcal{S}[\![te]\!] = \mathcal{S}[\![\phi(te)]\!]$.
3. Define $2\mathcal{G}$ from $\mathcal{G}$ by restricting the size-relation arcs to those connecting static parameters.
$$2\mathcal{G}[\![te]\!] = \{\langle\!\langle f, n, G' \rangle\!\rangle \mid \langle\!\langle f, n, G \rangle\!\rangle \in \mathcal{G}[\![\phi(te)]\!], G' = \{x \xrightarrow{r} y \in G \mid Bt(x) = Bt(y) = S\}\}$$
4. Define the set of *abstract unfolding transitions* as follows. (The operator $2\mathcal{G}$, like $\mathcal{G}$, approximates the closures assigned to an expression during evaluation (by the specializer). To model unfolding transitions, we use $2\mathcal{G}$ to approximate the result of evaluating a rator in an application, and account for any static argument leading to the unfolding.
$$\begin{aligned} 2Tr = \{g \xrightarrow{G} f \mid (te_1\ te_2)^{\mathtt{S}} \text{ in } te^g,\ \langle\!\langle f, ar(f)-1, G \rangle\!\rangle \in 2\mathcal{G}[\![te_1]\!],\ bt(te_2) = D\} \cup \\ \{g \xrightarrow{G'} f \mid (te_1\ te_2)^{\mathtt{S}} \text{ in } te^g,\ \varrho \in 2\mathcal{G}[\![te_1]\!],\ bt(te_2) = S,\ ext_{2\mathcal{S}[\![te_2]\!]}\varrho = \langle\!\langle f, ar(f), G' \rangle\!\rangle\} \end{aligned}$$

Definition 6.8.2 (Further memoization points)

1. Let $f \to f'$ if there exists some $f \overset{G'}{\to} f'$ in *2Tr*. Let $F' = \{f' \mid f \to^* f'\}$, the set of functions reachable from f. Describe f as *a threat to finite unfolding* if the following set of abstract unfolding transitions is not proved terminating by some criterion: $2Tr_f = \{f' \overset{G''}{\to} f'' \in 2Tr \mid f' \in F'\}$

   (A good method for approximating the set of *all* threats to finite unfolding, other than the brute-force approach, is desirable.)

2. Consider an expression of the form $(te_1\ te_2)^{\mathtt{S}}$ in the subject program *tp* a memoization point if $\langle\!\langle f, ar(f)-1, G\rangle\!\rangle \in 2\mathcal{G}[\![te_1]\!]$ for some G, where f is a threat to finite unfolding.

The intention is for each memoization point $(te_1\ te_2)^{\mathtt{S}}$ in *tp* to be converted to $(te_1\ te_2)^{\mathtt{M}}$, unless te_1 has become dynamic. (This is treated in the next section.) Like Similix's strategy of *specialization-point insertion* [JGS93, BD91], the proposed approach may result in some unnecessary folding. In particular, for a mutual recursion involving some f that is a threat to finite unfolding, neither function call would be unfolded. Occasionally, a residual function may turn out to be referenced in a single function call, which is actually safe to unfold. Any unnecessary function calls may be eliminated during post-processing.

Proposition 6.8.3 (Safety of abstract analyses) In the following, we refer to items embedded in a *2Val* value, the size $\#\zeta$ of *2Val* value ζ, and the free variables $FV(te)$ of *2L* expression te. These notions are extended naturally from the corresponding ones for L values and L expressions.

1. Let $\gamma(\varsigma) = \{\langle f, \xi_1, \ldots, \xi_n\rangle \mid \langle\!\langle f, n\rangle\!\rangle \in \varsigma,\ \xi_i \in 2Val\} \cup Baseval \cup Expr$. Then for $\tau; te$ reachable and $x \in FV(te)$, it holds that $\tau(x) \in \gamma(2\mathcal{Cl}[\![x]\!])$. Further, if ξ' is embedded at $f'(k)$ in $\tau(x)$, it holds that $\xi' \in \gamma(2\mathcal{Cl}[\![f'^{(k)}]\!])$.

   If $\tau; te \downarrow \zeta$ then $\zeta \in \gamma(2\mathcal{Cl}[\![te]\!])$. Further, for any ξ' that is embedded at $f'^{(k)}$ in ζ, it holds that $\xi' \in \gamma(2\mathcal{Cl}[\![f'^{(k)}]\!])$. Thus, $2\mathcal{Cl}$ is correct for $\downarrow$.

2. For $\tau; te$ reachable and $\tau; te \downarrow \zeta \in Baseval$, if $2\mathcal{S}[\![te]\!] = \{\downarrow(x)\}$ where $Bt(x) = S$, then $\#\tau(x) > \#\zeta$. If $2\mathcal{S}[\![e]\!] = \{\mp(x)\}$ where $Bt(x) = S$, then $\#\tau(x) \geq \#\zeta$. Consequently,

   (a) for $\tau; (te_1\ te_2)^{\mathtt{S}}$ reachable such that $\tau; te_1 \downarrow cl = \langle f, \xi_1, \ldots, \xi_n\rangle$ where $n < ar(f) - 1$, and $\tau; te_2 \downarrow \xi_{n+1}$ where $\xi_{n+1} \in Baseval \setminus \{\mathtt{Err}\}$, if ϱ is safe for (cl, τ) then $ext_{2\mathcal{S}[\![te_2]\!]}\ \varrho$ is safe for (cl_1, τ) where cl_1 is the closure $\langle f, \xi_1, \ldots, \xi_{n+1}\rangle$;

   (b) for $\tau; (te_1\ te_2)^{\mathtt{S}}$ reachable such that $\tau; te_1 \downarrow cl = \langle f, \xi_1, \ldots, \xi_{ar(f)-1}\rangle$ and $\tau; te_2 \downarrow \xi_{ar(f)}$ where $\xi_{ar(f)} \in Baseval \setminus \{\mathtt{Err}\}$, if ϱ is safe for (cl, τ) and $ext_{2\mathcal{S}[\![te_2]\!]}\varrho = \langle\!\langle f, ar(f), G\rangle\!\rangle$, then G is safe for the environment pair (τ, τ_1), where $\tau_1 = \tau[f^{(i)} \mapsto \xi_i]_{i=1,ar(f)}$.

3. For $\tau; te$ reachable and $\tau; te \downarrow \langle f, \xi_1, \ldots, \xi_n\rangle$, there exists $\langle\!\langle f, n, G\rangle\!\rangle \in 2\mathcal{G}[\![te]\!]$ such that for the arc $x \overset{\downarrow}{\to} f^{(i)} \in G$ where $i \leq n$, it holds that $\#\tau(x) > \#\xi_i$, and for the arc $x \overset{\mp}{\to} f^{(i)} \in G$ where $i \leq n$, it holds that $\#\tau(x) \geq \#\xi_i$. Other arcs of G have the form $x \overset{\mp}{\to} x$ where $x \notin \{f^{(1)}, \ldots, f^{(n)}\}$.

4. For $\tau_0; te_0$ reachable where te_0 is a subexpression of te^g, and $\tau_0; te_0 \overset{S}{\hookrightarrow} \tau_1; te_1$ is an unfolding transition to f, there exists $g \overset{G}{\to} f \in 2Tr$ such that for $x \overset{\downarrow}{\to} y \in G$, $\#\tau_0(x) > \#\tau_1(y)$ and for $x \overset{\mp}{\to} y \in G$, $\#\tau_0(x) \geq \#\tau_1(y)$.

The development in Ch. 2 and 3 can be adapted for Parts 1 and 3 of the above proposition, despite the fact that there may be certain transitions in $\hookrightarrow$ not corresponding to any transition in $\rightsquigarrow$. (Such transitions arise due to the conservative treatment of dynamic conditionals and applications by the specializer.) This is because $\mathcal{Cl}$ and $\mathcal{G}$ have been designed to be *flow-insensitive.* For Part 2 of the proposition, we need to show first that the relevant parameters of each function for size analysis are static. The formal development is omitted.

Finite unfolding strategy is safe. Suppose, for a contradiction, that there exists an infinite $\overset{S}{\hookrightarrow}$ chain (from a $\hookrightarrow$ reachable state), after ensuring that the subject *2L* program has no static applications that are memoization points. Then there exists some f such that this chain contains infinitely many unfolding transitions to f, since the set of functions is finite. Every unfolding transition in the chain is modelled by an abstract unfolding transition in *2Tr*, by Prop. 6.8.3, Part 4. The concatenation of these abstract unfolding transitions has a tail that is an infinite $2Tr_f$-multipath. If $2Tr_f$ satisfies a termination criterion, this multipath has a thread of infinite descent, which induces an infinite sequence of sizes-decreases. Since this is impossible, we have the required contradiction. We explain next how to ensure that the subject *2L* program has no static applications that are memoization points.

6.9 Deriving safe annotations

Below is a brute-force approach to determine, for a given L program p, a set of annotations and binding-times satisfying well-annotatedness. We also ensure that the variable boundedness proposition holds, using an (as yet unspecified) analysis that detects possibly unbounded variables for a given annotated program.

During the procedure, each formal parameter in p is marked S or D, while each expression is marked S, L, D, M or A, signifying *static*, *lift*, *dynamic*, *memoize* and *(dynamic) application* respectively. Other than S, the rest of the markings correspond to the BT type D (note: *not* BT annotation) The markings indicate variously whether a lift or memoization point should be inserted. They have been ordered according to decreasing amount of specialization. For instance, an expression currently marked S may receive a new marking L to indicate that it should be lifted. This implies less specialization than the original marking. The lift operation becomes unnecessary if the expression receives the marking D for some reason; this entails even less specialization. Or a function call may be marked D. It is then decided that unfolding should be forbidden for the function call, and the marking is changed to M to indicate an `M`-annotation. Subsequently, the `M`-annotation is rendered unnecessary, when the rator becomes dynamic. The marking is changed to A to reflect this. At each stage, the amount of specialization is reduced.

The procedure below is given an L program p and a subset X of $f_{initial}$ parameters that are required to be dynamic. We will use @V(x) and @E(e) to denote the markings for variable x and expression e respectively. Note that the same expression e at different places in p may be marked differently, due to `lifts`. Strictly, @E should take as its argument a *position* in p.

1. Initialize all markings @V(x) and @E(e) to S.
2. For $x \in X$, set @V$(x) := D$.
3. Repeatedly apply the following steps until all the markings are stable. The procedure *gen* is for ensuring that an expression is *not* marked S. It is defined in Fig. 6.5.
   - [S1] (Respecting $Bt(x)$)
     For @V$(x) = D$ and $e \equiv x$, set @E$(e) := D$.
   - [S2] (Consistent annotations)
     - For $e \equiv$ (`if` $e_1\ e_2\ e_3$), if for some i, @E$(e_i) \neq S$, then apply $gen(e_2)$ and $gen(e_3)$, and set @E$(e) := D$.
     - For $e \equiv (op\ e_1\ \ldots\ e_n)$, if for some i, @E$(e_i) \neq S$, then for each i, apply $gen(e_i)$, and set @E$(e) := D$.
     - For $e \equiv (e_1\ e_2)$, if @E$(e_1) \neq S$, then apply $gen(e_2)$, and set @E$(e) := A$.
   - [S3] (Delaying conditionals)
     For $e \equiv$ (`if` $e_1\ e_2\ e_3$), if $\mathcal{C}l[\![e_1]\!] \neq \{\}$, then apply $gen(e_1)$, $gen(e_2)$ and $gen(e_3)$, and set @E$(e) := D$.
   - [S4] (Delaying operations)
     For $e \equiv (op\ e_1\ \ldots\ e_n)$, if for some i, $\mathcal{C}l[\![e_i]\!] \neq \{\}$, then for each i, apply $gen(e_i)$, and set @E$(e) := D$.

- [S5] (Closures in dynamic contexts)
  For $e \equiv (e_1\ e_2)$, if @E$(e) = D$ and $\langle\!\langle f, n \rangle\!\rangle \in \mathcal{C}l[\![e]\!]$ (note that $n < ar(f)$), then set @E$(e) := M$.
- [S6] regarding `lift` is guaranteed by *gen*.
- [S7] (Congruence)
  For $e \equiv (e_1\ e_2)$ and $\langle\!\langle f, n \rangle\!\rangle \in \mathcal{C}l[\![e_1]\!]$, do the following.
  - If @V$(f^{(n+1)}) = D$, then apply $gen(e_2)$.
  - If @E$(e_2) \neq S$, then set @V$(f^{(n+1)}) := D$.
  - If $n = ar(f) - 1$ and @E$(e) \neq S$, then apply $gen(e^f)$.
  - If $n = ar(f) - 1$, @E$(e^f) \neq S$ and @E$(e) \in \{S, L\}$, then set @E$(e) := D$.
- [S8] (Closure arguments)
  For e such that @E$(e) \neq S$ and $\langle\!\langle f, n \rangle\!\rangle \in \mathcal{C}l[\![e]\!]$ (note that $n < ar(f)$), set @V$(f^{(i)}) := D$ for $n < i \leq ar(f)$.
- [S9] (Dynamic definitions)
  First, apply $gen(e^{f_{initial}})$. Next, for e such that @E$(e) \neq S$ and $\langle\!\langle f, n \rangle\!\rangle \in \mathcal{C}l[\![e]\!]$ (note that $n < ar(f)$), apply $gen(e^f)$.

4. (Re-)annotation: For each source function f, the annotated version of e^f is given by $ann(e^f)$, where procedure ann is defined as follows. Given expression e in p:

- If $e \equiv x$ and @E$(e) = L$ then $ann(e) = (\texttt{lift}\ x)$.
- If $e \equiv x$ and @E$(e) \in \{S, D\}$ then $ann(e) = x$.
- If $e \equiv f$ and @E$(e) = S$ then $ann(e) = f^{\texttt{S}}$.
- If $e \equiv f$ and @E$(e) = M$ then $ann(e) = f^{\texttt{M}}$.
- If $e \equiv (\texttt{if}\ e_1\ e_2\ e_3)$ and @E$(e) = L$ then $ann(e) = (\texttt{lift}\ (\texttt{if}\ te_1\ te_2\ te_3)^{\texttt{S}})$, where $te_i = ann(e_i)$ for $i = 1, 2, 3$.
- If $e \equiv (\texttt{if}\ e_1\ e_2\ e_3)$, @E$(e) \in \{S, D\}$ and @E$(e_1) = S$ then $ann(e) = (\texttt{if}\ te_1\ te_2\ te_3)^{\texttt{S}}$, where $te_i = ann(e_i)$ for $i = 1, 2, 3$.
- If $e \equiv (\texttt{if}\ e_1\ e_2\ e_3)$, @E$(e) \in \{S, D\}$ and @E$(e_1) \neq S$ then $ann(e) = (\texttt{if}\ te_1\ te_2\ te_3)^{\texttt{D}}$, where $te_i = ann(e_i)$ for $i = 1, 2, 3$.
- If $e \equiv (op\ e_1\ \ldots\ e_n)$ and @E$(e) = L$ then $ann(e) = (\texttt{lift}\ (op\ te_1\ \ldots\ te_n)^{\texttt{S}})$, where for $i = 1, \ldots, n$, $te_i = ann(e_i)$.
- If $e \equiv (op\ e_1\ \ldots\ e_n)$ and @E$(e) = S$ then $ann(e) = (op\ te_1\ \ldots\ te_n)^{\texttt{S}}$, where $te_i = ann(e_i)$ for $i = 1, \ldots, n$.
- If $e \equiv (op\ e_1\ \ldots\ e_n)$ and @E$(e) = D$ then $ann(e) = (op\ te_1\ \ldots\ te_n)^{\texttt{D}}$, where $te_i = ann(e_i)$ for $i = 1, \ldots, n$.
- If $e \equiv (e_1\ e_2)$ and @E$(e) = L$ then $ann(e) = (\texttt{lift}\ (te_1\ te_2)^{\texttt{S}})$, where $te_i = ann(e_i)$ for $i = 1, 2$.
- If $e \equiv (e_1\ e_2)$ and @E$(e) \in \{S, D\}$ then $ann(e) = (te_1\ te_2)^{\texttt{S}}$, where $te_i = ann(e_i)$ for $i = 1, 2$.
- If $e \equiv (e_1\ e_2)$ and @E$(e) = M$ then $ann(e) = (te_1\ te_2)^{\texttt{M}}$, where $te_i = ann(e_i)$ for $i = 1, 2$.
- If $e \equiv (e_1\ e_2)$ and @E$(e) = A$ then $ann(e) = (te_1\ te_2)^{\texttt{D}}$, where $te_i = ann(e_i)$ for $i = 1, 2$.

The BT division for the annotated program is given by $Bt(x) =$ @V(x). For expression e in p and $ann(e) = te$, we derive the following.

- If @E$(e) = S$ then $bt(te) = S$.
- If @E$(e) \in \{D, M, A\}$ then $bt(te) = D$.
- If @E$(e) = L$ then $te \equiv (\texttt{lift}\ te_1)$. In this case, $bt(te) = D$ and $bt(te_1) = S$.

```
proc gen (e : Expr)
  if @E(e) = S then
    if Cl⟦e⟧ = {} then
      @E(e) := L
    else gencases(e)
    end if
end proc

proc gencases (e : Expr)
  case e of
    x:  @V(x) := D;   @E(e) := D
    f:  @E(e) := M
    (if e1 e2 e3):  gen(e2);  gen(e3);  @E(e) := D
    (op e1 ... en):  @E(e) := L
    (e1 e2):  @E(e) := M
  end case
end proc
```

Figure 6.5: Procedure $gen(e)$ for ensuring that e is *not* marked static

5. Enforcing memoization points: For memoization point e determined from the current annotations and binding-times, set $@\mathrm{E}(e) := M$.
6. Enforcing variable boundedness: Based on some strategy, determine from the current annotations and binding-times a subset of the static parameters that are definitely bounded during specialization. Generalize any static parameter x not in this set by setting $@\mathrm{V}(x) := D$.
7. If generalization is performed in the previous step or there exists a memoization point in the penultimate step, repeat the procedure from Step 3, else terminate.

The annotation process terminates because the markings are non-decreasing in the total ordering $S < L < D < M < A$. We claim, without proof, that upon termination, the annotated program tp is a well-annotated version of the given L program p. Moreover, $Bt(x) = D$ for each $x \in X$, tp has no memoization points, and every x for which $Bt(x) = S$ assumes finitely many values during any specialization. It follows that any result of specialization is correct (in the sense of Prop. 6.7.1), and also the finite unfolding proposition and variable boundedness proposition hold. By Theorem 6.6.6, specialization always terminates.

6.10 The variable boundedness problem, defined

Thus, all is well if the boundedness of evaluation states is guaranteed for each initial environment τ_0. We state the *variable boundedness problem* as follows.

> Given a well-annotated program tp, determine a subset X of parameters guaranteed to assume only boundedly many values during any specialization of tp. In other words, for any initial environment τ_0, the following set should be finite.

$$\{\tau(x) \mid \tau_0; te^{f_{initial}} \hookrightarrow^* \tau; te \text{ and } x \in X\}$$

This gives a strategy for (controlled) generalization, missing from the annotation process described before: Compute X and generalize all the static variables not in X. Eventually, X contains all the static parameters of tp. This signals that the variable boundedness proposition is satisfied, by the following lemma.

Lemma 6.10.1 Assume that the finite unfolding proposition holds. And let τ_0 be an initial environment. Suppose that the following set is finite.

$$\{\tau(x) \mid \tau_0; te^{f_{initial}} \hookrightarrow^* \tau; te \text{ and } Bt(x) = S\}$$

Then the set of states reachable from $\tau_0; te^{f_{initial}}$ is finite, i.e., the variable boundedness proposition is satisfied.

Proof Let τ_0 be an initial environment. Suppose that the following set is finite.

$$V = \{\tau(x) \mid \tau_0; te^{f_{initial}} \hookrightarrow^* \tau; te \text{ and } Bt(x) = S\}$$

Let $St^\star$ contain $\tau_0; te^{f_{initial}}$, and all $\tau_1; te_1$ such that $\tau; te \overset{D}{\hookrightarrow} \tau_1; te_1$ for some $\tau; te$ reachable from $\tau_0; te^{f_{initial}}$. Then any state reachable from $\tau_0; te^{f_{initial}}$ is reachable by a $\overset{S}{\hookrightarrow}$ chain (a sequence of 0 or more $\overset{S}{\hookrightarrow}$ transitions) originating from some state in $St^\star$.

The set of $\tau_1; te_1$ in $St^\star$ is finite because for each x such that $Bt(x) = S$, $\tau_1(x) \in V$, and for each x such that $Bt(x) = D$, $\tau_1(x) = x$ by design. Consider the $\overset{S}{\hookrightarrow}$ relation restricted to states reachable from $\tau_1; te_1$ for a particular $\tau_1; te_1 \in St^\star$. This relation is finite-branching, and has only finite chains, by the finite unfolding proposition. By König's Lemma, the relation is finite. Thus, the set of states reachable by a $\overset{S}{\hookrightarrow}$ chain originating from any particular state in $St^\star$ is finite. It follows that the set of states reachable from $\tau_0; te^{f_{initial}}$ is finite, as desired.

Different criteria for variable boundedness. We refer to a condition for deciding whether parameter x is bounded during any specialization (based on known bounded static parameters) as a *variable boundedness criterion*. Such a criterion is used to accumulate a set of bounded parameters X for solving the variable boundedness problem. In this work, we propose and consider different variable boundedness criteria.

The overall approach. We have proposed a number of modifications to the standard offline partial evaluation strategy while working towards a definition of the variable boundedness problem.

1. Unfolding is suspended, even on configurations for which all of the called function's parameters are bound to static values, if allowing it might lead to infinite unfolding. This sacrifices specialization for better termination behaviour.

2. On the other hand, unfolding that discards unused code is permitted if the discarded code is provably terminating and non-erroneous. Hopefully, this will lead to more compact and efficient residual code.

3. By implementing variable boundedness analysis, termination of specialization is ensured at the expense some reduction. The methods we will study behave well for programs amenable to "descent reasoning" (where unbounded sequences of data construction would give rise to unbounded sequences of size decreases for some parameter values).

Chapter 7

Abstract Interpretation of Argument Dependencies

7.1 An ordering for the specializer values *2Val*

The boundedness of the result of specializing an expression depends on the boundedness of some of the variables in the expression. Such information is crucial for variable boundedness analysis [Hol91, Gle99]. For example, the boundedness of `(if z x y)` is dependent on the boundedness of `x` and `y`, but not `z`. Another example: the boundedness of `(cons x y)` is dependent on the boundedness of `x` and `y`.

In fact, `(cons x y)` is usually considered as having *constructive dependence* on `x` and `y`. The notion of a constructive dependence is not an assertion about the size of the return value per se; what is expressed is this: If the result of evaluating the expression `(cons x y)` is assigned to `x` or `y` in a loop, then unboundedly many different values can be generated. The concept of *construction* is motivated by the following observation: For variable boundedness to be violated, it must be possible to observe unbounded sequences of such constructions during some specialization (i.e., for any N, it is possible to observe a sequence of more than N constructions, possibly at different times during the specialization). A variable boundedness criterion seeks to show that for every specialization, some *uniform* bound exists on the length of any observed construction sequence.

For termination analysis, we discussed size functions for *Val*, the set of data values. For characterizing constructive dependence, the size function must be sensitive to the amount of storage for representing data values, in the following sense: There exists a function $s : I\!N \to I\!N$ such that if all the members in a set of data values have sizes bounded by N, then each of them can be represented using $s(N)$ units of storage. Note that this does not hold for all size functions. For example, if the size of a list is its length, then the data values: `[[]]`, `[[[]]]`, `[[[[]]]]`, ... all have size 1, but no (uniformly) fixed amount of storage is sufficient to represent every value. Such a size function is clearly unsuitable for characterizing constructive dependence. What is required is a size function that has the *downward-closure property*: For every N, the set of data values with sizes no greater than N is finite.

A size-function induces a pre-order on the set of data values. For our dependency analysis, it will be sufficient to have a *downward-closed* pre-order on the set of data values *2Val*, i.e., a pre-order $\leq_{2Val}$ such that for every $\zeta \in 2Val$, the set of ζ' such that $\zeta' \leq_{2Val} \zeta$ is finite[1]. This is more convenient than insisting on a downward-closed size function for *2Val*. The ordering $\leq_{2Val}$ will be used in the following way: The result ζ of an expression will be considered constructive if for every value ζ' of a *dependent* variable, it does *not* hold that $\zeta \leq_{2Val} \zeta'$. For this work, we will fix $\leq_{2Val}$ as follows.

Definition 7.1.1 ($\leq_{2Val}$) For $\zeta, \zeta' \in 2Val$, let $\zeta' \leq_{2Val} \zeta$ if one of the following holds:

- $\zeta' = \zeta$, *OR*
- $\zeta = [\zeta_1, \ldots, \zeta_n]$ where $n > 0$, and either $\zeta' \leq_{2Val} \zeta_1$ or $\zeta' \leq_{2Val} [\zeta_2, \ldots, \zeta_n]$, *OR*
- $\zeta = \langle f, \xi_1, \ldots, \xi_n \rangle$, and $\zeta' \leq_{2Val} \xi_i$ for some i.

If $\zeta' \leq_{2Val} \zeta$, then ζ' is said to *yield to* ζ.

7.2 Dependency analysis

Given an annotated expression *te*, a description of its *variable dependence* is a pair $(D^{\not\uparrow}, D^{\uparrow})$ of parameter sets, signifying the non-constructive and constructive dependencies of *te* respectively. A description is *safe* if whenever we fix the values of the constructive variables, any results from the specialization of *te* that do not yield to the value of each non-constructive variable, form a finite set. A result not yielding to the value of each non-constructive variable indicates that the computation was not dependent on those variables. In that case, the computation of *te* should be uniquely (or up to finite variation) determined by the constructive variables. We formalize these concepts below.

[1] In the author's opinion, downward-closed pre-orders are more natural than well-founded partial orders in the present context. Consider a partial ordering on the data values that is well-founded, but suppose there are infinitely many values less than ζ in the ordering. If there is primitive operator `dec`, which when given ζ, non-deterministically evaluates to some value yielding to ζ, and the result of `dec` is not considered to have increased ζ, then the variable boundedness principle cannot be applied to ensure variable boundedness. Of course, an operator such as `dec` is not realistic.

Definition 7.2.1 (Dependence description)

1. A *dependence description* for *te* is a pair $(D^{\gamma}, D^{\uparrow}) \subseteq FV(te) \times FV(te)$, where $D^{\gamma} \cap D^{\uparrow} = \{\}$.
2. Let $dd = (D^{\gamma}, D^{\uparrow})$ be a dependence description for *te*. A result of specializing *te* with the $D^{\uparrow}$ variables fixed according to τ is *fresh with respect to* τ if it does not yield to the value of any D^{γ} variable in the specialization environment. The set of values that are fresh with respect τ is denoted $1comp(\tau, te, dd)$, and defined as follows.

$$\{\zeta \mid \tau_1; te \downarrow \zeta \text{ where } \forall\, x \in D^{\uparrow} : \tau_1(x) = \tau(x) \text{ and } \forall\, x \in D^{\gamma} : \neg\, \zeta \leq_{2Val} \tau_1(x)\}$$

3. A dependence description $dd = (D^{\gamma}, D^{\uparrow})$ for *te* is *safe for te* if for all τ, the set $1comp(\tau, te, dd)$ is finite.

The above notion of dependency is very general.

1. For *te* $\equiv$`(if ... x (cons ... y)`^{S}`)`^{S}, it is safe to regard *te* as having a non-constructive dependence on `x` and a constructive dependence on `y`. The result of evaluating *te* is constructive *when* it is dependent on `y`.
2. For *te* $\equiv$`(if (member x ...)`^{S} `x y)`^{S} where `member` is the standard list-membership predicate, it is safe to regard *te* as having a non-constructive dependence on `y`, and *no* dependence on `x`. Note that the values of `x` triggering the first case of the conditional form a finite set.
3. For *te* $\equiv$`(if (member x ...)`^{S} `x nil`^{S}`)`^{S}, it is safe to regard *te* as having no dependence on any variable. This example is a variation of the previous one.
4. For *te* $\equiv$`(if (lt x y)`^{S} `(succ x)`^{S} `0`^{S}`)`^{S} where `lt` is the less-than predicate for natural numbers, and `succ` and `0` are the successor and zero operators respectively, it is safe to regard *te* as having a non-constructive dependence on `y` and no dependence on `x`. It is also safe to regard *te* as having a constructive dependence on `x` and no dependence on `y`.
5. For *te* $\equiv$`(nullout zs)`^{S} where `nullout` is given by:

```
nullout xs =
  (if (= xs nilˢ)ˢ
      nilˢ
      (cons nilˢ (nullout (tl xs)ˢ)ˢ)ˢ)ˢ
```

   it is *unsafe* to regard *te* as not having any dependence on `zs`. Glenstrup calls the dependence on `zs`, induced by recursive deconstruction of the value of `zs`, an *immaterial dependence* [Gle99].

General-destructive versus constructive.

Definition 7.2.2 (Program constant) $\zeta \in 2Val$ is a *program constant* if for some specialization environment τ, and expression *te* in the subject *2L* such that $FV(te) = \{\}$: $\tau; te \downarrow \zeta$.

In line with our policy of not placing emphasis on the supporting analyses, we will conservatively distinguish two kinds of expressions: An expression is a *general destructor* if whenever it evaluates non-erroneously, the result definitely part of an input, or part of a program constant; otherwise the expression is considered *constructive*. The analysis $\mathcal{D}es$ of Fig. 7.1 determines whether an expression is definitely general-destructive. It is monotonic in the boolean domain *True* < *False*, and therefore always terminates. The auxiliary function $\mathcal{A}rg$ checks whether the arguments of an application are all general-destructive, and for any unfolding transition encountered, whether the body of the target function is also general-destructive.

$$
\begin{array}{lcl}
\mathcal{D}es[\![te]\!] & = & \mathit{True},\ \text{if } FV(te) = \{\}\ (te \text{ is } \mathit{constant}),\ \text{else} \\
\mathcal{D}es[\![x]\!] & = & \mathit{True} \\
\mathcal{D}es[\![(\texttt{hd}\ te_1)^{\texttt{S}}]\!] & = & \mathcal{D}es[\![te_1]\!] \\
\mathcal{D}es[\![(\texttt{tl}\ te_1)^{\texttt{S}}]\!] & = & \mathcal{D}es[\![te_1]\!] \\
\mathcal{D}es[\![(\texttt{if}\ te_1\ te_2\ te_3)^{\texttt{S}}]\!] & = & \mathcal{D}es[\![te_2]\!] \wedge \mathcal{D}es[\![te_3]\!] \\
\mathcal{D}es[\![(te_1\ te_2)^{\texttt{S}}]\!] & = & (\forall \langle\!\langle f, n\rangle\!\rangle \in \mathit{2Cl}[\![te_1]\!] : n = ar(f) - 1) \text{ and } \mathcal{A}rg[\![(te_1\ te_2)^{\texttt{S}}]\!] \\
\mathcal{D}es[\![te]\!] & = & \mathit{False},\ \text{otherwise.} \\
 & & \\
\mathcal{A}rg[\![(te_1\ te_2)^{\texttt{S}}]\!] & = & \bigwedge\{\mathcal{D}es[\![te^f]\!] \mid \langle\!\langle f, ar(f)-1\rangle\!\rangle \in \mathit{2Cl}[\![te_1]\!]\} \wedge \mathcal{D}es[\![te_2]\!] \wedge \mathcal{A}rg[\![te_1]\!] \\
\mathcal{A}rg[\![(\texttt{if}\ te_1\ te_2\ te_3)^{\texttt{S}}]\!] & = & \mathcal{A}rg[\![te_2]\!] \wedge \mathcal{A}rg[\![te_3]\!] \\
\mathcal{A}rg[\![te]\!] & = & \mathit{True},\ \text{otherwise.}
\end{array}
$$

Figure 7.1: Analysis $\mathcal{D}es$

Lemma 7.2.3 Let te be an expression in the subject *2L* program tp. Suppose that $\tau; te \downarrow \zeta \neq \texttt{Err}$.

1. If $\mathcal{D}es[\![te]\!]$ is true, then either ζ yields to some program constant, or for some $x \in FV(te)$, it holds that $\zeta \leq_{\mathit{2Val}} \tau(x)$.
2. If $\mathcal{A}rg[\![te]\!]$ is true and $\zeta = \langle f, \xi_1, \ldots, \xi_n\rangle$, then for $1 \leq i \leq n$, either ξ_i yields to some program constant, or there exists $x \in FV(te)$ such that $\xi_i \leq_{\mathit{2Val}} \tau(x)$.

Proof is by induction on the size of the proof tree for $\tau; te \downarrow \zeta \neq \texttt{Err}$. We omit the details.

A conservative dependence description.

Definition 7.2.4 Define the function *Dep*, mapping an annotated expression te to a dependency description, as follows.

$$
Dep(te) = \begin{cases} (FV(te), \{\}), & \text{if } \mathcal{D}es[\![te]\!] = \mathit{True}, \\ (\{\}, FV(te)), & \text{otherwise.} \end{cases}
$$

Lemma 7.2.5 *Dep*(te) gives a safe dependency description for te.

Proof If $\mathcal{D}es[\![te]\!]$ is false, then $Dep(te) = (\{\}, FV(te))$. For any τ, if for all $x \in FV(te)$, $\tau_1(x) = \tau(x)$, it is easily proved that the result of specializing $\tau_1; te$ and $\tau; te$ are equal, when they exist. Thus, the set $\mathit{1comp}(\tau, te, Dep(te)) = \{\zeta \mid \tau_1; te \downarrow \zeta,\ \forall x \in FV(te) : \tau(x)_1 = \tau(x), \ldots\}$ is a singleton or empty, hence finite.

If $\mathcal{D}es[\![te]\!]$ is true, then $Dep(te) = (FV(te), \{\})$. By Lemma 7.2.3, the set $\mathit{1comp}(\tau, te, Dep(te))$, which is $\{\zeta \mid \tau_1; te \downarrow \zeta, \ldots, \forall x \in FV(te) : \neg\,\zeta \leq_{\mathit{2Val}} \tau_1(x)\}$ contains ζ only if ζ yields to some program constant. It is therefore finite.

Operator *Dep* gives safe dependency descriptions for the argument expressions of function applications, but ones that are much more conservative than allowed by the notion of safety. However, our aim in this chapter is to present a simple, rather than strong dependency analysis. An intuitive justification of the appropriateness of *Dep* is this: Argument expressions that are designed to avoid data construction during mixed computation tend to be simple anyway, often general-destructive.

7.3 Collection of abstract transitions

Given annotated expression te, the operator $\mathcal{T}r$, defined in Fig. 7.2, collects abstractions of $\hookrightarrow$ transitions due to the specialization of te. These are analogous to the abstract function-call transitions for ordinary L evaluation.

The following definitions use *ext*, defined for SRG analysis, and $\mathcal{2G}$ defined in the last chapter. The next operator extends a closure abstraction by an argument.

$$
\begin{aligned}
bind(\langle\!\langle f, n, \Gamma, \Delta\rangle\!\rangle, te) \;&=\; \langle\!\langle f, n+1, \Gamma', \Delta'\rangle\!\rangle,\\
&\quad \text{where } \langle\!\langle f, n+1, \Gamma'\rangle\!\rangle = \begin{cases} ext_{\mathcal{2S}[\![te]\!]}\ \langle\!\langle f, n, \Gamma\rangle\!\rangle, & \text{if } bt(te) = S,\\ ext_{\{\}}\ \langle\!\langle f, n, \Gamma\rangle\!\rangle, & \text{if } bt(te) = D,\end{cases}\\
&\quad \text{and } \Delta' = \Delta[f^{(n+1)} \mapsto Dep(te)].
\end{aligned}
$$

$\mathcal{P}App$ collects abstract closures for te, maximizing argument-dependency information by recursing up the arguments until the rator possibly gives rise to an unfolding transition, or is not an application.

$$
\begin{aligned}
\mathcal{P}App[\![(\texttt{if}\ te_1\ te_2\ te_3)^{\texttt{S}}]\!] &= \mathcal{P}App[\![te_2]\!] \cup \mathcal{P}App[\![te_3]\!]\\
\mathcal{P}App[\![(te_1\ te_2)^{\texttt{S}}]\!] &= \{bind(\varrho, te_2) \mid \varrho \in \mathcal{P}App[\![te_1]\!]\}\\
&\qquad\qquad \text{if for } \langle\!\langle f, n\rangle\!\rangle \in \mathcal{2C}l[\![te_1]\!],\ n \neq ar(f) - 1,\\
\mathcal{P}App[\![te]\!] &= \{\langle\!\langle f, n, \Gamma, \Delta\rangle\!\rangle \mid \langle\!\langle f, n, \Gamma\rangle\!\rangle \in \mathcal{2G}[\![te]\!],\\
&\qquad \Delta(f^{(i)}) = Dep(te) \text{ for } 1 \leq i \leq n\},\ \text{otherwise.}
\end{aligned}
$$

$\mathcal{T}r$ collects abstract $\hookrightarrow$ transitions. Assume that its argument is a subexpression of te^g.

$$
\begin{aligned}
\mathcal{T}r[\![(te_1\ te_2)^{\texttt{S}}]\!] &= \{g \xrightarrow{\Gamma,\Delta} f \mid \varrho \in \mathcal{P}App[\![te_1]\!], bind(\varrho, te_2) = \langle\!\langle f, ar(f), \Gamma, \Delta\rangle\!\rangle\}\\
\mathcal{T}r[\![(te_1\ te_2)^{\texttt{M}}]\!] &= \{g \xrightarrow{\Gamma,\Delta'} f \mid \varrho \in \mathcal{P}App[\![te_1]\!], bind(\varrho, te_2) = \langle\!\langle f, n, \Gamma, \Delta\rangle\!\rangle,\\
&\qquad \Delta'(f^{(i)}) = \Delta(f^{(i)}) \text{ if } Bt(f^{(i)}) = S \text{ and}\\
&\qquad \Delta'(f^{(i)}) = (\{\}, \{\}) \text{ if } Bt(f^{(i)}) = D\}\\
\mathcal{T}r[\![f^{\texttt{M}}]\!] &= \{g \xrightarrow{\Gamma,\Delta} f\},\ \text{where } \Delta(x) = (\{\}, \{\}) \text{ for all } x \in Param(f),\\
&\qquad \text{and } \Gamma = \{x \xrightarrow{\overline{\downarrow}} x \mid Bt(x) = S\}.
\end{aligned}
$$

Figure 7.2: $\mathcal{T}r$ collects abstract $\hookrightarrow$ transitions

Definition 7.3.1 (Abstract specialization transitions)

1. An abstract specialization transition has the form $g \xrightarrow{\Gamma,\Delta} f$, where $g, f \in Fun$, Γ is a size-relation graph with arcs connecting static parameters, and Δ maps each f parameter to a dependency description.

2. The abstract specialization transitions for the *2L* program *tp* is denoted $Tr(tp)$, and is given by the union of $\mathcal{T}r[\![te]\!]$ over te of the forms: $(te_1\ te_2)^{\texttt{S}}$, $(te_1\ te_2)^{\texttt{M}}$ and $f^{\texttt{M}}$.

Every possible unfolding transition or memoization transition should be modelled by some abstract specialization transition in $Tr(tp)$. The auxiliary function $\mathcal{P}App$ derives abstract closures of the form $\langle\!\langle f, n, \Gamma, \Delta\rangle\!\rangle$ for each expression te where $bt(te) = S$, in a way that preserves argument-dependency information. For example, given $((f\ te_1)^{\texttt{S}}\ te_2)^{\texttt{S}}$, where f has arity 2, $(f\ te_1)^{\texttt{S}}$ is analyzed as a partial application, so that dependency information for $f^{(1)}$ is collected. Let f have arity 1 instead, and suppose that $(f\ te_1)^{\texttt{S}}$ may return an h closure with 2 arguments. Then in the abstraction of the h closure, the dependencies of $h^{(1)}$ and $h^{(2)}$ are both approximated by the dependencies of $(f\ te_1)^{\texttt{S}}$. In general, a static application can be analyzed as a partial application if we are certain that it does not lead to an unfolding transition during specialization.

Definition 7.3.2 The abstract specialization transition $g \xrightarrow{\Gamma,\Delta} f$ *models* the environment pair (τ, τ') means:

1. For each $x \overset{\downarrow}{\to} y \in \Gamma$: $\#\tau(x) > \#\tau_1(y)$, and for each $x \overset{\bar{\bar{\downarrow}}}{\to} y \in \Gamma$: $\#\tau(x) \geq \#\tau_1(y)$.
2. For $y = f^{(i)}$ and $\Delta(y) = (D^{\curlyvee}, D^{\uparrow})$, one of the following holds.
   - $\tau_1(y)$ is an L expression consisting of just a variable, or
   - $\tau_1(y) \leq_{\mathit{2Val}} \tau(x)$ for some $x \in D^{\curlyvee}$, or
   - $\tau_1(y) \in \mathit{1comps}(V)$, where $V = \{\tau(x) \mid x \in D^{\uparrow}\}$, and *1comps* is defined as follows.

$$
\begin{array}{rcl}
\mathit{1comps}(V) & = & \{\zeta' \mid \mathit{te} \text{ in } \mathit{tp}, \\
 & & \quad \mathit{Dep}(\mathit{te}) = (D^{\curlyvee}, D^{\uparrow}), \\
 & & \quad \tau; \mathit{te} \downarrow \zeta \text{ where } \forall x \in D^{\uparrow} : \tau(x) \in V \text{ and} \\
 & & \qquad\qquad\qquad\quad \forall x \in D^{\curlyvee} : \neg\, \zeta \leq_{\mathit{2Val}} \tau(x), \\
 & & \quad \zeta' \leq_{\mathit{2Val}} \zeta\}.
\end{array}
$$

Remark: Provided that *Dep* is safe, $\mathit{1comps}(V)$ is a downward-closed set including all the values obtainable by a single computation of some expression in *tp*, using the values of V. Easy claim: $\mathit{1comps}(V)$ is finite if V is finite.

Proposition 7.3.3 (Safety of $\mathit{Tr}(\mathit{tp})$) Assume that alpha-conversion by SubV is suspended when processing `M`-annotated expressions. Let $\tau; \mathit{te}$ be reachable, where te is a subexpression of te^g. Then if $\tau; \mathit{te} \hookrightarrow \tau_1; \mathit{te}^f$ is a memoization transition or an unfolding transition to f (so te must have one of the forms $(\mathit{te}_1\ \mathit{te}_2)^{\mathtt{S}}$, $(\mathit{te}_1\ \mathit{te}_2)^{\mathtt{M}}$, or $f^{\mathtt{M}}$, there exists some $g \xrightarrow{\Gamma,\Delta} f \in \mathit{Tr}(\mathit{tp})$ that models the environment pair (τ, τ_1).

Justification of Prop. 7.3.3. The safety of $\mathit{Tr}(\mathit{tp})$ depends on the conservativeness of $\mathit{2}\mathcal{G}[\![\mathit{te}]\!]$ in its approximation of the closures bound to *te* during the specialization of *tp* (Prop. 6.8.3). The new element here is the dependency information. We consider the conservativeness of $\mathcal{P}\mathit{App}$ and $\mathcal{T}\mathit{r}$ below.

1. $\mathcal{P}\mathit{App}$ *is conservative*, in the sense that for $\tau; \mathit{te}$ reachable, and $\tau; \mathit{te} \downarrow \mathit{cl} = \langle f, \xi_1, \ldots, \xi_n \rangle$, there is an abstract closure $\langle\!\langle f, n, \Gamma, \Delta \rangle\!\rangle \in \mathcal{P}\mathit{App}[\![\mathit{te}]\!]$ such that $\langle\!\langle f, n, \Gamma \rangle\!\rangle$ is safe for (cl, τ); furthermore for $\Delta(f^{(i)}) = (D^{\curlyvee}, D^{\uparrow})$ and $i = 1, \ldots, n$, either $\xi_i \leq_{\mathit{2Val}} \tau(x)$ for some $x \in D^{\curlyvee}$, or $\xi_i \in \mathit{1comps}(V)$ where $V = \{\tau(x) \mid x \in D^{\uparrow}\}$. The proof of this claim is by induction on the size of *te*. Proceed by case analysis on the form of *te*. Let $\tau; \mathit{te}$ be reachable, and suppose that $\tau; \mathit{te} \downarrow \langle f, \xi_1, \ldots, \xi_n \rangle$ for each case below.
   - $\mathit{te} \equiv (\mathtt{if}\ \mathit{te}_1\ \mathit{te}_2\ \mathit{te}_3)^{\mathtt{S}}$: In this case, either $\tau; \mathit{te}_2$ is reachable and $\tau; \mathit{te}_2 \downarrow \langle f, \xi_1, \ldots, \xi_n \rangle$; or $\tau; \mathit{te}_3$ is reachable and $\tau; \mathit{te}_3 \downarrow \langle f, \xi_1, \ldots, \xi_n \rangle$. By the induction hypothesis, the required $\langle\!\langle f, n, \Gamma, \Delta \rangle\!\rangle$ is in $\mathcal{P}\mathit{App}[\![\mathit{te}_2]\!]$ or $\mathcal{P}\mathit{App}[\![\mathit{te}_3]\!]$, and thus in $\mathcal{P}\mathit{App}[\![\mathit{te}]\!]$.
   - $\mathit{te} \equiv (\mathit{te}_1\ \mathit{te}_2)^{\mathtt{S}}$: By the side condition in the definition of $\mathcal{P}\mathit{App}$ definition for *te* of this form (and using the safety of $\mathit{2}\mathcal{C}\mathit{l}$, Prop. 6.8.3, Part 1), *te* is *not* an unfolding transition. Thus, we deduce that $n > 0$ and $\tau; \mathit{te}_1 \downarrow \mathit{cl} = \langle f, \xi_1, \ldots, \xi_{n-1} \rangle$. By the induction hypothesis, there exists $\varrho = \langle\!\langle f, n-1, \Gamma, \Delta \rangle\!\rangle$ in $\mathcal{P}\mathit{App}[\![\mathit{te}_1]\!]$ such that the following hold.
     - $\langle\!\langle f, n-1, \Gamma \rangle\!\rangle$ is safe for (cl, τ).
     - For $\Delta(f^{(i)}) = (D^{\curlyvee}, D^{\uparrow})$ and $i = 1, \ldots, n-1$, either $\xi_i \leq_{\mathit{2Val}} \tau(x)$ for some $x \in D^{\curlyvee}$, or $\xi_i \in \mathit{1comps}(V)$ where $V = \{\tau(x) \mid x \in D^{\uparrow}\}$.

By definition of $\mathcal{P}App$, $bind(\varrho, te_2) = \langle\!\langle f, n, \Gamma', \Delta' \rangle\!\rangle$ is in $\mathcal{P}App[\![te]\!]$, where dependency function $\Delta' = \Delta[f^{(n)} \mapsto Dep(te_2)]$, and if $bt(te_2) = S$, then $\langle\!\langle f, n, \Gamma' \rangle\!\rangle = ext_{2\mathcal{S}[\![te_2]\!]} \langle\!\langle f, n-1, \Gamma \rangle\!\rangle$, else $\langle\!\langle f, n, \Gamma' \rangle\!\rangle = ext_{\{\}} \langle\!\langle f, n-1, \Gamma \rangle\!\rangle$.

From Part 2 of Prop. 6.8.3 (safety of $2\mathcal{S}$), if $bt(te_2) = S$, then $\langle\!\langle f, n, \Gamma' \rangle\!\rangle$ is safe for (cl', τ), where $cl' = \langle f, \xi_1, \ldots, \xi_n \rangle$. Otherwise, Γ' makes no assertion about $f^{(n+1)}$, which is dynamic. As for the dependency information, we only need to consider $\Delta(f^{(n)})$, which is $Dep(te_2)$ by definition. Let $Dep(te_2) = (D^{\gamma}, D^{\uparrow})$. Then either $\xi_n \leq_{2Val} \tau(x)$ for some $x \in D^{\gamma}$, or it follows logically that $\xi_n \in 1comps(V)$ where $V = \{\tau(x) \mid x \in D^{\uparrow}\}$: To see this, take te in the definition of $1comps$ as te_2.

- For any other te: $\mathcal{P}App$ simply augments the abstract closures given by $2\mathcal{G}$ with dependency information. By Part 3 of Prop. 6.8.3, for $\tau; te \downarrow cl = \langle f, \xi_1, \ldots, \xi_n \rangle$, there is an abstract closure $\varrho = \langle\!\langle f, n, \Gamma \rangle\!\rangle$ in $2\mathcal{G}[\![te]\!]$ safe for (cl, τ). By the definition of $\mathcal{P}App$, $\langle\!\langle f, n, \Gamma, \Delta \rangle\!\rangle$ is in $\mathcal{P}App[\![te]\!]$, where Δ maps each $f^{(i)}$ to $Dep(te)$, for $i = 1, \ldots, n$. It remains to show that $Dep(te)$ has the required property. Let $Dep(te) = (D^{\gamma}, D^{\uparrow})$ and $0 < i \leq n$. Then either $\xi_i \leq_{2Val} \langle f, \xi_1, \ldots, \xi_n \rangle \leq_{2Val} \tau(x)$ for some $x \in D^{\gamma}$, or it follows logically that $\langle f, \xi_1, \ldots, \xi_n \rangle$ is in $1comps(V)$ where $V = \{\tau(x) \mid x \in D^{\uparrow}\}$ (consult the definition of $1comps$). In the latter case, any argument ξ_i is also in $1comps(V)$ since this set is downward-closed.

2. *That $\mathcal{T}r$ is conservative* follows from the conservativeness of $\mathcal{P}App$. The definition of $\mathcal{T}r$ accounts for every possible memoization/ unfolding transition.

   - *Unfolding*: Consider a reachable state $\tau; (te_1\ \ te_2)^{\mathtt{S}}$ where $\tau; te_1 \downarrow cl = \langle f, \xi_1, \ldots, \xi_{ar(f)-1} \rangle$, and $\tau; te_2 \downarrow \xi_{ar(f)} \neq \mathtt{Err}$, leading to an unfolding transition. Let $\tau_1 = \tau[f^{(i)} \mapsto \xi_i]_{i=1,ar(f)}$. By conservativeness of $\mathcal{P}App$, there exists abstract closure $\langle\!\langle f, ar(f)-1, \Gamma, \Delta \rangle\!\rangle$ in $\mathcal{P}App[\![te_1]\!]$ such that $\langle\!\langle f, ar(f)-1, \Gamma \rangle\!\rangle$ is safe for (cl, τ). If $bt(te_2) = S$, the SRG appearing in the abstract specialization transition ϱ for the unfolding is from $ext_{2\mathcal{S}[\![te_2]\!]} \langle\!\langle f, ar(f)-1, \Gamma \rangle\!\rangle$. This SRG safely describes (τ, τ_1) by Prop. 6.8.3, Part 2b (safety of $2\mathcal{S}$). If $bt(te) = D$, the SRG appearing in the abstract specialization transition does not assert any size relation for $f^{(n+1)}$, since $f^{(n+1)}$ is dynamic. Once again, the SRG is safe for (τ, τ_1).

     The abstract transition has dependency function of the form $\Delta[f^{(ar(f))} \mapsto Dep(te_2)]$. Let $Dep(te_2) = (D^{\gamma}, D^{\uparrow})$. Then either $\xi_{ar(f)} \leq_{2Val} \tau(x)$ for some $x \in D^{\gamma}$, or it follows logically that $\xi_{ar(f)} \in 1comps(V)$ where $V = \{\tau(x) \mid x \in D^{\uparrow}\}$ (take te in the definition of $1comps$ as te). The same claim for each of $\xi_1, \ldots, \xi_{ar(f)-1}$ holds by the conservativeness of $\langle\!\langle f, ar(f)-1, \Gamma, \Delta \rangle\!\rangle$. Thus, the abstract specialization transition ϱ models the environment pair (τ, τ_1), and all of Prop. 7.3.3 is satisfied in this case.

   - *Memoization, form 1*: For any reachable state $\tau; (te_1\ \ te_2)^{\mathtt{M}}$ where $\tau; te_1 \downarrow \langle f, \xi_1, \ldots, \xi_n \rangle$ and $\tau; te_2 \downarrow \xi_{n+1} \neq \mathtt{Err}$, argue as before, except note that the parameters $f^{(n+2)}, \ldots, f^{(ar(f))}$ are dynamic by [S8], so no size information is available for them in any SRG.

     The dependency description is set to $(\{\}, \{\})$ in the abstract specialization transition ϱ, for any dynamic parameter (including $f^{(n+2)}, \ldots, f^{(ar(f))}$). This is safe because the updated environment τ_1 is obtained from $\tau[f^{(i)} \mapsto \xi_i]_{i=1,n+1}$ by setting each dynamic $f^{(i)}$ to the expression $f^{(i)}$ (apart from alpha-converting dynamic arguments of partially static closures). Expression values of the form $f^{(i)}$ are explicitly accounted for in Def. 7.3.2, so ϱ indeed models (τ, τ_1).

     The suspension of alpha-conversion is crucial for Prop. 7.3.3; its omission means that closures with (embedded) dynamic arguments must be deemed constructive when they are copied, *with modifications*, at a memoization transition.

   - *Memoization, form 2*: A memoization transition also occurs due to a reachable state of the form $\tau; f^{\mathtt{M}}$. This is modelled by an abstract specialization transition of the form $\varrho = g \xrightarrow{\Gamma,\Delta} f$, where $\Gamma = \{x \stackrel{\ddagger}{\to} x \mid Bt(x) = S\}$ and $\Delta(x) = (\{\},\{\})$ for all $x \in \mathit{Param}(f)$. By [S8], the parameters of f are all dynamic, thus there is no size information for them in Γ. The updated environment τ_1 maps each $f^{(i)}$ to the expression $f^{(i)}$, and this is explicitly allowed for in Def. 7.3.2. Certainly, ϱ models the environment pair (τ, τ_1).

Precision of $Tr(tp)$. SRG analysis employs a "single-level" closure representation, where size information about the arguments captured by a closure is tracked, but there is no direct modelling of the closures occurring within a closure. The proposed approach to dependency information is even less precise. It is the *trivial* extension of a first-order analysis: The modelling of closures is dispensed with except locally. Consider the specialization of the following L program, where every parameter is assumed to be static.

```
f x y = g (f (x ⊖ 1)) y
g k z = (k z)
```

Size analysis yields that `x` is decreased into `x` for the transition to `f` from the body of `g`. On the other hand, `k` is considered constructive in `x`, and `x` is dependent on `k` in the transition to `f` from the body of `g`, so `x` is judged to be possibly auto-constructive. This leads to a curious situation where unbounded sequences of auto-constructions in `x` are deduced to be impossible, because they would give rise to unbounded sequences of size decreases in `x` (see next chapter). This is not an error: The values assigned to `x` are measured in two distinct ways.

The weak dependency analysis is fine for common invocations of functionals like `map` and `fold`, where the closure parameter does not inadvertently become auto-constructive. We can also expect any straightforward descent in a closure's (captured) argument to be detected by SRG analysis. However, there is a common pattern of recursion that causes the dependency analysis to over-approximate the amount of auto-construction in a way that is significant. Consider a recursive function `f` with formal arguments `cst` and `lst`, whose body contains a call to `map` of the form: `map (f cst) lst`. The definition of `map` is the usual one:

```
map clo ls = if (ls = nil) nil
                (clo (hd ls)) : (map clo (tl ls))
```

Assume that every parameter is static. The parameter `cst` may be unchanged throughout the computation of a call to `f`, but it will always be considered auto-constructive. This is because `clo` has a constructive dependence on `cst`, and a part of `clo` is considered to be copied back (non-constructively) to `cst` during the transition due to `(clo (hd ls))`. There is no way to tell, after dependency analysis, that the value of `cst` is not altered between the two assignments, since dependency information is not tracked for any value that is entered into a closure, and subsequently made available when the closure is completed.

Of course, SRG analysis would detect that in the sequence of transitions from `f` to `map` and back to `f`, the value of `cst` is not increased in size. We would like to use this fact to "suppress" the auto-construction in `cst` over that transition sequence. Unfortunately, the size analysis used to create our SRGs is not based on a size function compatible with $\leq_{\mathit{2Val}}$. In anticipation of the needs of variable boundedness analysis, let us make a small modification to the SRG analysis.

Adjustment to size analysis. Recall that for any expression te, $\mathit{2}\mathcal{S}[\![te]\!]$ is the empty set, or a singleton set containing an element of the form $\downarrow(x)$ or $\ddagger(x)$ (not both), where x is a program variable. Augment $\mathit{2}\mathcal{S}[\![te]\!]$ with an element of the form: $\trianglelefteq(x)$, when $\phi(te)$ is a sequence of `hd` and `tl` operations on variable x, e.g., (`hd` (`tl` (`hd` x))). SRG analysis then uses this information to create arcs of the form: $x \stackrel{\trianglerighteq}{\to} y$. The semantics of such arcs is the obvious one: $x \stackrel{\trianglerighteq}{\to} y$ indicates that in the transition $\tau; te \hookrightarrow \tau_1; te_1$ modelled: $\tau(x) \trianglerighteq \tau_1(y)$. The theory of the SRG is easily extended to accommodate such arcs. Naturally, a connected sequence of $\stackrel{\trianglerighteq}{\to}$-arcs is called a $\stackrel{\trianglerighteq}{\to}$-thread.

7.4 A sufficient condition for variable boundedness

Towards a criterion for variable boundedness. Let $Tr = Tr(tp)$ be given.

Definition 7.4.1 (Bounded variation) A parameter y is *bounded-variation (BV)*, or simply *bounded*, if for any initial environment τ_0, the set $\{\tau(y) \mid \tau; te \text{ is reachable from } \tau_0; te^{f_{initial}}\}$ is finite.

Definition 7.4.2 (Observed multipath)

1. A (legal) $Tr(tp)$-multipath has the form:

$$f_0 \xrightarrow{\Gamma_1,\Delta_1} f_1, f_1 \xrightarrow{\Gamma_2,\Delta_2} f_2, \ldots, f_t \xrightarrow{\Gamma_{t+1},\Delta_{t+1}} f_t, \ldots$$

   We also write:

$$f_0 \xrightarrow{\Gamma_1,\Delta_1} f_1 \xrightarrow{\Gamma_2,\Delta_2} f_2 \ldots \xrightarrow{\Gamma_{t+1},\Delta_{t+1}} f_t \ldots$$

2. Let τ_0 be an initial environment. For every $\hookrightarrow$ chain of the form:

$$\tau_0; te^{f_{initial}} \hookrightarrow \tau_1; te_1 \hookrightarrow \ldots \tau_t; te_t \hookrightarrow \ldots$$

   there is a subsequence of states corresponding to the unfolding transitions and memoization transitions:

$$\tau_{i_0}; te^{f_0} \hookrightarrow^{+} \tau_{i_1}; te^{f_1} \hookrightarrow^{+} \ldots \tau_{i_t}; te^{f_t} \hookrightarrow^{+} \ldots$$

   where $i_0 = 0$ and $f_0 = f_{initial}$, and every environment from each τ_{i_t} to $\tau_{i_{t+1}-1}$ is the same. By the safety of Tr, there is an abstract specialization transition in Tr that models the environment pair $(\tau_{i_t}, \tau_{i_{t+1}})$. The abstract specialization transition has the form $f_t \xrightarrow{\Gamma_{t+1},\Delta_{t+1}} f_{t+1}$. Denote it by a_{t+1}. Then $a_1 a_2 \ldots$ is called an *observed* τ_0 *multipath.*

Definition 7.4.3 (Dependency chain)

1. For $a = g \xrightarrow{\Gamma,\Delta} f \in Tr$, $y \in Param(f)$, $\Delta(y) = (D^{\not\uparrow}, D^{\uparrow})$, write $x \xrightarrow{\uparrow}_a y$ if $x \in D^{\uparrow}$, and $x \xrightarrow{\not\uparrow}_a y$ if $x \in D^{\not\uparrow}$.

2. A *dependency chain* is a (finite or infinite) sequence of dependencies:

$$x_0 \xrightarrow{\delta_1}_{a_1} x_1, x_1 \xrightarrow{\delta_2}_{a_2} x_2, \ldots, x_t \xrightarrow{\delta_t}_{a_t} x_t, \ldots .$$

   We also write:

$$x_0 \xrightarrow{\delta_1}_{a_1} x_1 \xrightarrow{\delta_2}_{a_2} x_2 \ldots \xrightarrow{\delta_t}_{a_t} x_t \ldots .$$

3. Let $\mathcal{M}$ be the Tr-multipath $a_1 \ldots a_T$. And let B be a set of known BV parameters. Define the *degree of* x *in* $\mathcal{M}$ *subject to* B, denoted $deg_{x,B}(\mathcal{M})$, as follows.

   - If $T = 0$ then $deg_{x,B}(\mathcal{M}) = 0$.
   - If $a_T = g \xrightarrow{\Gamma,\Delta} f$ and $x \notin Param(f)$, then $deg_{x,B}(\mathcal{M}) = deg_{x,B}(a_1 \ldots a_{T-1})$.

- If $a_T = g \stackrel{\Gamma,\Delta}{\longrightarrow} f$ and $x \in \mathit{Param}(f)$, then consider dependency chains of the form
$$x_t \stackrel{\delta_{t+1}}{\to}_{a_{t+1}} \ldots \stackrel{\delta_T}{\to}_{a_T} x_T = x,$$
where $x_{t+1}, \ldots, x_T$ are not in B. In this case, define $\mathit{deg}_{x,B}(\mathcal{M})$ to be the number of $\delta_i =\uparrow$ for $t < i \leq T$, maximized over all chains of the specified form.

Theorem 7.4.4 Assume that alpha-conversion by SubV at memoization transitions is suspended. Let y be the parameter of interest, and B be a set of known BV parameters. Let τ_0 be a given initial environment, and V be the (finite) set of possible values for B variables in the states reachable from $\tau_0; te^{f_{initial}}$. Suppose there exists K such that for any observed τ_0 multipath $\mathcal{M} = a_1 \ldots a_T$, it holds that $\mathit{deg}_{y,B}(\mathcal{M}) \leq K$ (i.e., the number of constructions leading to y, measured from V values, is limited by K). Then y is BV.

Proof Define a sequence $V_{(i)}$ of *finite* sets of *2Val* values as follows.

$$\begin{array}{lcl} V_{input} & = & \{\tau_0(x) \mid x \in \mathit{Var}\} \\ X & = & \{x \in \mathit{Expr} \mid \mathit{Bt}(x) = D\} \\ V_{(0)} & = & \{\zeta' \mid \zeta \in V_{input} \cup X \cup V \cup \mathit{1comps}(\{\}),\ \zeta' \leq_{2Val} \zeta\} \\ V_{(k+1)} & = & V_{(k)} \cup \mathit{1comps}(V_{(k)}) \end{array}$$

For any $\hookrightarrow$ chain (finite or infinite): $\tau_0; te_0 \hookrightarrow \tau_1; te_1 \hookrightarrow \ldots$, there is a sub-sequence corresponding to unfolding transitions and memoization transitions: $\tau_{i_0}; te_{i_0} \hookrightarrow^+ \tau_{i_1}; te_{i_1} \hookrightarrow^+ \tau_{i_2}; te_{i_2} \hookrightarrow^+ \ldots$, where $i_0 = 0$. By Def. 7.4.2, there is an observed τ_0 multipath $a_1 a_2 \ldots$ representing this transition sequence such that each a_{t+1} models the environment pair $(\tau_{i_t}, \tau_{i_{t+1}})$ (in the sense of Def. 7.3.2). Let $\mathcal{M}_T$ be the T-element initial of $\mathcal{M}$. We will prove that for $d = \mathit{deg}_{y,B}(\mathcal{M}_T)$: $\tau_{i_T}(y) \in V_{(d)}$. Since $d \leq K$ by assumption, it follows that the values ever assigned to y in a $\hookrightarrow$ chain is from a finite set, for the specialization with initial environment τ_0. Therefore y is BV.

The proof is by induction on the value of T. For $T = 0$, $\tau_0(y) \in V_{input} \subseteq V_{(0)} \subseteq V_{(d)}$ for any d. For the inductive case, consider $\mathcal{M}_{T+1} = a_1 \ldots a_T a_{T+1}$. If the target function of a_{T+1} is not the function of y, then $\tau_{i_{T+1}}(y) = \tau_{i_T}(y)$. By the induction hypothesis, $\tau_{i_T}(y) \in V_{(d)}$, where $d = \mathit{deg}_{y,B}(\mathcal{M}_T)$. As $\mathit{deg}_{y,B}(\mathcal{M}_{T+1}) = \mathit{deg}_{y,B}(\mathcal{M}_T)$, in this case, we have the desired conclusion. Therefore, suppose that the target function of a_{T+1} is the function of y. Let $(D^{\overline{\downarrow}}, D^{\uparrow})$ be the dependency description for y according to a_{T+1}. And let $d = \mathit{deg}_{y,B}(\mathcal{M}_{T+1})$. First, if $y \in B$, then $\tau_{i_{T+1}}(y) \in V$ and $V \subseteq V_{(d)}$. So suppose $y \notin B$. Consider $x \in D^{\overline{\downarrow}}$. Since $x \stackrel{\overline{\downarrow}}{\to}_{a_{T+1}} y$, it follows that $\mathit{deg}_{x,B}(\mathcal{M}_T) \leq d$. By the induction hypothesis, $\tau_{i_T}(x) \in V_{(d)}$. Now consider $x \in D^{\uparrow}$. Since $x \stackrel{\uparrow}{\to}_{a_{T+1}} y$, it follows that $\mathit{deg}_{x,B}(\mathcal{M}_T) = d' < d$. By the induction hypothesis, $\tau_{i_T}(x) \in V_{(d')} \subseteq V_{(d-1)}$. By Prop. 7.3.3, $\tau_{i_{T+1}}(y)$ may be an expression consisting of just a variable, and thus in $X \subseteq V_{(d)}$; or $\tau_{i_{T+1}}(y) \leq_{2Val} \tau_{i_T}(x)$ for some $x \in D^{\overline{\downarrow}}$, in which case $\tau_{i_{T+1}}(y)$ is in $V_{(d)}$ because $V_{(d)}$ is downward-closed; or $D^{\uparrow}$ is non-empty and $\tau_{i_{T+1}}(y)$ is in $\mathit{1comps}(V_{(d-1)}) \subseteq V_{(d)}$; or $D^{\uparrow}$ is empty and $\tau_{i_{T+1}}(y)$ is in $\mathit{1comps}(\{\}) \subseteq V_{(d)}$. In any case, $\tau_{i_{T+1}}(y)$ is contained in $V_{(d)}$.

Further considerations about variable boundedness.

1. In spite of Lemma 6.10.1, which makes it unnecessary to prove the BV of dynamic parameters, we will be careful, in our developments, to track the dependencies of dynamic variables. These dependencies are significant because the values of dynamic variables may be used to construct partially static closures. And it is possible for the dynamic part of a partially static closure to "accumulate" during specialization. For example, consider the following L program.

```
goal x = (f (k x))

f cl = (cl nil)

k d s = (f (k (tl d)))
```

For `x` dynamic, a well-annotated version of the above program is the following *2L* program *tp*.

$\texttt{goal x = (f (k x)}^{\texttt{S}}\texttt{)}^{\texttt{M}}$

$\texttt{f cl = (cl nil}^{\texttt{S}}\texttt{)}^{\texttt{S}}$

$\texttt{k d s = (f (k (tl d)}^{\texttt{D}}\texttt{)}^{\texttt{S}}\texttt{)}^{\texttt{M}}$

During specialization, `cl` is successively assigned the values $\langle \texttt{k}, \texttt{x}\rangle$, $\langle \texttt{k}, (\texttt{tl x})\rangle$, $\langle \texttt{k}, (\texttt{tl (tl x)})\rangle$, ... Note that no infinite unfolding is possible, as every expression that could give rise to an unfolding transition to `f` has been annotated `M`. We could force the annotation on $(\texttt{cl nil}^{\texttt{S}})^{\texttt{S}}$ to become `M`, which would cause the context around `x` to be deposited, and thus prevent it from accumulating. However, this reasoning required us to consider accumulating contexts. It is more natural in our framework to regard `cl` as unbounded (which it is), and generalize it.

For the example, if dependence on dynamic parameters were ignored, then `cl` might be mistakenly classified as bounded, because it is not dependent on any static variable. In actual fact, `cl` has a constructive dependence on `d`, due to $(\texttt{tl d})^{\texttt{D}}$, and `d` has a dependence on `cl`, leading to auto-construction in `cl`.

2. Using Theorem 7.4.4, and some suitable criterion to determine that the stipulated bound K exists, a subset of BV parameters (static and dynamic) can be derived. It makes sense to generalize any static parameter not in this set. At some stage of the annotation procedure, every static parameter is proved BV. Then Lemma 6.10.1 implies that the variable boundedness proposition is satisfied. Closures whose dynamic arguments accumulate during specialization would be impossible, because some variable they are assigned to would have been generalized.

3. A final concern is the "suspension" of alpha-conversion in the correctness propositions, and the use of *newvar* identifiers. Can this be a source of problems for variable boundedness? *Claim*: As long as the set of *newvar* identifiers used in a single specialization is finite, the finiteness of reachable evaluation states is not affected. By design, the use of unboundedly many *newvar* identifiers entails static closure values of arbitrary size, assigned to variable(s) during specialization, which contradicts the boundedness of static parameters. This establishes that the use of unboundedly many *newvar* identifiers is impossible.

The next chapter discusses criteria, based on the application of Theorem 7.4.4, that will allow a parameter to be proved BV, based on known bounded static parameters. We study a formulation of the main criterion for deducing variable boundedness investigated by other researchers, and show a natural generalization of that criterion. We argue that all the methods in the literature rely on a condition that is PSPACE-hard to decide. This motivates the search for a PTIME approximation that is sufficiently general. We propose a good candidate, which we believe will be efficient in practice, and illustrate its workings.

Chapter 8

Criteria for Variable Boundedness

8.1 Managing dependency information

All the works on variable boundedness analysis that we are aware of [Hol91, JGS93, AH96, GJ96, Gle99] use essentially the same criterion to ensure that a given parameter is bounded based on known bounded static parameters. A natural starting point for our discussion of criteria for variable boundedness is to formulate that criterion in the present framework. First, we introduce some concepts for managing variable dependencies. Throughout this chapter, let *Tr* be the set of abstract specialization transitions reachable in $Tr(tp)$. (Abstract specialization transitions are described in Def. 7.3.1. Concerning the notion of *reachable* abstract transitions, consult Ch. 4).

Definition 8.1.1 (Dependency graph)

1. The *dependency graph* is a labelled multigraph (V, A), where $V = Var$ is the vertex set and the set $A \subseteq Var \times \{\not\uparrow, \uparrow\} \times Tr \times Var$ contain annotated (directed) arcs, defined as follows.

$$A = \{(x, \delta, a, y) \mid x \xrightarrow{\delta}_a y, a \in Tr\}$$

2. Define the *companion set for* y, denoted C_y, as the strongly-connected component of the dependency graph that contains y.

3. By "x is level with y," we mean that $x \in C_y$. By "x is in-neighbour to y," we mean that $x \notin C_y$, and for some δ and a, $x \xrightarrow{\delta}_a y'$ such that $y' \in C_y$. The relations "level with" and "in-neighbour" can be computed efficiently by adapting the standard graph algorithms [AHU75].

Definition 8.1.2 (Composition of abstract specialization transitions)

1. Define the composition of abstract specialization transitions as follows. For $a, a' \in Tr$, where $a = f \xrightarrow{\Gamma', \Delta'} f'$ and $a' = f' \xrightarrow{\Gamma'', \Delta''} f''$, the composition of a and a', denoted $a; a'$, has the form $f \xrightarrow{\Gamma, \Delta} f''$, where $\Gamma = \Gamma'; \Gamma''$ is the usual SRG composition, and Δ is such that the following hold.

   - $x \xrightarrow{\uparrow}_{a;a'} y$ if for some z: $x \xrightarrow{\delta}_a z$ and $z \xrightarrow{\delta'}_{a'} y$, where one of δ, δ' is $\uparrow$.
   - $x \xrightarrow{\not\uparrow}_{a;a'} y$ if for some z: $x \xrightarrow{\not\uparrow}_a z$ and $z \xrightarrow{\not\uparrow}_{a'} y$, where it *does not hold* that $x \xrightarrow{\uparrow}_{a;a'} y$.

2. Define $\overline{Tr}$ to be the closure of *Tr* under composition. It is the least set satisfying:

$$\overline{Tr} = Tr \cup \{a; a' \mid a, a' \in \overline{Tr}\}.$$

Lemma 8.1.3 Composition of abstract specialization transitions is associative.

Lemma 8.1.4 Let $a = a_1; \ldots; a_T$. Then the following hold.

1. $x_0 \xrightarrow{\delta}_a x_T$ for some δ iff there exist $x_1, \ldots, x_{T-1}$ and $\delta_1, \ldots, \delta_T$ such that for $t = 0, \ldots, T-1$: $x_t \xrightarrow{\delta_{t+1}}_{a_{t+1}} x_{t+1}$.

2. $x_0 \xrightarrow{\uparrow}_a x_T$ iff there exist $x_1, \ldots, x_{T-1}$ and $\delta_1, \ldots, \delta_T$ such that for $t = 0, \ldots, T-1$: $x_t \xrightarrow{\delta_{t+1}}_{a_{t+1}} x_{t+1}$, and for some t: $\delta_t = \uparrow$.

8.2 A domination principle for variable boundedness

The use of higher-order functionals (e.g., `map`) in a recursion tends to introduce unnecessary constructive auto-dependencies. For example, function `f` may recursively inspect the subtrees of a tree argument `tr` by an indirect recursive call of the form: `(map (f cst) (tl tr))`. Let the first formal parameter of `map` be `clo`. Assume that all the parameters are static. Then `clo` has a constructive dependence on `cst`, and `cst` is dependent on `clo` when `clo` is completed in the body of `map`. This leads to a constructive auto-dependency for `cst`, even if `cst` has a constant value during any specialization.

To hedge against such effects, we have introduced the $\stackrel{\triangleright}{\rightarrow}$ arcs. The idea: Some variables are bounded because their values always yield to those of known bounded parameters (in the sense of $\trianglelefteq$). This is regardless of any auto-construction in the dependency graph. Such deductions partly compensate for the confusion of dataflow due to the involvement of closures. In various works on variable boundedness analysis, if y is determined to have values yielding to those of bounded variables, it is concluded directly that y is bounded. We want to achieve this level of precision in our analysis.

Definition 8.2.1 (Function in-neighbours)

1. Let CG denote the call graph. Its vertex set is Fun and its arc set is $A = \{f \to f' \mid f \xrightarrow{\Gamma,\Delta} f' \in Tr\}$.
2. For $f \in Fun$, let SC_f denote the strongly-connected component of the call graph containing f.
3. For $f \in Fun$, define the *in-neighbours* of f, denoted $IN(f)$, as the set of g such that $g \notin SC_f$, and there exists some $f' \in SC_f$ such that $g \to f'$ is an arc of the call graph.

Remark: By assumption [A2], the initial function $f_{initial}$ is not referenced anywhere in the subject program. Thus for any f reachable from $f_{initial}$ in the call graph, $IN(f)$ is non-empty. $IN(f)$ is empty only if f is in an unreachable part of the call graph. By assumption, all of the call graph is reachable.

Theorem 8.2.2 (The domination condition) Let y be the parameter of interest, and B be a set of known BV parameters. Suppose that $y \in Param(f)$ and $IN(f)$ is non-empty. If for $g \in IN(f)$ and $g \xrightarrow{\Gamma,\Delta} f \in \overline{Tr}$, there exists some $x \in B$ such that $x \stackrel{\triangleright}{\rightarrow} y \in \Gamma$ (describe y as *dominated by* the variables of B), then y is BV.

Proof Let τ_0 be any initial environment. And consider a $\hookrightarrow$ chain with initial state $\tau_0; te^{f_{initial}}$. The values of BV variables in the environments of this chain form a finite set. Denote it by V. Define the set $\overline{V} = \{\zeta' \mid \zeta' \trianglelefteq \zeta \text{ where } \zeta \in V\}$. Thus $\overline{V}$ is also finite.

The $\hookrightarrow$ chain induces an observed τ_0 multipath of the form $\mathcal{M} = a_1 a_2 \ldots = f_0 \xrightarrow{\Gamma_1,\Delta_1} f_1, f_1 \xrightarrow{\Gamma_2,\Delta_2} f_2, \ldots$, where $f_0 = f_{initial}$ (see Def. 7.4.2). If f does not appear in this multipath, then y has the value `Err` in every environment of the $\hookrightarrow$ chain (a transition to f is needed to assign y). Otherwise, let f_i be the earliest in-neighbour to f in the sequence $f_0, f_1, f_2, \ldots$. By design, $f_j = f$ implies that $j > i$. By the premise and Lemma 4.1.4, for $f_j = f$, there is a complete $\stackrel{\triangleright}{\rightarrow}$ thread from some $x \in B$ to y over $a_{i+1} \ldots a_j$. By the correctness of the $\stackrel{\triangleright}{\rightarrow}$ arcs, whenever y is assigned in the $\hookrightarrow$ chain (at points corresponding to $f_j = f$), it receives a value in $\overline{V}$. (We omit the details.) Thus y is BV.

Realizing Theorem 8.2.2. Theorem 8.2.2 is directly realizable by a procedure in EXPTIME. We will discuss a polynomial-time approximation later in this chapter.

1. Compute $\overline{Tr}$ by a trite algorithm.
   - Initialization: Include all of Tr in the set being computed.
   - Iteration: For a, a' in this set, make sure that $a; a'$ is included if it is defined.
2. Compute $IN(f)$ directly, and check whether for each $g \in IN(f)$ and $a \in \overline{Tr}$ of the form $g \xrightarrow{\Gamma,\Delta} f$, there exists $x \in B$ such that $x \stackrel{\triangleright}{\rightarrow} y$ is in Γ. If so, declare that y is BV. Otherwise, the status of y is unclear.

8.3 The ISD criterion for variable boundedness (ISDB)

Theorem 8.3.1 Let y be the parameter of interest, and B be a set of known BV parameters. The premise is:

1. For all x in-neighbour to y, we have $x \in B$.

2. For every $a = f \xrightarrow{\Gamma',\Delta'} f' \in \overline{Tr}$ where for some $x \in C_y$: $x \xrightarrow{\uparrow}_a x$, there exists $z \in B$ such that $z \xrightarrow{\downarrow} z \in \Gamma'$ ("every relevant auto-construction has an anchor").

Then for any initial environment τ_0, there exists K such that for any observed τ_0 multipath $\mathcal{M}$, we have $\deg_{y,B}(\mathcal{M}) < K$.

Proof Informally, the condition of decreasing BV parameter sizes places a limit on the amount of growth for any potential y value (measured from bounded in-neighbour values). The theorem is a consequence of the Finite Ramsey's Theorem.

Fix the initial environment τ_0. Now assume that the theorem is false, for a contradiction. The theorem is false means that for all $k > 1$, there exists an observed τ_0 multipath $\mathcal{M}$ such that $deg_{y,B}(\mathcal{M}) \geq k$. By the definition of *deg*, $\mathcal{M}$ has the form $\mathcal{M} = \mathcal{M}_0 a_1 a_2 \ldots a_T \mathcal{M}_1$ such that for some $x_0, \ldots, x_T$, we have:

$$x_0 \xrightarrow{\delta_1}_{a_1} x_1, x_1 \xrightarrow{\delta_2}_{a_2} x_2, \ldots, x_{T-1} \xrightarrow{\delta_T}_{a_T} x_T$$

where $x_T = y$, $x_1, \ldots, x_T$ are *not* in B, and at least k of $\delta_1, \ldots, \delta_T$ are $\uparrow$. (Note that $x_1, \ldots, x_T$ are level with y, by Part 1 of the premise.) Let $\delta_{i_1} = \ldots = \delta_{i_k} = \uparrow$. Denote the number of parameters in the program by Q. Suppose that k is sufficiently large, and put $k' = \lceil k/Q \rceil - 1$. Then by the pigeonhole principle, there are $k'+1$ of $x_{i_1}, \ldots, x_{i_k}$ equal to one another. Let $x_{j_1} = \ldots = x_{j_{k'+1}} = x \in C_y$. For each $a_{(t)} = a_{j_t+1}; \ldots ; a_{j_{t+1}} \in \overline{Tr}$, where $0 < t \leq k'$, Lemma 8.1.4 implies that $x \xrightarrow{\uparrow}_{a_{(t)}} x$.

Applying the Finite Ramsey's Theorem: Given any $a_{(1)}, \ldots a_{(k')}$, define the *property* P_a as a set of distinct pairs of natural numbers given by $P_a = \{\{t, t'\} \mid t < t',\ a_{(t+1)}; \ldots ; a_{(t')} = a\}$. Then the set of properties:

$$\{P_a \mid a \in \overline{Tr}, x \xrightarrow{\uparrow}_a x \text{ for } x \in C_y\}$$

is always (for any $a_{(t)}$-sequence) a set of mutually exclusive properties covering every pair of distinct elements of $[0, k']$. The Finite Ramsey's Theorem [Ram30] states that for k' sufficiently large, given any K, it is possible to find a K-subset S of $[0, k']$ that is *homogeneous*, i.e., such that every pair of distinct elements of S is in the same P_a. The size of k' needed for this depends on K and the cardinality of $\{P_a\}$.

In other words, for any K at all, we can find $a_{(1)}, \ldots, a_{(k')}$ that contains K consecutive continuous sections such that the composition of the abstract specialization transitions in each section is the same $a^\star \in \overline{Tr}$, where $x \xrightarrow{\uparrow}_{a^\star} x$ for some $x \in C_y$. Formally, there exist indices $i_0, \ldots, i_K$ and $a^\star$ as described such that for $0 \leq t < K$: $a_{(i_t+1)}; \ldots ; a_{(i_{t+1})} = a^\star$. By the premise of the theorem, $a^\star$ has SRG with some arc $z \xrightarrow{\downarrow} z$, where z is in B. This means that in the original *observed* τ_0 multipath $\mathcal{M} = a_1 \ldots a_T$ (from which $a_{(1)}, \ldots, a_{(k')}$ was derived), it can be found K consecutive continuous sections, where each section has a descending thread from z to z (using Lemma 4.1.4). This induces a sequence of z values whose sizes are non-increasing, and decrease at least K times. The boundedness of z means that the set of all z values for the specialization with τ_0 is finite, so for some H, the number of decreases is bounded by H. By making K large enough (possible since T is arbitrary), the boundedness of z is violated. This gives the required contradiction.

Corollary 8.3.2 In Theorem 8.3.1, for the same conclusion, we may establish the premise with respect to a simplified *Tr*, where $x \xrightarrow{\delta}_a x'$ does not hold for $x' \in B$. (In other words, it is valid to treat any $x' \in B$ like a program constant; its dependencies are not significant.)

Proof Note that $x_0 \xrightarrow{\delta_1}_{a_1} x_1, x_1 \xrightarrow{\delta_2}_{a_2} x_2, \ldots, x_{T-1} \xrightarrow{\delta_T}_{a_T} x_T$ ($x_T = y$) in the proof of Theorem 8.3.1 is a valid dependency chain in the simplified *Tr* where $x \xrightarrow{\delta}_a x'$ does not hold for $x' \in B$ (since $x_1, \ldots, x_T$ are not in B). Altering *Tr* affects the meaning of C_y and the in-neighbour relation, which are mentioned in the premise, but in a way that does not disturb the argument in the proof of Theorem 8.3.1.

Criterion 8.3.3 (ISDB) Let y be a given parameter. The in-situ descent criterion for variable boundedness (ISDB) is: For every $a = f \xrightarrow{\Gamma',\Delta'} f' \in \overline{Tr}$, where for some $x \in C_y$, we have $x \xrightarrow{\uparrow}_a x$, there exists a BV parameter z such that $z \xrightarrow{\downarrow} z \in \Gamma'$.

Corollary 8.3.4 Let the parameter y be given. If for every x in-neighbour to y, x is BV, and y satisfies the ISDB criterion, then y is BV.

Proof The corollary is an immediate consequence of Theorem 8.3.1 and Theorem 7.4.4.

Realizing of ISDB. Corollary 8.3.4 is realizable in a straightforward way.

1. Compute $\overline{Tr}$ by a trite algorithm.
2. Given parameter y and the set B of known BV variables, to decide if y is BV, first check whether each x that is in-neighbour to y is in B. If so, obtain the companion set C_y, and check, for each $a = f \xrightarrow{\Gamma',\Delta'} f' \in \overline{Tr}$ where $x \xrightarrow{\uparrow}_a x$ for some $x \in C_y$, whether there exists $z \in B$ such that $z \xrightarrow{\downarrow} z \in \Gamma'$. If this holds, declare that y is BV. Otherwise, the status of y is unclear.

For collecting a set of BV variables, it makes sense to begin by working out the strongly-connected components of the dependency graph, and sorting them in reverse topological order. All this can be done efficiently [AHU75]. By inspecting the components in reverse topological order, any in-neighbour to a variable is processed before the variable. Clearly, an entire component can be classified at once.

As discussed in Ch. 6, any remaining static parameter not classified BV should be generalized. At some stage, every static parameter will be determined to be BV (at worst, the set of static parameters becomes empty). At this point, the variable boundedness proposition is satisfied, according to Lemma 6.10.1. By all the mechanisms that have been put in place to ensure finite unfolding, congruence, and other safety requirements, every specialization is now guaranteed to terminate, and produce a correct residual program. Hopefully, the residual program performs better than the subject program on at least some inputs.

Complexity of ISDB. The straightforward procedure for deciding ISDB is to compute $\overline{Tr}$ and check for each $a = f \xrightarrow{\Gamma',\Delta'} f' \in \overline{Tr}$, whether $x \xrightarrow{\uparrow}_a x$ for some $x \in C_y$ implies there exists $z \in B$ such that $z \xrightarrow{\downarrow} z \in \Gamma'$. This procedure is EXPTIME, as $\overline{Tr}$ could be exponentially large. Since the violation of ISDB can clearly be decided in non-deterministic polynomial-space, by standard techniques, it is possible to check the satisfaction of ISDB in PSPACE, although in practice, the overhead and effort for doing so probably makes it not worth the while. More interestingly: Can we do better than PSPACE?

As we have shown in Ch. 5, given any boolean program b, it is possible to construct a first-order subject program p with corresponding abstract function-call transitions Tr' (without dependency information), such that Tr' fails the ISD *termination* criterion just when b terminates. The program p uses a single function identifier `F`, and each actual argument of an `F` call is a constant, a variable $\texttt{F}^{(i)}$, or a unary destructive operation of the form (`tl` x), where x is some $\texttt{F}^{(i)}$. Thus p has a particularly simple form.

Modify the program p as follows. Augment the parameter set *Param*(`F`) with a "recursion depth" parameter `dp`. In every `F` call, make the actual argument for `dp` the expression `(dp + 1)`. Suppose that every parameter of `F` is static. An initial annotation of p classifies every expression and parameter as static. The abstract specialization transitions are the abstract function-call transitions of Tr', augmented with dependency information. For $\texttt{F} \xrightarrow{\Gamma} \texttt{F} \in Tr'$, include in the set *Tr* of abstract specialization transitions $a = \texttt{F} \xrightarrow{\Gamma,\Delta} \texttt{F}$, where Δ is such that $\texttt{dp} \xrightarrow{\uparrow}_a \texttt{dp}$. For any other required dependency $x \xrightarrow{\delta}_a y$, it is safe to take $\delta = \not\uparrow$ (since static arguments of the form x or (`tl` x) have no constructive dependence on any variable).

It is clear that every parameter other than `dp` is BV (no auto-construction is possible for them in *Tr*). Apply ISDB to decide whether `dp` is BV, based on the boundedess of the other parameters. Now, $C_{\texttt{dp}} = \{\texttt{dp}\}$. We have to determine whether for every $a = \texttt{F} \xrightarrow{\Gamma,\Delta} \texttt{F} \in \overline{Tr}$ (every a in $\overline{Tr}$ is such that $\texttt{dp} \overset{\uparrow}{\to}_a \texttt{dp}$), there is an arc of the form $z \overset{\downarrow}{\to} z$ in Γ, where $z \neq \texttt{dp}$. The condition is fulfilled just when Tr' satisfies the ISD termination criterion, which is just when the boolean program b fails to terminate. Therefore, any application of ISDB entails the solution to a problem that is complete for $\overline{\text{PSPACE}} = \text{PSPACE}$. We conclude that ISDB is PSPACE-hard. As it is decidable in PSPACE, it is a PSPACE-complete condition.

8.4 The size-change criterion for variable boundedness (SCTB)

For practical application, it is desirable to find a PTIME approximation to ISDB. We propose one by analogy with the SCP termination criterion later in the chapter. For now, we point out a natural generalization of last section's condition, made apparent by the application of the Finite Ramsey's Theorem in the correctness proof. The observation is: The abstract transition $a^\star$ in the proof of Theorem 8.3.1 is idempotent. Thus, Theorem 8.3.1 can be strengthened as follows.

Theorem 8.4.1 Let y be the parameter of interest and B be a set of known BV parameters. The new premise is:

1. For all x in-neighbour to y, x is in B.
2. For every $a = f \xrightarrow{\Gamma',\Delta'} f' \in \overline{Tr}$ where for some $x \in C_y$: $x \overset{\uparrow}{\to}_a x$ *and* $a = a; a$, there exists $z \in B$ such that $z \overset{\downarrow}{\to} z \in \Gamma'$.

Then for any initial environment τ_0, there exists some value K such that for all observed τ_0 multipath $\mathcal{M}$, $deg_{y,B}(\mathcal{M}) < K$.

Proof The proof is the same as Theorem 8.3.1, except note that the Finite Ramsey's Theorem gives a sequence of indices $i_0, \ldots, i_K$ in $[0, k']$ such that $a_{(i_0+1)}; \ldots; a_{(i_1)} = a_{(i_1+1)}; \ldots; a_{(i_2)} = \ldots = a^\star$ (these are the consecutive continuous sections of the sequence $a_{(1)}, \ldots, a_{(k')}$ described in the proof of Theorem 8.3.1), *and* as i_0 and i_2 are also in the same $P_{a^\star}$, it holds that $a^\star = a_{(i_0+1)}; \ldots; a_{(i_2)}$, so:

$$\begin{aligned} a^\star &= a_{(i_0+1)}; \ldots; a_{(i_2)} \\ &= (a_{(i_0+1)}; \ldots; a_{(i_1)}); (a_{(i_1+1)}; \ldots; a_{(i_2)}) \\ &= a^\star; a^\star. \end{aligned}$$

Thus the premise in the present theorem (*weakened* from the one in Theorem 8.3.1) still gives the necessary unbounded descent in some BV variable, for obtaining a contradiction.

Corollary 8.4.2 In Theorem 8.4.1, for the same conclusion, we may establish the premise with respect to a simplified *Tr*, where $x \overset{\delta}{\to}_a x'$ does not hold for $x' \in B$.

Proof Refer to Corollary 8.3.2 and its proof.

Criterion 8.4.3 (SCTB) Let y be a given parameter. The size-change criterion for variable boundedness (SCTB) is: for every $a = f \xrightarrow{\Gamma',\Delta'} f' \in \overline{Tr}$ such that $a = a; a$, and for some $x \in C_y$, $x \overset{\uparrow}{\to}_a x$, there exists a BV parameter z such that $z \overset{\downarrow}{\to} z \in \Gamma'$.

Corollary 8.4.4 Let parameter y be given. If for every x in-neighbour to y, x is BV, and y satisfies the SCTB criterion, then y is BV.

Proof The corollary is an immediate consequence of Theorem 8.4.1 and Theorem 7.4.4.

Complexity of SCTB. The SCTB criterion is realizable the same way as the ISDB criterion. It is easy (and low-order polynomial-time) to check whether any given a is idempotent. The non-idempotent a's are simply ignored. The amended procedure is EXPTIME, but SCTB can be decided in PSPACE, by the same reasoning used to deduce that ISDB is in PSPACE.

Let b be a boolean program. We have mentioned the construction of a subject program p whose annotated version tp has abstract specialization transitions Tr and a parameter `dp` such that `dp` satisfies ISDB just when b fails to terminate. In particular, if b fails to terminate, the ISDB criterion holds for `dp`. This implies that the SCTB holds for `dp` too, since the latter specifies a (strictly) weaker condition. Conversely, if b terminates, there is some $a = \mathtt{F} \overset{\Gamma^\star,\Delta^\star}{\longrightarrow} \mathtt{F} \in \overline{Tr}$ where $\mathtt{dp} \overset{\uparrow}{\rightarrow}_a \mathtt{dp}$, such that $\Gamma^\star$ has no arc of the form $z \overset{\downarrow}{\rightarrow} z$. By design (refer to the construction of the original abstract function-call transitions in the PSPACE-hardness proof of ISD termination, Theorem 5.4.1), $\Gamma^\star$ is idempotent. It is a simple consequence of the Infinite Ramsey's Theorem that for all a there exists n such that a^n is idempotent. Now, a^n has the form $\mathtt{F} \overset{\Gamma^\star,\Delta'}{\longrightarrow} \mathtt{F}$ for some Δ'. So here is an element of $\overline{Tr}$ that is idempotent and such that $\mathtt{dp} \overset{\uparrow}{\rightarrow}_{(a^n)} \mathtt{dp}$, but whose SRG $\Gamma^\star$ contains no arc of the form $z \overset{\downarrow}{\rightarrow} z$. Therefore, the SCTB criterion is violated for `dp`.

As the SCTB criterion for `dp` holds just when b fails to terminate, any decision procedure for SCTB entails the solution to a problem that is complete for $\overline{\text{PSPACE}}$ = PSPACE. We conclude that SCTB is PSPACE-hard. As it can be decided in PSPACE, it is a PSPACE-complete condition.

Precision of the generalization. The difference between SCTB and ISDB is in their handling of descent among "permuted arguments." Consider the following (essentially) first-order program, where every parameter is assumed to be static.

```
p m n r dp = if ... (p n (m ⊖ 1) r (dp + 1))
                    (p (r ⊖ 1) m n (dp + 1))
```

The parameters `r`, `m`, `n` are clearly BV. Is `dp` BV? The parameter `dp` has companion set {`dp`}. There are two abstract specialization transitions in Tr (though more in $\overline{Tr}$). Each abstract transition a in $\overline{Tr}$ is such that $\mathtt{dp} \overset{\uparrow}{\rightarrow}_a \mathtt{dp}$. The idempotent ones each has SRG with an arc of the form $x \overset{\downarrow}{\rightarrow} x$ where x is `r`, `m`, or `n`, but there certainly exists an abstract transition a (any one in Tr qualifies) such that $\mathtt{dp} \overset{\uparrow}{\rightarrow}_a \mathtt{dp}$, but a has SRG with no arc of the form $x \overset{\downarrow}{\rightarrow} x$.

The example is adapted from the one intended to demonstrate the difference between the SCT and ISD termination criteria. Point of the present example: Both variable boundedness criteria attempt to show that unbounded sequences of construction in `dp` would give rise to unbounded sequences of size decreases for a bounded variable's values, which is impossible, except the SCTB criterion does not insist on in-situ descent in every loop. It is successful above for this reason.

8.5 The polytime criterion for variable boundedness (SCPB)

In Ch. 4, we proposed a polynomial-time approximation to the SCT termination criterion, which we called the SCP (size-change polytime) criterion. In this section, we adapt the SCP criterion for variable boundedness analysis.

Important assumption. The SCP condition operates under the assumption that Tr has SRGs that are fan-in-free:

$$\text{For } a = f \overset{\Gamma',\Delta'}{\longrightarrow} f' \in Tr,\ \Gamma' \text{ is fan-in free, i.e., } x \overset{r}{\rightarrow} y, x' \overset{r'}{\rightarrow} y \in \Gamma' \text{ implies that } x = x'.$$

It so happens that every Tr generated by the methods described in this work already has the property. Any SRG produced using the size analysis $\mathcal{S}$ is fan-in-free because the result of an expression is either deemed to have size no greater than *or* strictly less than a particular variable's value. Otherwise, no size information is available for that expression. We do not accurately model, for instance, the result of `(min x y)`, where `min` computes the minimum of two natural-number arguments, and returns `Err` for any other inputs.

The assumption is natural when the size analysis does not take the conditions of `if` constructs into account. If it does, "lateral size relations" (size relations among argument values in a single environment) are important to record information like: $\#\tau(x_1) \geq \#\tau(x_2)$ holds for any environment τ that gives rise to a transition at a particular program point. Then if it is known that $\#\tau(x_2) \geq \#\tau_1(x_3)$, where τ_1 is the updated environment after the transition, it can be deduced that $\#\tau(x_1) \geq \#\tau_1(x_3)$. In this case, fan-ins arise naturally.

8.5.1 Analysis by component

The SCP termination criterion was conceived to avoid computing $\overline{Tr}$, which could be exponentially large. The motivation behind SCP was the following observation: For abstract function-call transitions that can loop (i.e., they give rise to an infinite multipath), and that satisfy SCT, it is usually easy to find a "progress point"— some abstract transition whose infinite occurrence in a multipath causes infinite descent. SCT is established if after repeatedly removing progress points, the remaining set cannot loop.

The analogous observation for SCTB is perhaps too convoluted to state. However, we will see that the size-change polytime criterion for variable boundedness (SCPB) is based on the SCP termination criterion.

Definition 8.5.1 (SCTB violation sets) Let y be a given parameter, and B be the set of known BV parameters.

1. Call any $a = f \stackrel{\Gamma',\Delta'}{\longrightarrow} f' \in \overline{Tr}$ an *SCTB violation witness for y wrt B* if a shows SCTB to be violated for y, i.e., $a = a; a$, and for some $x \in C_y$, $x \stackrel{\uparrow}{\to}_a x$, but there does not exist $z \in B$ such that $z \stackrel{\downarrow}{\to} z \in \Gamma'$.
2. Let $a_1, \ldots, a_T \in Tr$. If $a_1; \ldots; a_T$ is an SCTB violation witness for y wrt B, call $\{a_1, \ldots, a_T\}$ an *SCTB violation set for y wrt B*.

Let y be a given parameter, and B be the set of known BV parameters. A motivation for our polytime approximation of SCTB is the following observation. For a set of abstract specialization transitions that fail to satisfy SCTB for y, it is usually easy to find an element that for some reason cannot belong to any SCTB violation set for y wrt B: The inclusion of this element might prohibit the auto-construction of any $x \in C_y$ in the corresponding SCTB violation witness, or introduce in-situ descent of a B variable there. The SCTB approximation removes such elements from consideration. If after repeatedly removing members of Tr, nothing remains, SCTB is established for y.

Definition 8.5.2 (Strongly-connected Tr subset) Let $S \subseteq Tr$ and $S \neq \{\}$. Call S *strongly-connected* if the set $\{f, f' \mid f \stackrel{\Gamma',\Delta'}{\longrightarrow} f' \in S\}$ is strongly-connected in the graph with arc set $\{f \to f' \mid f \stackrel{\Gamma',\Delta'}{\longrightarrow} f' \in S\}$.

Definition 8.5.3 (Constructive subset of Tr) $S \subseteq Tr$ is *constructive for x* if there exists an S-multipath $a_1 \ldots a_T$ such that the following hold:

- For each $a \in S$, some $a_i = a$ (the multipath visits every $a \in S$).
- There exist $x_0, \ldots, x_T$ and $\delta_1, \ldots, \delta_T$ such that for $t = 0, \ldots, T-1$: $x_t \stackrel{\delta_{t+1}}{\to}_{a_{t+1}} x_{t+1}$, $x_0 = x_T = x$, and for some t: $\delta_t = \uparrow$.

We recall the definition of "thread-preserving" (TP) parameters for a set S of abstract function-call transitions. These are pairs of the form $f \cdot x \in Fun \times Var$. The TP-parameter $f \cdot x$ signifies that any thread reaching parameter x at function f will be continued by a size-relation arc to another TP-parameter (some x' at f') in an S-multipath.

Definition 8.5.4 (TP-set) Let $S \subseteq Tr$. Let B be the set of known BV variables.

1. $TP_B(S)$ is the greatest fixedpoint solution of the following equation:

$$\begin{array}{rcl} TP_B(S) & = & \{f \cdot x \mid \forall f \xrightarrow{\Gamma',\Delta'} f' \in S : \exists x \xrightarrow{r} y \in \Gamma' \text{ such that} \\ & & f' \cdot y \in TP_B(S) \text{ and } x, y \in B\} \end{array}$$

2. For $f \xrightarrow{\Gamma',\Delta'} f' \in S$, $x \xrightarrow{r} y \in \Gamma'$ such that $f \cdot x, f' \cdot y \in TP_B(S)$, call $x \xrightarrow{r} y$ a *TP-arc*. If $r = \downarrow$, the TP-arc is *descending* or *decreasing*.

3. A connected sequence of TP-arcs in an S-multipath is a *TP-thread*. A maximal connected sequence of TP-arcs in an S-multipath is a *maximal TP-thread*. A TP-thread with a decreasing arc is a *decreasing TP-thread*.

4. $TP_B(S)|_f = \{x \mid f \cdot x \in TP_B(S)\}$ are the *parameters with focus for* f.

$TP_B(S)$ locates data values whose sizes are either preserved or decreased in any specialization transition described by an S element. SCTB is satisfied for y with respect to S means that unbounded sequences of auto-constructions[1] in $x \in C_y$ would give rise to unbounded sequences of size decreases for some bounded-variable values. SCPB looks for such sequences of size decreases among the values located by $TP_B(S)$.

Theorem 8.5.5 Let parameter y be given. Let B be the set of known BV parameters. The premise is: For every $S \subseteq Tr$ that is constructive for some $x \in C_y$, there exists $f \xrightarrow{\Gamma',\Delta'} f' \in S$ such that Γ' has a decreasing TP-arc.

Then provided that Tr has fan-in-free SRGs, the SCTB criterion is satisfied for y, i.e., for every $a = f \xrightarrow{\Gamma',\Delta'} f' \in \overline{Tr}$, where $a = a; a$ and for some $x \in C_y$, $x \xrightarrow{\uparrow}_a x$, there exists $z \in B$ such that $z \xrightarrow{\downarrow} z \in \Gamma'$.

Proof We prove the contrapositive of the implication. Suppose $a^\star = f \xrightarrow{\Gamma^\star,\Delta^\star} f \in \overline{Tr}$ is such that $a^\star = a^\star; a^\star$ and for some $x \in C_y$: $x \xrightarrow{\uparrow}_{a^\star} x$, but $\Gamma^\star$ does not contain any arc of the form $z \xrightarrow{\downarrow} z$ with $z \in B$. We will prove that the premise is violated.

By definition, $a^\star$ is the composition of abstract specialization transitions of Tr. So $a^\star = a_1; \ldots; a_T$ for some $(a)_t$ sequence. Now, by Lemma 8.1.4, it follows from $x \xrightarrow{\uparrow}_{a^\star} x$ that S is constructive for $x \in C_y$. We have to show that S has no descending TP-arc.

Let us consider the TP-arcs of $a^\star$. Let $a_{t+1} = f_t \xrightarrow{\Gamma_{t+1},\Delta_{t+1}} f_{t+1}$ for $t = 0, \ldots, T-1$. It has been explained before (while discussing the SCP termination criterion) that $\|TP_B(S)|_{f_t}\| \leq \|TP_B(S)|_{f_{t+1}}\|$, by the assumption of no fan-in. However since $f_T = f_0$, in fact $\|TP_B(S)|_{f_i}\| = \|TP_B(S)|_{f_j}\|$ for all i, j. It follows that for $t = 0, \ldots, T-1$:

- If $z'_1 \xrightarrow{r_1} z_1, z'_2 \xrightarrow{r_2} z_2 \in \Gamma_{t+1}$ are TP-arcs, either $z'_1 = z'_2$ and $z_1 = z_2$ or $z'_1 \neq z'_2$ and $z_1 \neq z_2$ (so Γ_{t+1} is 1-1);
- For every $z' \in TP_B(S)|_{f_t}$, there exists $z \in TP_B(S)|_{f_{t+1}}$ and r such that $z' \xrightarrow{r} z \in \Gamma_{t+1}$, and for every $z \in TP_B(S)|_{f_{t+1}}$, there exists $z' \in TP_B(S)|_{f_t}$ and r such that $z' \xrightarrow{r} z \in \Gamma_{t+1}$.

In other words, ignoring the relation r on the size-relation arcs, each Γ_{t+1} is a bijection of focus-parameter sets. It is easy to prove that the above properties hold for the SRG of a composition of S elements. In particular, since $a^\star = f \xrightarrow{\Gamma^\star,\Delta^\star} f$, we have:

- For $z'_1 \xrightarrow{r_1} z_1, z'_2 \xrightarrow{r_2} z_2 \in \Gamma^\star$: either $z'_1 = z'_2$ and $z_1 = z_2$ or $z'_1 \neq z'_2$ and $z_1 \neq z_2$.

[1] Recall that this means for every N, it is possible to observe a sequence of more than N such auto-constructions during the specialization.

- For every $z' \in TP_B(S)|_f$, there exists $z \in TP_B(S)|_f$ and r such that $z' \xrightarrow{r} z \in \Gamma^\star$, and there exists $z \in TP_B(S)|_f$ and r such that $z \xrightarrow{r} z' \in \Gamma^\star$.

Since $a^\star$ is idempotent, so is $\Gamma^\star$. Let $z' \xrightarrow{r} z \in \Gamma^\star$. Suppose that $z' \neq z$. By the second item above, $z \xrightarrow{r'} z'' \in \Gamma^\star$ for some z'' and r'. So $z' \xrightarrow{r''} z'' \in \Gamma^\star$ for some r'' (by the definition of composition, and idempotence). If $z'' \neq z$, then the first item above is violated. On the other hand, if $z'' = z$, then both $z' \xrightarrow{r} z$ and $z \xrightarrow{r'} z$ are in $\Gamma^\star$, and the first item is violated anyway. Conclusion: $z' = z$. So $\Gamma^\star$ is of a particularly simple form. It has TP-arcs of the form $z \xrightarrow{r} z$ for each $z \in TP_B(S)|_f$.

Now, assume for a contradiction that some $a_{t+1} = f_t \xrightarrow{\Gamma_{t+1},\Delta_{t+1}} f_{t+1} \in S$ is such that Γ_{t+1} has a descending TP-arc. Then, by the foregoing, we know that there is a descending TP-thread in $a_1 \dots a_T$ extending over the length of the multipath. By Lemma 4.1.4, there is a decreasing TP-arc in $\Gamma^\star$. This arc has the form $z \xrightarrow{\downarrow} z$, where $z \in B$, since all the parameters of TP-arcs are in B. Therefore the supposition at the beginning of the proof is violated.

Conclusion: For the strongly-connected S that is constructive for $x \in C_y$, it is *impossible* to find $f \xrightarrow{\Gamma',\Delta'} f' \in S$ such that Γ' has a descending TP-arc. This means that the premise is violated.

Corollary 8.5.6 In Theorem 8.5.5, for the same conclusion, we may establish the premise with respect to a simplified *Tr*, where $x \xrightarrow{\delta}_a x'$ does not hold for $x' \in B$.

Criterion 8.5.7 (SCPB) Let y be a given parameter. Let B be the set of known BV parameters. The size-change polytime criterion for variable boundedness (SCPB) is: For every $S \subseteq Tr$ constructive for some $x \in C_y$, there exists $f \xrightarrow{\Gamma',\Delta'} f' \in S$ such that Γ' has a descending TP-arc.

Corollary 8.5.8 Assume that *Tr* has fan-in-free SRGs. Let parameter y be given. If for every x in-neighbour to y, x is BV, and y satisfies the SCPB criterion, then y is BV.

Proof By Theorem 8.5.5, given that *Tr* has fan-in-free SRGs, if y satisfies the SCPB criterion, it satisfies the SCTB criterion. If in addition, every x in-neighbour to y is BV, then by Corollary 8.4.4, y is BV.

8.5.2 Deciding SCPB in polytime

Deciding the SCPB criterion for y entails identifying a subset of *Tr* that is constructive for some x in C_y, the companion set of y. Consideration of what it means for a subset of *Tr* to be constructive for x reveals that it is a property related to the strongly-connected subgraphs of the dependency graph. The main work in the SCPB algorithm of Fig. 8.1 is in *SCPB1*, which is called from the main procedure *SCPB*. The argument of this call is the subgraph corresponding to the dependency-graph SCC C_y. The procedure *SCPB1* recursively narrows its attention to smaller subgraphs in search of a subset of the abstract specialization transitions that shows SCPB to be violated for y. The main procedure *SCPB* returns *False* just when such a subset exists.

For any strongly-connected $S' \subseteq S$, the auxiliary function $W(S')$ computes "witnesses" of decreasing TP-arcs among SRGs of S' elements. The recursive procedure *SCPB1* causes *SCPB* to terminate with *False* if a strongly-connected subgraph of the dependency graph is found that can be used to prove the violation of SCPB for y. We recognize such a subgraph by checking that it has an $\uparrow$-annotated arc, and the set S' of abstract specialization transitions that annotate its arcs is such that $W(S')$ is empty. Procedure *SCPB1* hits a base case when the input graph has no $\uparrow$-annotated arc, or when no strongly-connected subgraph remains after removing every arc annotated with a $W(S')$ element. In the latter case, the empty conjunction returns *True*. We have to prove that any subgraph of the dependency graph that can be used to prove the violation of SCPB for y is never missed.

Lemma 8.5.9 *SCPB*(y) returns *True* just when SCPB is satisfied for y.

Proof Clearly, *SCPB* terminates. The only recursion is in procedure *SCPB1*, and every recursive invocation of *SCPB1* causes the size of its input argument to be reduced. Referring to Fig. 8.1, observe

// B contains the set of known BV parameters.

fun $W(S')$ // computes the SCPB "witnesses" among S'
    return $\{f \xrightarrow{\Gamma',\Delta'} f' \in S' \mid \Gamma'$ has a decreasing TP-arc wrt $S'\}$
end fun

fun *SCPB1*$(\mathcal{D}')$
    if $\mathcal{D}'$ has no arc of form $(x, \uparrow, a, x')$ then
      return *True*
    else
      $S' := \{a \mid \mathcal{D}'$ has arc $(x, \delta, a, x')\}$
      if $W(S')$ is empty then
        return *False*
      else
        let $\mathcal{D}''$ have arc (x, δ, a, x') of $\mathcal{D}'$ if $a \notin W(S')$
        return $\bigwedge\{$*SCPB1*$(\mathcal{D}''') \mid U$ is a non-trivial SCC of $\mathcal{D}''$,
                            $\mathcal{D}'''$ is $\mathcal{D}''$ restricted on $U\}$
      end if
    end if
end fun

fun *SCPB*(y)
    let $\mathcal{D}$ have arc (x, δ, a, x') if $x \xrightarrow{\delta}_a x'$, where $x, x' \in C_y$
    return *SCPB1*$(\mathcal{D})$
end fun

Figure 8.1: Algorithm for deciding the SCPB criterion

that in *SCPB1*, $\mathcal{D}''$ always has fewer arcs than $\mathcal{D}'$. Therefore $\mathcal{D}'''$, the argument for the recursive invocation of *SCPB1*, has fewer arcs than $\mathcal{D}'$.

Now, procedure *SCPB* returns *False* only if *SCPB1* has discovered a strongly-connected subgraph $\mathcal{D}'$ of the dependency graph such that $\mathcal{D}'$ has an $\uparrow$-annotated arc, and the set S' of abstract specialization transitions that annotate the arcs of $\mathcal{D}'$ does not contain any element whose SRG has a decreasing TP-arc. We will argue that S' is constructive for some element of C_y. This shows that SCPB is violated for y.

As $\mathcal{D}'$ is strongly-connected, there exists a path $(x_0, \delta_1, a_1, x_1), \ldots, (x_{T-1}, \delta_T, a_T, x_T)$ of $\mathcal{D}'$ such that $x_0 = x_T$, and every arc (x, δ, a, x') of $\mathcal{D}'$ occurs somewhere in the path. This implies a dependency chain: $x_0 \overset{\delta_1}{\rightarrow}_{a_1} x_1, \ldots, x_{T-1} \overset{\delta_T}{\rightarrow}_{a_T} x_T$ such that $x_0 = x_T$, and for some t, $x_t \overset{\uparrow}{\rightarrow}_{a_{t+1}} x_{t+1}$. By definition, S' is constructive for x_0. By design, every vertex of $\mathcal{D}'$ is in C_y. Therefore S' is constructive for some element of C_y.

Finally, we claim that if a strongly-connected subset $S^\star$ of *Tr* exists such that $S^\star$ is constructive for an element of C_y, and every element of $S^\star$ has SRG without a decreasing TP-arc, then *SCPB*(y) evaluates to *False*.

Now, $S^\star$ is constructive for an element of C_y implies the existence of a dependency chain of the form: $x_0 \overset{\delta_1}{\rightarrow}_{a_1} x_1, \ldots, x_{T-1} \overset{\delta_T}{\rightarrow}_{a_T} x_T$ such that $x_0 = x_T = x$ for some $x \in C_y$, $x_t \overset{\uparrow}{\rightarrow}_{a_{t+1}} x_{t+1}$ for some t, and the set of a_t is exactly $S^\star$. The set of arcs $(x_0, \delta_1, a_1, x_1), \ldots, (x_{T-1}, \delta_T, a_T, x_T)$ forms a strongly-connected subgraph $\mathcal{D}^\star$ of the dependency graph. By definition of C_y, each of $x_0, \ldots, x_T$ is in C_y. So $\mathcal{D}^\star$ is included in the graph $\mathcal{D}$ computed by *SCPB*, the one used to invoke *SCPB1*. We will prove that if $\mathcal{D}^\star$ is a subgraph of $\mathcal{D}'$ in the call *SCPB1*$(\mathcal{D}')$, then *SCPB1*$(\mathcal{D}')$ returns *False*. This establishes the claim.

The proof is by induction on the value of $d = \|\#arcs(\mathcal{D}') - \#arcs(\mathcal{D}^\star)\|$, where $\#arcs$ gives the number of arcs in a graph.

- If $d = 0$, we deduce that $\mathcal{D}' = \mathcal{D}^\star$ (since $\mathcal{D}^\star$ is a subgraph of $\mathcal{D}'$ by assumption). The properties of $\mathcal{D}^\star$ imply that *SCPB1* returns *False* in this case.

- Let $d > 0$. The induction hypothesis is: For $\mathcal{D}''' \subseteq \mathcal{D}'$ such that $\mathcal{D}^\star \subseteq \mathcal{D}'''$, if the difference $\|\#arcs(\mathcal{D}''') - \#arcs(\mathcal{D}^\star)\| < d$, then *SCPB1*$(\mathcal{D}''')$ evaluates to *False*.

  Now, there does exist an $\uparrow$-annotated arc in $\mathcal{D}'$, so *SCPB1* has to compute $W(S')$, where S' is the set of abstract specialization transitions that annotate the arcs of $\mathcal{D}'$. If $W(S')$ is empty, then *SCPB1* returns *False*, so assume that $W(S')$ is non-empty.

  By definition, $W(S')$ has all the elements of S' whose SRG has a descending TP-arc wrt S'. It has been proved while discussing the SCP termination criterion that for a smaller set of abstract transitions, the TP-set increases (because it is the result of a greatest fixedpoint calculation). Since $S^\star$ does not feature any descending TP-arc, it cannot contain any abstract transition from $W(S')$. Otherwise, the descending TP-arc of the $W(S')$ element would be inherited by $S^\star$. We conclude that $S^\star$ is a subset of $S' - W(S')$, and $\mathcal{D}^\star$ is a subgraph of $\mathcal{D}''$, obtained from $\mathcal{D}'$ by removing all the arcs annotated with elements of $W(S')$. As $\mathcal{D}^\star$ is strongly-connected, it is included in one of the graphs used to invoke *SCPB1* recursively. Call it $\mathcal{D}'''$.

  Let $d_1 = \#arcs(\mathcal{D}''') - \#arcs(\mathcal{D}^\star)$. Then as $\mathcal{D}^\star \subseteq \mathcal{D}''' \subseteq \mathcal{D}'' \subset \mathcal{D}'$, we deduce that $d_1 < d$. By the induction hypothesis, *SCPB1*$(\mathcal{D}''')$ evaluates to *False*, so *SCPB1*$(\mathcal{D}')$ evaluates to *False*.

Time complexity of SCPB. Each SCC of *Tr* is processed separately. For a particular component S, let $K = \|S\|$, F be the set of source and target functions of S elements, and A be the maximum arity of functions in F. (For many components, K and A would be small.) Now, S refers to $O(K)$ functions, so the (fan-in-free) SRG of each S element may be represented explicitly with only $O(AK)$ parameters and $O(AK)$ arcs, as the parameters of functions not in F can be ignored. This gives S a size of $O(AK^2)$. The dependency graph $\mathcal{D}$ computed by *SCPB* has size $O(A^2K)$.

The number of levels of recursion for *SCPB1* is bounded by K. Different invocations of *SCPB1* at the same level work on different parts of $\mathcal{D}$. Ignoring any calls to $W(S')$, each invocation of *SCPB1* has time complexity linear in the input $\mathcal{D}'$. Therefore, the computation in a single level of recursion, *ignoring* $W(S')$ *calls*, is $O(A^2K)$.

The number of $W(S')$ calls to be handled in one level of recursion is bounded by $O(A^2K)$. The main processing in $W(S')$ is computing $TP_B(S')$. Recall from Ch. 4 that this calculation has time complexity linear in the size of S', which is $O(AK^2)$. Thus all the operations in one level of *SCPB1* invocations can be accomplished in time $O(AK^2 \times A^2K) = O(A^3K^3)$. It follows that the time complexity of *SCPB1* is $O(A^3K^4)$.

The $TP_B(S')$ calculations require an initialization phase, performed once for the SRGs of S, to index all the functions and parameters, and set up the necessary indexing tables. However, its cost of $O(A^2K^2)$ is subsumed by the cost of evaluating *SCPB1*. Setting up the dependency graph $\mathcal{D}$ (computed by *SCPB*) in a manner suitable for working out SCCs efficiently requires time quadratic in the size of $\mathcal{D}$. This adds another cost of $O(A^4K^2)$.

8.6 Variable boundedness analysis in polytime

8.6.1 The domination condition, revisited

Recall that the domination condition refers to $\overline{Tr}$ (specifically the SRGs of abstract specialization transitions in $\overline{Tr}$). It is interesting to ask whether this (apparently straightforward) condition can be decided in polynomial-time. The assumption that SRGs have no fan-ins suggests the following solution.

Let $y \in \mathit{Param}(f)$ be the parameter of interest, and B be the set of known BV parameters. Suppose that $\mathit{IN}(f)$, the set of in-neighbours to f, is non-empty. Consider the procedure below for checking the domination condition.

1. Determine the digraph *CONN* with vertex set $\mathit{Fun} \times (\mathit{Var} \cup \{\epsilon\})$, and arcs:

$$\begin{array}{ll} & \{(h' \cdot z', h \cdot z) \mid h \xrightarrow{\Gamma', \Delta'} h' \in \mathit{Tr}, z \overset{\trianglerighteq}{\rightarrow} z' \in \Gamma'\} \\ \cup & \{(h' \cdot z', h \cdot \epsilon) \mid h \xrightarrow{\Gamma', \Delta'} h' \in \mathit{Tr},\ \neg\exists\, z : z \overset{\trianglerighteq}{\rightarrow} z' \in \Gamma')\} \end{array}$$

2. Traverse a tree rooted at $f \cdot y$ by calling $\mathit{visit}(f \cdot y)$. To evaluate $\mathit{visit}(h \cdot z)$:
   (a) If $z = \epsilon$, or $h \in \mathit{IN}(f)$ and $z \notin B$, return *False*, else continue.
   (b) If $h \in \mathit{IN}(f)$ and $z \in B$, return *True*, else continue.
   (c) Mark $h \cdot z$ as *seen*.
   (d) For each $h_1 \cdot z_1$ such that $(h \cdot z, h_1 \cdot z_1)$ is an arc of *CONN* and $h_1 \cdot z_1$ is not marked *seen*:
      i. Call $\mathit{visit}(h_1 \cdot z_1)$.
      ii. If the result is *False*, return *False*, else continue.
3. Declare y BV if $\mathit{visit}(f \cdot y)$ returns *True*, otherwise the status of y is unclear.

The digraph *CONN* indicates how z' at h' may be connected by a $\overset{\trianglerighteq}{\rightarrow}$ arc leftwards in a multipath; $z = \epsilon$ indicates that a $\overset{\trianglerighteq}{\rightarrow}$ thread may begin with z' at h' *after* the start of the multipath. The recursive procedure $\mathit{visit}(h \cdot z)$ in Step 2 is standard depth-first traversal. The leaves of the tree traversed by $\mathit{visit}(f \cdot y)$, when it returns *True*, have the form $h \cdot z$, where h in the same SCC as f, or $z \in B$.

The next lemma facilitates the correctness proof of the above procedure for checking the domination condition. For the remainder of this section, an arc refers to a $\overset{\trianglerighteq}{\rightarrow}$-arc, and a thread refers to a $\overset{\trianglerighteq}{\rightarrow}$-thread. We loosely refer to an arc of an abstract transition's SRG as an arc of the abstract transition.

Lemma 8.6.1 (Property of depth-first traversal of *CONN*)

1. Each branch of the tree traversed by $\mathit{visit}(f \cdot y)$ ending in $h \cdot z$, where $z \neq \epsilon$, corresponds to a complete thread in some *Tr*-multipath from h to f, connecting z to y.

2. Suppose that *visit*$(f \cdot y)$ returns *True*. Then any *Tr*-multipath from h to f, where h is in the same strongly-connected component of the call graph as f, has a complete thread from some z to y, such that $h \cdot z$ is a node in the tree traversed by *visit*$(f \cdot y)$.

Proof The first item is proved by induction on the length of the branch the tree traversed by *visit*$(f \cdot y)$. The case for a branch of length 1 is clear from the definition of *CONN*. Suppose that the claim holds for some branch of the tree ending in $h' \cdot z'$, where $z' \neq \epsilon$, and suppose that the depth-first traversal extends this branch by a single arc to $h \cdot z$. By definition of *CONN*, there exists an abstract transition from h to h' having an arc from z to z'. By the induction hypothesis, there is a multipath from h' to f with a complete thread connecting z' to y. Therefore, there exists a multipath from h to f with a complete thread connecting z to y.

The second item is proved by induction on the length of the multipath. For a multipath of length 1 (i.e., a single abstract transition) from h to f, where h is in the same strongly-connected component as f, if *visit*$(f \cdot y)$ returns *True*, by the definition of *visit* and *CONN*, such an abstract transition has an arc from some z to y such that $h \cdot z$ is a descendant of $f \cdot y$ in the tree traversed by *visit*$(f \cdot y)$.

For the inductive case, consider a multipath, of length greater than 1, from h to f where h is in the same strongly-connected component as f. Suppose that the first abstract transition a in the multipath is from h to h'. As h' is in the same strongly-connected component as f (because h is, and h' is on a path between h and f), by the induction hypothesis, there is some $h' \cdot z'$ in the tree traversed by *visit*$(f \cdot y)$ such that a complete thread connects z' to y in the multipath from h' to f. By the definition of *visit* and *CONN*, and the fact that *visit*$(f \cdot y)$ returns *True*, there is an arc from some z to z' in the abstract transition a. Thus the entire multipath (including a) has a complete thread from z to y. If $h \cdot z$ is not an immediate descendant of $h' \cdot z'$ in the tree traversed by *visit*$(f \cdot y)$, it must occur elsewhere in the tree.

Lemma 8.6.2 Given the parameter $y \in \mathit{Param}(f)$, and the set B of known BV parameters, the procedure based on *visit* declares that y is BV just when the domination condition for y is satisfied. (See Theorem 8.2.2.)

Proof Suppose that the procedure declares that the status of y is unclear. Then a branch has been found in the tree traversed by *visit*$(f \cdot y)$ leading to some $h \cdot \epsilon$, or some $h \cdot z$ where $h \in \mathit{IN}(f)$ and $z \notin B$. Argue that the domination condition for y is violated, as follows.

- *Case 1*: Suppose that a branch has been found that leads to some $h \cdot \epsilon$. If this branch has length 1, there is an abstract transition a in *Tr* with source function h and target function f not containing any arc joined to y. If $h \in \mathit{IN}(f)$, $a \in \overline{\mathit{Tr}}$ violates the domination condition for y. Otherwise, consider a multipath from some $h'' \in \mathit{IN}(f)$ to h. (Such a multipath always exists.) Let a'' be the composition of this multipath's elements. Then, $a''; a \in \overline{\mathit{Tr}}$ has no arc joined to y, violating the domination condition for y.

  Now suppose that the branch leading to $h \cdot \epsilon$ has length greater than 1. Let $h \cdot \epsilon$ be immediately preceded by $h' \cdot z'$ in the branch. By Lemma 8.6.1, there is a multipath from h' to f with a complete thread connecting z' to y. By definition of *CONN*, there is an abstract transition a from h to h' without any arc joined to z'. Now, by the assumption of no fan-in, the complete thread from z' to y over the multipath from h' to f is the unique one connected to y at the end of the multipath. By Lemma 4.1.4, the only arc joined to y in the composition of this multipath's elements, call it a', originates from z'. Thus, $a; a'$ has no arc joined to y. If $h \in \mathit{IN}(f)$, $a; a' \in \overline{\mathit{Tr}}$ violates the domination condition for y. If $h \notin \mathit{IN}(f)$, h is in the same strongly-connected component as f, so there exists a multipath from some $h'' \in \mathit{IN}(f)$ to h. Let a'' be the composition of this multipath's elements. Then $a''; a; a' \in \overline{\mathit{Tr}}$ violates the domination condition for y.

- *Case 2*: A branch has been found leading to $h \cdot z$ where $h \in \mathit{IN}(f)$ and $z \notin B$. It follows from Lemma 8.6.1 that there is a multipath from h to f with a complete thread connecting z to y. By the assumption of no-fan-in, this thread is the unique one connected to y at the end of the multipath. By Lemma 4.1.4, the composition of this multipath's elements is an abstract transition $a' \in \overline{\mathit{Tr}}$ from h to f where the only arc joined to y originates from z, which is not in B. Thus, the domination condition is violated for y.

For the reverse direction, suppose that the procedure declares that y is BV. Then $visit(f \cdot y)$ has returned *True*. Consider any $a' \in \overline{Tr}$ with source function $h \in IN(f)$ and target function f. We will argue that a' has an arc from some $z \in B$ to y. Such an a' corresponds to a *Tr*-multipath $\mathcal{M}$ from h to f. Suppose that $\mathcal{M}$ has length 1. Then $\mathcal{M} = a'$, and by the definition of *visit* and *CONN*, and the fact that $visit(f \cdot y)$ has returned *True*, we deduce that a' has an arc from some $z \in B$ to y, as desired. If $\mathcal{M}$ has length greater than 1, consider the multipath that results from omitting the initial abstract transition. The truncated multipath is from some h' to f where h' is in the same strongly-connected component as f. By Lemma 8.6.1, it has a complete thread from some z' to y, where $h' \cdot z'$ is in the tree traversed by $visit(f \cdot y)$. By the definition of *visit* and *CONN*, and the fact that $visit(f \cdot y)$ has returned *True*, the initial abstract transition must have an arc from some $z \in B$ to z'. Therefore, the entire multipath $\mathcal{M}$ has a complete thread from z to y. By Lemma 4.1.4, the abstract transition a' has the arc from z to y (where $z \in B$), as desired.

Teaser: For *Tr* whose SRGs admit fan-ins, the procedure *approximates* the domination condition. Show that a polynomial-time algorithm for deciding the domination condition in this case is highly unlikely because the problem is at least coNP-hard.

Time complexity of the procedure. The program functions can be indexed and the call graph extracted from *Tr* in adjacency-list format in time quadratic in the cardinality of *Tr*. We can also compute the following explicitly in the same time complexity:

- all F such that F is the union of a call-graph SCC and the set of in-neighbours of the SCC's elements, and
- for each F above, $S \subseteq Tr$ such that $S = \{f \xrightarrow{\Gamma', \Delta'} f' \mid f, f' \in F\}$.

Each S will be processed separately. Let $K = \|S\|$, F be the source and target functions of S elements, and A be the maximum arity of functions in F. (For many S, K and A would be small.) Now, S refers to $O(K)$ functions, so each (fan-in-free) SRG of an S element may be represented explicitly with only $O(AK)$ parameters and $O(AK)$ arcs, as the parameters of functions not in F can be ignored. This gives S a size of $O(AK^2)$. The number of function-parameter pairs $h \cdot z$ is $O(AK^2)$.

It takes time $O(A^2K^2)$ to index each parameter x at function f uniquely in the SRGs of S elements. Thereafter, *CONN* can be computed in time linear in its size, i.e., $O(AK^2)$. It is clear that any call to *visit* takes time linear in the size of *CONN*. In fact, a set of function-parameter pairs can be processed in a single depth-first traversal of *CONN*, in time linear in the size of *CONN*, with straightforward modifications to *visit*. To classify $f_1 \cdot y_1, \ldots, f_n \cdot y_n$:

1. Set $H = \bigcup_i IN(f_i)$.
2. Mark as *seen* each $h \cdot z$ such that $z = \epsilon$ or $h \in H$.
3. Mark as *dubious* each $h \cdot z$ such that $z = \epsilon$, or $h \in H$ and $z \notin B$.
4. For i from 1 to n, if $f_i \cdot y_i$ is not marked *seen*, evaluate $visit(f_i \cdot y_i)$, else continue.
5. To evaluate $visit(h \cdot z)$:
   (a) Mark $h \cdot z$ as *seen*.
   (b) For each $h_1 \cdot z_1$ such that $(h \cdot z, h_1 \cdot z_1)$ is an arc of *CONN*:
      i. If $h_1 \cdot z_1$ is not marked *seen*, call $visit(h_1 \cdot z_1)$, else continue.
      ii. If $h_1 \cdot z_1$ is marked *dubious*, mark $h \cdot z$ *dubious* and return, else continue.
6. Declare $f_i \cdot y_i$ BV if it is *not* marked *dubious*, otherwise the status of $f_i \cdot y_i$ is unclear.

8.6.2 Procedure for collecting bounded variables

The following is a straightforward procedure for collecting bounded variables based on the theory in this chapter. The problem: Given an annotated program *tp*, and the binding-times of its variables and expressions, classify as many variables BV as possible. Any static variable not classified BV will be generalized by binding-time analysis. If all the static variables are classified BV, then by Lemma 6.10.1, *tp* satisfies the variable boundedness proposition. The procedure:

1. Classify as BV every $x \in \mathit{Param}(f_{initial})$.
2. Classify as BV every dynamic parameter of function f if it has been ensured that no unfolding transition to f will be possible.
3. Determine the call graph from *Tr*. Compute the SCCs of the call graph, and sort them in reverse topological order. Work out the function in-neighbour relation.

   For each component, determine the set F consisting of the component's functions and their in-neighbours. For each F, determine the set S of abstract specialization transitions in *Tr* whose source and target functions are in F. This gives a list $(S)_i$ of subsets of *Tr*.
4. Apply to each S_i the procedure of Sect. 8.6.1 for checking the domination condition. (Each S_i can be treated like a complete *Tr* by the procedure.)
5. For each S_i:
   (a) Determine the dependency graph where all the arcs into known BV parameters are removed. Compute the SCCs of the dependency graph, and sort them in reverse topological order. Work out the parameter in-neighbour relation.
   (b) Each dependency graph component is processed as follows.
      i. If the component contains any variable with an in-neighbour that is not classified BV, proceed to the next component.
      ii. Otherwise the component is processed as if it has been assigned to $\mathcal{D}$ in the *SCPB* procedure of Fig. 8.1. (In this case, the y in the procedure is irrelevant.) If the SCPB procedure returns *True*, then all the parameters of the component are classified BV.
6. Repeat from Step 4 until no more variable can be classified BV.

Remarks: Step 1 is valid given Assumption [A2]: $f_{initial}$ is not referenced anywhere in the program. Step 2 is valid because if f is a threat to finite unfolding, or f is dangerous to unfold, it must be arranged so that unfolding transitions to f are impossible. Thus a dynamic parameter x of f will never be assigned any value other than x in a transition. Step 5(b)ii is the place where the main variable boundedness criterion, the SCPB, is applied. Step 5b is justified by Corollary 8.5.8.

8.6.3 Effectiveness of the polytime methods

The SCPB criterion is an attractive polynomial-time alternative to the traditional ISDB criterion, if it proves to be sufficiently precise for variable boundedness analysis. It is an analogue of the SCP termination criterion discussed in Ch. 4, which has been demonstrated to subsume many size-descent behaviours encountered in practice. In particular, indirect recursion, "permuted argument" descent and lexical descent are all handled.

The SCPB condition is a strict approximation of SCTB condition, but is incomparable with ISDB. For example, consider the "permuted argument" example:

```
p m n r dp = if ... (p n (m ⊖ 1) r (dp + 1))
                    (p (r ⊖ 1) m n (dp + 1))
```

```
goal p vs =
  ⁰eval p (parm_of (hd p)) vs (body_of (hd p))

eval prg names vals e =
  case e of
    const          : const
    var            : ¹lookup var names vals
    op es          : apply op (²map (eval prg names vals) es)
    if e0 e1 e2    : if (³eval prg names vals e0)
                        (⁴eval prg names vals e1)
                        (⁵eval prg names vals e2)
    let x=e0 in e1 : ⁶eval prg (x : names) (⁷eval prg names vals e0 : vals) e1
    call f es      : ⁸eval prg (⁹parm f prg)
                               (¹⁰map (eval prg names vals) es)
                               (¹¹body f prg)

lookup var1 names1 vals1 =
  if (hd names1 = var1) (hd vals1) (¹²lookup var1 (tl names1) (tl vals1))

body f1 prg1 =
  if (name_of (hd prg1) = f1) (body_of (hd prg1)) (¹³body f1 (tl prg1))

parm f2 prg2 =
  if (name_of (hd prg2) = f2) (parm_of (hd prg2)) (¹⁴parm f2 (tl prg2))

map clo ls =
  if (ls = nil) nil (¹⁵clo (hd ls) : ¹⁶map clo (tl ls))

// param_of, body_of and name_of all involve the application of a sequence
// of destructors to the input argument.
```

Figure 8.2: An interpreter loop

where every parameter is assumed to be static. For the annotated program, `m`, `n` are `r` are level with one another, and `dp` has the companion set $\{$`dp`$\}$. There are no in-neighbours to any of these parameters. By Corollary 8.5.8, `m`, `n` and `r` are BV, since no subset of *Tr* is constructive for any of them. Let us apply SCPB to `dp`. Every non-trivial subset of *Tr* is constructive for `dp`. However, each subset has an element whose SRG contains a descending TP-arc. It follows from Corollary 8.5.8 that `dp` is BV. For this example, the SCPB criterion has performed better than the ISDB criterion, although they are incomparable in general.

Next, we examine whether the polynomial-time methods are able to handle a class of programs of particular interest in offline partial evaluation: the interpreters. The specialization of interpreters is an area where offline partial evaluation has been successful. We show the use of SCPB and the domination condition does not lead to unnecessary generalization for common interpreter loops.

The first interpreter (Fig. 8.2) is for a first-order functional language with static name-binding, taken from [GJ96, Gle99]. The code uses "syntactic sugar." For example, the `case` construct (in pseudo-ML syntax [Pau96]) should be regarded as shorthand for a nested conditional. Callsites (where potential memoization or unfolding transitions occur) are indicated by numerical superscripts next to the rator, for easy reference. For this essentially first-order example, each callsite will correspond to exactly 1 abstract specialization transition.

Example 1: An interpreter loop. We will often write "by SCPB" in our reasoning below. This is understood to be an invocation of Corollary 8.5.8. We describe a function f as "marked for memoization" if it has been ensured that any potential transitions to f are memoization transitions.

Consider the interpreter loop of Fig. 8.2. Assume that `p` is static, while `vs` is dynamic. The program is annotated accordingly. The challenge here is to prove that the static `names` is BV, so that it is not (unnecessarily) generalized. Making `names` dynamic would cause variable dereferencing to be retained in the specialization of the interpreter to an object program.

It is not trivial to prove that `names` is BV because potential transitions at 6 would cause it to be auto-constructive, and while this auto-construction is accompanied by a size decrease in the BV parameter `e`, it is not completely obvious that *every* auto-construction of `names` is accompanied by a size decrease, since any transition at 8 would "reset" the value of `e`. Fortunately, every auto-construction of `names` has to exclude transitions at 8, which resets `names` along with `e`.

To apply the polynomial-time procedure for collecting bounded variables, we first derive the abstract transitions of $Tr(tp)$, based on initial annotations and binding-times that classify the following parameters as static: `prg`, `names`, `e`, `var1`, `names1`, `f1`, `prg1`, `f2`, `prg2`, `clo` and `ls`. This BT division, with `eval` and `map` marked for memoization (the rest of the functions are no threat to finite unfolding– at least for the moment), satisfy all the safety requirements, as well as the finite unfolding proposition. Ideally, every parameter can be certified BV (which they are). Then no generalization is performed, and specialization is permitted to proceed as normal.

The abstract specialization transitions are shown in Fig. 8.3. They have been worked out in a straight-forward way. Note that the results of calls to `parm` and `body` have no constructive dependence on any variable, explaining why the dependency arcs to `names` and `e` in the abstract transition for callsite 8 have been annotated $\not\uparrow$, rather than $\uparrow$. Parameters `vals` and `vals1` are dynamic. Dependency arcs with these parameters as targets only occur for the abstraction of unfolding transitions. This accounts for the $\uparrow$ dependency to `vals1` in the abstract transition for callsite 12. The actual argument here is `(tl vals1)`, which would not have a constructive dependence on any variable for a static `vals1`.

As for the SRG arcs, they are easily seen to represent true size relations. The parameters `vals` and `vals1` do not feature in any SRG because they are dynamic. The ...s represent those arguments "copied" from one evaluation environment to the next. Note that the $\ddagger$-relations originating from `prg` and `names` remain valid for the memoization transitions due to `clo` in `map`'s body (at callsite 15). Any size relation originating from `clo` and `ls` must be eliminated, but it is safe to retain size relations originating from `prg` and `names`. For this example, the $\stackrel{\triangleright}{\to}$ arcs are just the union of the $\stackrel{\downarrow}{\to}$ and $\stackrel{\ddagger}{\to}$ arcs.

Applying the variable boundedness conditions. Classify input parameters `p` and `vs` as BV. Classify `vals` as BV because it is a dynamic parameter of `eval`, which is marked for memoization.

In the call graph, `goal` is in-neighbour to the component {`eval`,`map`}, and `eval` is in-neighbour to the components {`lookup`}, {`parm`} and {`body`}. In any multipath from `goal` to `eval`, there is a complete thread of $\stackrel{\triangleright}{\to}$ arcs from `p` to `prg` (observe the underlined arcs in Fig. 8.3). By correctness of the domination condition, `prg` is BV. Similarly, argue that parameters `e` of `eval` and `ls` of `map` are dominated by `p` at `goal`, and therefore BV; each of `var1`, `f1` and `f2` is dominated by `e` at `eval`, and therefore BV; and each of `prg1` and `prg2` is dominated by `prg` at `eval` and therefore BV.

Now, remove all the dependencies of known BV parameters from the dependency graph. Figure 8.4 shows the part of the remaining graph relevant to the parameters of `goal`, `eval` and `map`. The heavy lines indicate constructive dependencies, and the boxed section is the strongly-connected component containing `names` and `clo`. The important thing to note is that the abstract transition for callsite 8 does not appear in this component. Any strongly-connected subset of the abstract transitions for callsites 2, 3, 4, 5, 6, 7, 10, 15, 16 always has a decreasing TP-arc of the form `e` $\stackrel{\downarrow}{\to}$ `e`, `e` $\stackrel{\downarrow}{\to}$ `ls` or `ls` $\stackrel{\downarrow}{\to}$ `ls`. The in-neighbours to the component's variables are already known to be BV. By SCPB, `names` and `clo` are BV.

The companion set for `names1` is {`names1`}. There does not exist any subset of abstract transitions that is constructive for it, and its only in-neighbour is `names`, which has been classified BV. By SCPB, `names1` is BV. The companion set for `vals1` is {`vals1`}. This dynamic parameter is actually auto-constructive, due to potential unfolding transitions at 12. However, the only set of abstract transitions constructive for it consists of just the abstract transition for callsite 12. And this set has the decreasing TP-arc `names1` $\stackrel{\downarrow}{\to}$ `names1`. The only in-neighbour to `vals1` is `vals`, which has been classified BV. By

Memo at 0	$\mathtt{p} \xrightarrow{\not\uparrow}_a \mathtt{prg}, \mathtt{p} \xrightarrow{\not\uparrow}_a \mathtt{names}, \mathtt{p} \xrightarrow{\not\uparrow}_a \mathtt{e}$
	$\mathtt{p} \xrightarrow{\overline{\overline{\downarrow}}} \mathtt{prg}, \underline{\mathtt{p} \xrightarrow{\trianglerighteq} \mathtt{prg}}, \mathtt{p} \xrightarrow{\downarrow} \mathtt{names}, \mathtt{p} \xrightarrow{\trianglerighteq} \mathtt{names}, \mathtt{p} \xrightarrow{\downarrow} \mathtt{e}, \mathtt{p} \xrightarrow{\trianglerighteq} \mathtt{e} \ldots$
Unfolding at 1	$\mathtt{e} \xrightarrow{\not\uparrow}_a \mathtt{var1}, \mathtt{names} \xrightarrow{\not\uparrow}_a \mathtt{names1}, \mathtt{vals} \xrightarrow{\not\uparrow}_a \mathtt{vals1}$
	$\mathtt{e} \xrightarrow{\overline{\overline{\downarrow}}} \mathtt{var1}, \mathtt{e} \xrightarrow{\trianglerighteq} \mathtt{var1}, \mathtt{names} \xrightarrow{\overline{\overline{\downarrow}}} \mathtt{names1}, \mathtt{names} \xrightarrow{\trianglerighteq} \mathtt{names1}, \ldots$
Memo at 2, 10	$\mathtt{prg} \xrightarrow{\uparrow}_a \mathtt{clo}, \mathtt{names} \xrightarrow{\uparrow}_a \mathtt{clo}, \mathtt{vals} \xrightarrow{\uparrow}_a \mathtt{clo}, \mathtt{e} \xrightarrow{\not\uparrow}_a \mathtt{ls}$
	$\mathtt{e} \xrightarrow{\downarrow} \mathtt{ls}, \mathtt{e} \xrightarrow{\trianglerighteq} \mathtt{ls}, \ldots, \underline{\mathtt{prg} \xrightarrow{\trianglerighteq} \mathtt{prg}}, \ldots$
Memo at 3, 4, 5, 7	$\mathtt{prg} \xrightarrow{\not\uparrow}_a \mathtt{prg}, \mathtt{names} \xrightarrow{\not\uparrow}_a \mathtt{names}, \mathtt{e} \xrightarrow{\not\uparrow}_a \mathtt{e}$
	$\mathtt{prg} \xrightarrow{\overline{\overline{\downarrow}}} \mathtt{prg}, \underline{\mathtt{prg} \xrightarrow{\trianglerighteq} \mathtt{prg}}, \mathtt{names} \xrightarrow{\overline{\overline{\downarrow}}} \mathtt{names}, \mathtt{names} \xrightarrow{\trianglerighteq} \mathtt{names}, \mathtt{e} \xrightarrow{\downarrow} \mathtt{e}, \mathtt{e} \xrightarrow{\trianglerighteq} \mathtt{e}, \ldots$
Memo at 6	$\mathtt{prg} \xrightarrow{\not\uparrow}_a \mathtt{prg}, \mathtt{names} \xrightarrow{\uparrow}_a \mathtt{names}, \mathtt{e} \xrightarrow{\uparrow}_a \mathtt{names}, \mathtt{e} \xrightarrow{\not\uparrow}_a \mathtt{e}$
	$\mathtt{prg} \xrightarrow{\overline{\overline{\downarrow}}} \mathtt{prg}, \underline{\mathtt{prg} \xrightarrow{\trianglerighteq} \mathtt{prg}}, \mathtt{e} \xrightarrow{\downarrow} \mathtt{e}, \mathtt{e} \xrightarrow{\trianglerighteq} \mathtt{e}, \ldots$
Memo at 8	$\mathtt{prg} \xrightarrow{\not\uparrow}_a \mathtt{prg}, \mathtt{prg} \xrightarrow{\not\uparrow}_a \mathtt{names}, \mathtt{e} \xrightarrow{\not\uparrow}_a \mathtt{names}, \mathtt{e} \xrightarrow{\not\uparrow}_a \mathtt{e}, \mathtt{prg} \xrightarrow{\not\uparrow}_a \mathtt{e}$
	$\mathtt{prg} \xrightarrow{\overline{\overline{\downarrow}}} \mathtt{prg}, \underline{\mathtt{prg} \xrightarrow{\trianglerighteq} \mathtt{prg}}, \mathtt{prg} \xrightarrow{\downarrow} \mathtt{names}, \mathtt{prg} \xrightarrow{\trianglerighteq} \mathtt{names}, \mathtt{prg} \xrightarrow{\downarrow} \mathtt{e}, \mathtt{prg} \xrightarrow{\trianglerighteq} \mathtt{e}, \ldots$
Unfolding at 9	$\mathtt{e} \xrightarrow{\not\uparrow}_a \mathtt{f1}, \mathtt{prg} \xrightarrow{\not\uparrow}_a \mathtt{prg1}$
	$\mathtt{e} \xrightarrow{\downarrow} \mathtt{f1}, \mathtt{e} \xrightarrow{\trianglerighteq} \mathtt{f1}, \mathtt{prg} \xrightarrow{\overline{\overline{\downarrow}}} \mathtt{prg1}, \mathtt{prg} \xrightarrow{\trianglerighteq} \mathtt{prg1}, \ldots$
Unfolding at 11	$\mathtt{e} \xrightarrow{\not\uparrow}_a \mathtt{f2}, \mathtt{prg} \xrightarrow{\not\uparrow}_a \mathtt{prg2}$
	$\mathtt{e} \xrightarrow{\downarrow} \mathtt{f2}, \mathtt{e} \xrightarrow{\trianglerighteq} \mathtt{f2}, \mathtt{prg} \xrightarrow{\overline{\overline{\downarrow}}} \mathtt{prg2}, \mathtt{prg} \xrightarrow{\trianglerighteq} \mathtt{prg2}, \ldots$
Unfolding at 12	$\mathtt{var1} \xrightarrow{\not\uparrow}_a \mathtt{var1}, \mathtt{names1} \xrightarrow{\not\uparrow}_a \mathtt{names1}, \mathtt{vals1} \xrightarrow{\uparrow}_a \mathtt{vals1}$
	$\mathtt{var1} \xrightarrow{\overline{\overline{\downarrow}}} \mathtt{var1}, \mathtt{var1} \xrightarrow{\trianglerighteq} \mathtt{var1}, \mathtt{names1} \xrightarrow{\downarrow} \mathtt{names1}, \mathtt{names1} \xrightarrow{\trianglerighteq} \mathtt{names1}, \ldots$
Unfolding at 13	$\mathtt{f1} \xrightarrow{\not\uparrow}_a \mathtt{f1}, \mathtt{prg1} \xrightarrow{\not\uparrow}_a \mathtt{prg1}$
	$\mathtt{f1} \xrightarrow{\overline{\overline{\downarrow}}} \mathtt{f1}, \mathtt{f1} \xrightarrow{\trianglerighteq} \mathtt{f1}, \mathtt{prg1} \xrightarrow{\downarrow} \mathtt{prg1}, \mathtt{prg1} \xrightarrow{\trianglerighteq} \mathtt{prg1}, \ldots$
Unfolding at 14	$\mathtt{f2} \xrightarrow{\not\uparrow}_a \mathtt{f2}, \mathtt{prg2} \xrightarrow{\not\uparrow}_a \mathtt{prg2}$
	$\mathtt{f2} \xrightarrow{\overline{\overline{\downarrow}}} \mathtt{f2}, \mathtt{f2} \xrightarrow{\trianglerighteq} \mathtt{f2}, \mathtt{prg2} \xrightarrow{\downarrow} \mathtt{prg2}, \mathtt{prg2} \xrightarrow{\trianglerighteq} \mathtt{prg2}, \ldots$
Memo at 15	$\mathtt{clo} \xrightarrow{\not\uparrow}_a \mathtt{prg}, \mathtt{clo} \xrightarrow{\not\uparrow}_a \mathtt{names}, \mathtt{ls} \xrightarrow{\not\uparrow}_a \mathtt{e}$
	$\mathtt{prg} \xrightarrow{\overline{\overline{\downarrow}}} \mathtt{prg}, \underline{\mathtt{prg} \xrightarrow{\trianglerighteq} \mathtt{prg}}, \mathtt{names} \xrightarrow{\overline{\overline{\downarrow}}} \mathtt{names}, \mathtt{names} \xrightarrow{\trianglerighteq} \mathtt{names}, \mathtt{ls} \xrightarrow{\downarrow} \mathtt{e}, \mathtt{ls} \xrightarrow{\trianglerighteq} \mathtt{e}, \ldots$
Memo at 16	$\mathtt{clo} \xrightarrow{\not\uparrow}_a \mathtt{clo}, \mathtt{ls} \xrightarrow{\not\uparrow}_a \mathtt{ls}$
	$\mathtt{clo} \xrightarrow{\overline{\overline{\downarrow}}} \mathtt{clo}, \mathtt{clo} \xrightarrow{\trianglerighteq} \mathtt{clo}, \mathtt{ls} \xrightarrow{\downarrow} \mathtt{ls}, \mathtt{ls} \xrightarrow{\trianglerighteq} \mathtt{ls}, \ldots, \underline{\mathtt{prg} \xrightarrow{\trianglerighteq} \mathtt{prg}}, \ldots$

Figure 8.3: Abstract transitions for the interpreter loop of Fig. 8.2

SCPB, `vals1` is BV. (Note that the descent in `names1` is the reason unfolding transitions to `lookup` have been allowed in the first place.) Now every parameter has been certified BV, as desired.

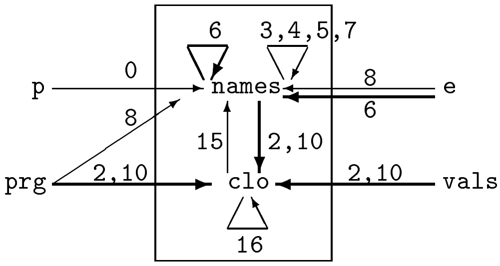


Figure 8.4: Part of the dependency graph for Ex. 1

Working through *SCPB*(`names`). The first application of SCPB in the example showed `names` to be BV. Let us explain how the SCPB condition for `names` is automatically checked by *SCPB*(`names`). Denote the abstract transition for callsite i by a_i.

1. *SCPB* begins by computing $\mathcal{D}$, the dependency graph restricted to vertices of $C_{\texttt{names}}$. In this case, $\mathcal{D}$ is the graph depicted pictorially by the boxed section of Fig. 8.4.

2. *SCPB* calls *SCPB1* with the graph $\mathcal{D}$.

3. Since `names` $\stackrel{\uparrow}{\rightarrow}_{a_6}$ `names`, the procedure *SCPB1* has to compute the witnesses $W(S')$, where $S' = \{a_2, a_3, a_4, a_5, a_6, a_7, a_{10}, a_{15}, a_{16}\}$ are the abstract transitions annotating the arcs of the input graph.

   To compute $W(S')$, it is first necessary to compute $TP_B(S')$. We demonstrate here the brute-force approach for this calculation. A better method is explained in Ch. 4.

   Initialize P to the set of $f \cdot x$, such that f is `eval` or `map`, and x is a bounded static parameter of `eval` or `map`, i.e., a member of $\{\texttt{prg}, \texttt{e}, \texttt{ls}\}$. Remove from P the element $f \cdot x$ if in the SRG of some a_i with source function f and target function f', the parameter x is not connected to any x' such that $f' \cdot x' \in P$. Repeat this until no more element can be removed:

   (a) a_2 does not continue `ls` (at all), so remove $\texttt{eval} \cdot \texttt{ls}$ from P. This leaves the following in P: $\texttt{eval} \cdot \texttt{prg}, \texttt{eval} \cdot \texttt{e}, \texttt{map} \cdot \texttt{prg}, \texttt{map} \cdot \texttt{e}, \texttt{map} \cdot \texttt{ls}$.

   (b) a_{15} does not continue `e` (at all), so remove $\texttt{map} \cdot \texttt{e}$ from P. This leaves the following in P: $\texttt{eval} \cdot \texttt{prg}, \texttt{eval} \cdot \texttt{e}, \texttt{map} \cdot \texttt{prg}, \texttt{map} \cdot \texttt{ls}$.

   With this P, it can be mechanically verified that each $a_i \in S'$ is such that for $f \cdot x \in P$, if a_i has source function f and target function f', then there exists x' such that $f' \cdot x' \in P$. In greater detail, any abstract transition of S' from `eval` to `map` has size-relation arcs from `prg` to `prg` and from `e` to `ls`; any abstract transition from `map` to `eval` has size-relation arcs from `prg` to `prg` and from `ls` to `e`; any abstract transition from `eval` to `eval` has size-relation arcs from `prg` to `prg` and from `e` to `e`. Thus P is indeed a fixedpoint of the $TP_B(S')$ equation.

   The abstract transitions of S' all have descending TP-arcs:

   (a) a_2, a_{10} with source function `eval` and target function `map` both have the arc `e` $\stackrel{\downarrow}{\rightarrow}$ `ls`.

   (b) a_3, a_4, a_5, a_6, a_7 with source function `eval` and target function `eval` all have the arc `e` $\stackrel{\downarrow}{\rightarrow}$ `e`.

   (c) a_{15} with source function `map` and target function `eval` has the arc `ls` $\stackrel{\downarrow}{\rightarrow}$ `e`.

(d) a_{16} with source function `map` and target function `map` has the arc `ls` $\overset{\downarrow}{\rightarrow}$ `ls`.

We determine that $W(S') = S'$.

4. There is therefore no recursive invocation of *SCPB1*. It follows that *SCPB1* returns *True*. Hence the main procedure *SCPB* returns *True*.

Further remarks:

1. Glenstrup [GJ96] has explained how ISDB handles an example analogous to Ex. 1. In fact, the descent that needs to be detected for the BV deduction of `names` is of a particularly simple form. It is in-situ and involves only a parameter in each function (`e` in `eval`, `ls` in `map`).

2. The use of functionals like `map` in different places of the subject program quickly confuses our variable boundedness analysis. This argues for some degree of polyvariance, at least for the common functionals. We do not pursue the issue of polyvariance in this work.

3. Consider changing the expression marked 8 to:

```
8eval prg (append (9parm f prg) names)
          (append (10map (eval prg names vals) es) vals)
          (11body f prg)
```

to implement a dynamic name-value binding policy. (The second argument of the call has changed from `(parm f prg)` to `(append (parm f prg) names)`.) This amendment introduces the abstract transition for callsite 8 to the constructive self-loop for `names` in the graph of Fig. 8.4, and means that there is now a set of abstract transitions (consisting of just the abstract transition for callsite 8) that is constructive for `names`, but has no decreasing TP-arc (because the value of `e` is reset for any transition at callsite 8). This forces `names` to be generalized.

The current annotations and binding-times with `names`, `clo` and `names1` made dynamic, and `lookup` marked for memoization, satisfy all the safety requirements, as well as the finite unfolding proposition. Applying the variable boundedness conditions, we classify `p`, `vs`, `vals`, `prg`, `e`, `ls`, `var1`, `f1`, `prg1`, `f2`, `prg2` as BV, just as before. The remaining parameters are BV as they are dynamic parameters of functions marked for memoization. The parameter `names` has been generalized with good reason: It really may not be bounded for some specialization.

Example 2: Alternative handling of name-value binding. Ex. 1 uses separate `names` and `values` lists for handling name-value bindings. This achieves better binding-time separation than using an association list of names and values (the natural approach), in a partial evaluation framework that does not handle partially static lists. However, *2L* specialization does treat partially static structures, namely, function closures. It is natural in *L* to represent name-value bindings using environments, as in the program of Fig. 8.5.

For `p` static and `vs` dynamic, the following parameters can be classified static without violating safety: `p`, `prg`, `env`, `e`, `f1`, `prg1`, `f2`, `prg2`, `clo`, `ls`, `names`, `x`, `env1`, `name1`, `x1`, `prg3`, `env3`, `x3`, `e3`, `prg4`, `f4`. Unfolding transitions to `eval`, `map`, `doCall`, `doLet`, or `extEnv` appear to risky, as termination following such transitions cannot be guaranteed using our analysis with the abstract transitions derived. In actual fact, unfolding transitions to `extEnv` do not lead to non-termination, because each time `env1` is completed at callsite 18, the subsequent `env1` is a smaller closure. This is some form of size descent. (We will briefly indicate how to capture such descent in the next chapter, as the same situation frequently arises when dealing with programs in CPS.) For now, having to mark `extEnv` for memoization is disappointing, but may not affect the end result if post-unfolding is performed. More critically, we want variable boundedness analysis to certify that all the variables are BV for the described BT division and corresponding annotated program.

```
goal p vs =
  ^0eval p (formEnv (parm_of (hd p)) vs) (body_of (hd p))

eval prg env e =
  case e of
    const          : const
    var            : (^1env var)
    op es          : apply op (^2map (eval prg env) es)
    if e0 e1 e2    : if (^3eval prg env e0)
                        (^4eval prg env e1)
                        (^5eval prg env e2)
    let x=e0 in e1 : ^6doLet prg env x (^7eval prg env e0) e1
    call f es      : ^8doCall prg f (^9map (eval prg env) es)

body, parm, map are exactly as in the last example.

formEnv names vals x =
  if (x = hd names) (hd vals) (^17formEnv (tl names) (tl vals) x)

extEnv env1 name1 val1 x1 =
  if (x1 = name1) val1 (^18env1 x1)

doLet prg3 env3 x3 v3 e3 =
  ^19eval prg3 (extEnv env3 x3 v3) e3

doCall prg4 f4 vs4 =
  ^20eval prg4 (formEnv (^21parm f4 prg4) vs4) (^22body f4 prg4)
```

Figure 8.5: Alternative handling of name-value bindings

Memo at 1 (target: `formEnv`)	$\mathtt{env} \overset{\not\uparrow}{\to}_a \mathtt{names}, \mathtt{e} \overset{\not\uparrow}{\to}_a \mathtt{x}$
	$\mathtt{p} \overset{\downarrow}{\to} \mathtt{names}, \mathtt{p} \overset{\triangleright}{\to} \mathtt{names}, \mathtt{e} \overset{\overline{\overline{\downarrow}}}{\to} \mathtt{x}, \mathtt{e} \overset{\triangleright}{\to} \mathtt{x}, \ldots$ *or*
	$\mathtt{prg4} \overset{\downarrow}{\to} \mathtt{names}, \mathtt{prg4} \overset{\triangleright}{\to} \mathtt{names}, \mathtt{e} \overset{\overline{\overline{\downarrow}}}{\to} \mathtt{x}, \mathtt{e} \overset{\triangleright}{\to} \mathtt{x}, \ldots$
Memo at 1 (target: `extEnv`)	$\mathtt{env} \overset{\not\uparrow}{\to}_a \mathtt{env1}, \mathtt{env} \overset{\not\uparrow}{\to}_a \mathtt{name1}, \mathtt{e} \overset{\not\uparrow}{\to}_a \mathtt{x1}$
	$\mathtt{e} \overset{\overline{\overline{\downarrow}}}{\to} \mathtt{x1}, \mathtt{e} \overset{\triangleright}{\to} \mathtt{x1}, \ldots$ (Note: `x3` and `env3` are dubious sources.)
Unfolding at 17	$\mathtt{names} \overset{\not\uparrow}{\to}_a \mathtt{names}, \mathtt{vals} \overset{\uparrow}{\to}_a \mathtt{vals}, \mathtt{x} \overset{\not\uparrow}{\to}_a \mathtt{x}$
	$\mathtt{names} \overset{\downarrow}{\to} \mathtt{names}, \mathtt{names} \overset{\triangleright}{\to} \mathtt{names}, \mathtt{x} \overset{\overline{\overline{\downarrow}}}{\to} \mathtt{x}, \mathtt{x} \overset{\triangleright}{\to} \mathtt{x} \ldots$
Memo at 18 (target: `formEnv`)	$\mathtt{env1} \overset{\not\uparrow}{\to}_a \mathtt{names}, \mathtt{x1} \overset{\not\uparrow}{\to}_a \mathtt{x}$
	$\mathtt{p} \overset{\downarrow}{\to} \mathtt{names}, \mathtt{p} \overset{\triangleright}{\to} \mathtt{names}, \mathtt{x1} \overset{\overline{\overline{\downarrow}}}{\to} \mathtt{x}, \mathtt{x1} \overset{\triangleright}{\to} \mathtt{x}, \ldots$ *or*
	$\mathtt{prg4} \overset{\downarrow}{\to} \mathtt{names}, \mathtt{prg4} \overset{\triangleright}{\to} \mathtt{names}, \mathtt{x1} \overset{\overline{\overline{\downarrow}}}{\to} \mathtt{x}, \mathtt{x1} \overset{\triangleright}{\to} \mathtt{x}, \ldots$
Memo at 18 (target: `extEnv`)	$\mathtt{env1} \overset{\not\uparrow}{\to}_a \mathtt{env1}, \mathtt{env1} \overset{\not\uparrow}{\to}_a \mathtt{name1}, \mathtt{x1} \overset{\not\uparrow}{\to}_a \mathtt{x1}$
	$\mathtt{x1} \overset{\overline{\overline{\downarrow}}}{\to} \mathtt{x1}, \mathtt{x1} \overset{\triangleright}{\to} \mathtt{x1}, \ldots$ (Note: `x3` and `env3` are dubious sources.)

Figure 8.6: Some abstract transitions for the example in Fig. 8.5

Applying the variable boundedness conditions. Classify input parameters p and vs as BV. Classify val1, v3, vs4 as BV because they are dynamic parameters of functions marked for memoization.

In the call graph, goal is in-neighbour to the component {eval,map,doCall,doLet}; eval is in-neighbour to the component {extEnv}; eval and extEnv are in-neighbours to the component {formEnv}; and doCall is in-neighbour to the components {body} and {parm}. As in Ex. 1, argue that each of prg, prg3 and prg4 is dominated by p at goal (this may require some reflection, but note that prg4 at doCall originates from prg at eval and the reverse is possible; prg3 at doLet originates from prg at eval and the reverse is possible; prg at eval may also originate from prg at eval or prg at map; and prg at map may originate from prg at map or prg at eval. Tracing backwards until goal, we find that prg at eval, prg3 at doLet and prg4 at doCall all originate from p at goal. This is of course the intuition behind the domination condition.) So classify prg, prg3 and prg4 as BV. Similarly, each of e, ls, x3, e3, f4 is dominated by p at goal, so they are BV. Now, prg1 and prg2 are dominated by prg4 at doCall, and therefore BV; f1 and f2 are dominated by f4 at doCall, and therefore BV; x1 is dominated by e at eval, and therefore BV; and x is dominated by e at eval and x1 at extEnv, and therefore BV.

The parameter env behaves like names in Ex. 1. After computing the dependency graph and removing the dependencies of known BV parameters, we determine a strongly-connected component comprising of env, clo and env3. The abstract transitions that annotate the arcs connecting these parameters in the dependency graph are those describing transitions to eval, map and doLet: They *exclude* the abstract transitions for callsites 8 and 20. Any strongly-connected subset of the included abstract transitions has a descending TP-arc of the form e $\stackrel{\downarrow}{\to}$ e, e $\stackrel{\downarrow}{\to}$ ls or e $\stackrel{\downarrow}{\to}$ e3. Further, all the in-neighbours to env, clo and env3 are BV (just note that the parameters of formEnv and extEnv are ineligible as in-neighbours, and all the other parameters are already known to be BV). By SCPB, env, clo and env3 are BV.

The remaining parameters are: names, vals, env1 and name1. Let us inspect the relevant abstract transitions, shown in Fig. 8.6. Note that for Ex. 1, which is essentially first-order, each callsite corresponds to one abstract transition. This is *not* the case for the present example. We have indicated the target function for each abstract transition in Fig. 8.6. However, it is possible for different abstract transitions to arise for the same callsite, having the same target function, but different SRGs. These abstract transitions describe closures originating from different locations in the subject program, and reflect the higher-order nature of SRG analysis. Note that the abstract transitions for callsites 1 and 18 have retained the size-relation arcs into names from p and prg4, but discarded size-relation arcs originating from env3 and x3 (action of ∇).

The companion set for env1 is {env1}, and no set of abstract transitions is constructive for it. Its in-neighbour env is already known to be BV. By SCPB, env1 is BV. The variable name1 is not in any cycle of the dependency graph, because name1 values are not propagated. Its in-neighbours env and env1 have been determined to be BV. By SCPB, name1 is BV. The companion set for names is {names}, and no set of abstract transitions is constructive for it. Its in-neighbours are env and env1, which are known to be BV. By SCPB, names is BV. Finally, the companion set for vals is {vals}. The only set of abstract transitions constructive for it consists of the single element describing the potential unfolding transitions at callsite 17. Fortunately, this set has the decreasing TP-arc names $\stackrel{\downarrow}{\to}$ names. The in-neighbours to vals are already known to be BV (because every other parameter has been classified BV). Therefore, by SCPB, vals is BV. Of course, since vals is dynamic, and all the static parameters have been checked, this final deduction is really unnecessary.

A curious observation: names may be classified BV by appealing to the domination principle. It is dominated by prg and prg4 at extEnv, and prg and prg4 at eval. Since prg is not even a parameter of extEnv, "prg at extEnv" really refers to the most recent value of prg in the call stack when control reaches extEnv. Such non-local deductions are made possible by the SRG-based modelling of closures.

Example 3: An interpreter for the lambda calculus. Figure 8.7 shows an interpreter for the lambda calculus with call-by-name semantics. It illustrates that generalization is necessary for certain examples.

If expr is dynamic, then all non-environment variables: expr, e, x1, e1, e2, x2, name3, val3, x3 are compelled to be dynamic. It certainly does not violate congruence for env, env1 and env3 to be classified as static: Indeed the values of these variables are statically determinable. However, there exist abstract

```
goal expr = (⁰eval empEnv expr)

eval env e  =
  case e of
    x           : (¹env x)
    (lam x1 e1) : contEval env x1 e1
    (e1 e2)     : (²(³eval env e1) e2)

contEval env1 x1 e1 e2 = (⁴eval (extEnv env1 x1 e2) e1)

empEnv x2 = nil

extEnv env3 name3 val3 x3 = if (x3 = name3) val3 (⁵env3 x3)
```

Figure 8.7: An interpreter for the lambda calculus

transitions a_2, a_4 for callsites 2 and 4 respectively (with respective target functions `contEval`, `eval`) such that `env` $\stackrel{\uparrow}{\rightarrow}_{a_2}$ `env1` and `env1` $\stackrel{\uparrow}{\rightarrow}_{a_4}$ `env`. Thus, $\{a_2, a_4\}$ is a constructive for `env`. There are no descending TP-arcs for this set of abstract transitions because there are no bounded static parameters! To ensure finite specialization, the parameter `env` should be generalized. This forces `env1` and `env3` to become dynamic, effectively stalling all specialization for the program.

In this case, the generalization has been necessary, since allowing `env` to be static would cause the specializer to attempt to create different versions of `eval` corresponding to `env` closures of ever increasing size, when the program is specialized with `expr` dynamic.

Example 4: An interpreter for ***L*****.** Figure 8.8 shows an interpreter for an *L*-like language. The function `getDef` replaces `parm` and `body` of Ex. 1 and 2. The interpreter has been written in CPS to achieve a good separation of binding-times. If `doApp` simply called:

```
(eval prg1 (formEnv (parm (hd lsThunk)) (tl lsThunk)) (body (hd lsThunk)))
```

to interpret a completed `lsThunk`, then `e` would be compelled to be dynamic by congruence, since `lsThunk`, which depends on the result of `eval`, is dynamic. In CPS, the argument that binds to `e`, which now appears in `getDef` (at callsite 14), is `(body_of (hd prg3))`, which is static. (We return to the issue of CPS programs after the example.)

An initial classification has the following static parameters: `p`, `prg`, `env`, `e`, `prg1`, `prg2`, `parm2`, `body2`, `prg3`, `cont3`, `prg4`, `clo`, `ls`, `names`, `x`, `env1`, `name1`, `x1`, `prg5`, `env5`, `x5`, `e5`, with the following functions marked for memoization: `eval`, `map`, `doApp`, `doLet`, `getDef`, `contEval` and `extEnv`. Variable boundedness analysis is applied to certify that every parameter is BV for the corresponding annotated program.

Applying the variable boundedness conditions. Classify input parameters `p` and `vs` as BV. Classify `lsThunk`, `argv2`, `f3`, `val1`, `val5` as BV since they are dynamic parameters of functions marked for memoization.

In the call graph, `goal` is in-neighbour to {`eval`, `map`, `doApp`, `doLet`, `getDef`, `contEval`}; `eval` is in-neighbour to {`extEnv`}; `eval` and `extEnv` are in-neighbours to {`formEnv`}; and `doApp` is in-neighbour to {`arity`}. As in the previous examples, argue that each of `prg`, `prg1`, `prg2`, `prg3` and `prg5` is dominated by `p` at `goal`. (Particularly interesting: `prg` at `eval` may be traced back to `prg2` at `contEval`, which is traced back to `prg1` at `getDef` (note: *not* `prg3` of `getDef`), which may be traced back to `prg1` at `doApp`, which is traced back to `prg` at `eval`. *Point*: Any such trace must find `p` at `goal` eventually.) So classify `prg`, `prg1`, `prg2`, `prg3` and `prg5` as BV. Similarly, each of `e`, `parm2`, `body2`, `ls`, `x5`, `e5` is dominated by `p` at `goal`, and therefore BV.

After computing the dependency graph and removing the dependencies of known BV parameters, we determine a strongly-connected component comprising of `env`, `clo` and `env5`. The abstract transitions

```
goal p vs =
  [0]eval p (formEnv (parm_of (hd p)) vs) (body_of (hd p))

eval prg env e =
  case e of
    var             : ([1]env var)
    fun             : (list fun)
    op es           : apply op ([2]map (eval prg env) es)
    if e0 e1 e2     : if ([3]eval prg env e0)
                         ([4]eval prg env e1)
                         ([5]eval prg env e2)
    let x=e0 in e1 : [6]doLet prg env x ([7]eval prg env e0) e1
    app e1 e2       : [8]doApp (append ([9]eval prg env e1)
                                     (list ([10]eval prg env e2)))
                              prg

doApp lsThunk prg1 =
  if (len lsThunk = ([11]arity (hd lsThunk) prg1) + 1)
     ([12]getDef (hd lsThunk) prg1 (contEval prg1 (tl lsThunk)))
     lsThunk

contEval prg2 argv2 parm2 body2 = ([13]eval prg2 (formEnv parm2 argv2) body2)

getDef f3 prg3 cont3 =
  if (f3 = (hd prg3)) ([14]cont3 (parm_of (hd prg3)) (body_of (hd prg3)))
                      ([15]getDef f3 (tl prg3) cont3)

arity f4 prg4 =
  if (f4 = (fun_of (hd prg4))) (len (parm_of (hd prg4))) ([16]arity f4 (tl prg4))

map clo ls =
  if (ls = nil) nil ([17]clo (hd ls) : [18]map clo (tl ls))

formEnv names vals x =
  if (x = hd names) (hd vals) ([19]formEnv (tl names) (tl vals) x)

extEnv env1 name1 val1 x1 =
  if (x1 = name1) val1 ([20]env1 x1)

doLet prg5 env5 x5 val5 e5 =
  [21]eval prg5 (extEnv env5 x5 val5) e5
```

Figure 8.8: An interpreter for an L-like language

```
append xs ys = (⁰append-k xs ys id)

append-k as bs k = if (as = nil) (¹k bs)
                      (²append-k (tl as) bs (cont k (hd as)))

cont c a r = (³c (a : r))

id x = x
```

Figure 8.9: The append program in CPS

that annotate the arcs connecting these parameters in the dependency graph describe transitions to `eval`, `map` and `doLet`. They *exclude* the abstract transitions to `doApp`, those from `doApp` to `getDef`; from `getDef` to `getDef` and `contEval`; and from `contEval` back to `eval` (namely the abstract transitions for callsites 8, 12, 15, 14 and 13). Any strongly-connected subset of the included abstract transitions has a descending TP-arc of the form $\mathtt{e} \xrightarrow{\downarrow} \mathtt{e}$, $\mathtt{e} \xrightarrow{\downarrow} \mathtt{ls}$ or $\mathtt{e} \xrightarrow{\downarrow} \mathtt{e5}$. Further, all the in-neighbours to `env`, `clo` and `env5` are already known to be BV. By SCPB, `env`, `clo` and `env5` are BV.

The remaining parameters are: `cont3`, `f4`, `prg4`, `names`, `vals`, `x`, `env1`, `name1` and `x1`. The companion set for `cont3` is {`cont3`}, and no set of abstract transitions is constructive for it. Its in-neighbours `lsThunk` and `prg1` are already known to be BV. By SCPB, `cont3` is BV. The companion set for `prg4` is {`prg4`}, and no set of abstract transitions is constructive for it. Its in-neighbour `prg1` is already known to be BV. By SCPB, `prg4` is BV. The companion set for `f4` is {`f4`}, and no set of abstract transitions is constructive for it. Its in-neighbour `lsThunk` is already known to be BV. By SCPB, `f4` is BV. Apply exactly the same reasoning as in Ex. 2 to determine that the parameters of `extEnv` and `formEnv` are BV.

Programs in CPS. A good separation of binding-times has been achieved for the above example by the use of a continuation variable [Bon92, FSDF93] (although see [DMP96]). Rather than look up the parameter list and body of a dynamic function name, and then call `eval` with the results, which would have rendered `e` dynamic, the potential call to `eval` is packaged as a continuation `contEval`, which is applied to the *static* parameter list and body of a matched function name. This approach allows `e` to keep its static binding-time.

In the example, `cont` is not accumulating: There is no recursive construction of continuation values. In general, automatic conversion into CPS may create accumulating continuation variables. For example, consider the simple `append` program, in continuation passing style, given in Fig. 8.9.

Let `xs` be dynamic and `ys` static. Initially, `k` is classified as static. There is one abstract transition for callsite 2. Call it a_2. Then, $\mathtt{k} \xrightarrow{\uparrow}_{a_2} \mathtt{k}$. The companion set for `k` is {`k`}. The set $\{a_2\}$ is constructive for `k` and has no decreasing TP-arc. (Since `as` is dynamic, `append·as` is ineligible as a TP-pair.) This forces `k` to be generalized, which is just as well, since otherwise the specializer would attempt to create different versions of `append-k` corresponding to ever increasing values of `k`, and diverge as a result. In Jones, Gomard and Sestoft's textbook [JGS93], blind conversion into CPS as a means of achieving better binding-time separation is not recommended. One of the reasons cited is the possibility of introducing accumulating parameters (such as `k`). Although variable boundedness analysis guards against non-termination, the advice to use CPS sparingly stands: CPS programs are more higher-order and therefore more likely to confuse static analysis, leading to over-generalization and under-specialization.

Now consider the `append` example with `xs` static and `ys` dynamic. Initially, `k`, `as`, `c` and `a` are static, with `cont` marked for memoization. The input parameters `xs` and `ys` are BV. The parameter `r` is BV because it is a dynamic parameter of a function marked for memoization. The parameter `as` is dominated by `xs` at `append`, and therefore BV. The companion set for `bs` is {`bs`}. No set of abstract transitions is constructive for it, and its in-neighbour `ys` is already known to be BV. By SCPB, `bs` is BV. The companion set for `k` is {`k`}. The only set of abstract transitions constructive for `k` consists of the single abstract transition for callsite 2. This set has the decreasing TP-arc $\mathtt{as} \xrightarrow{\downarrow} \mathtt{as}$. Only `as` is in-neighbour to `k`, and `as` is already known to be BV. By SCPB, `k` is BV. The companion set for `c` is {`c`}, and no set of abstract transitions is constructive for it. Its in-neighbour `k` is already known to be BV, so by SCPB, `c` is BV. The variable `a` is not in any cycle of the dependency graph, and its in-neighbours `k` and `c` are already known to be BV. By SCPB, `a` is BV. The variable `x` is not in any cycle of the dependency graph, and its in-neighbours are already known to be BV. By SCPB, `x` is BV.

As pointed out in Ex. 2, there is something unsatisfactory about having to mark for memoization functions like `cont`. It highlights the inadequacy of termination analysis to distinguish between non-recursive continuations and true recursion (by named-reference or the Y-combinator). The effect may not be significant if post-processing is performed. However, pure termination analysis would conservatively declare possible non-termination for CPS append. In the next chapter, we indicate a simple extension to SRG analysis to identify size decreases involving closure values, as observed in transitions at the expression `(c (a : r))` of the CPS append example, and `(env1 x1)` of Ex. 2 and 4.

Chapter 9

Conclusion

9.1 Future work

In this work, we have formulated sufficient conditions for program termination and variable boundedness in terms of the size relations and dependencies among arguments at potential function-call transitions. These conditions are realizations of general *principles* centered around impossible size descent that would arise in some argument values, should termination/ boundedness be violated. The conditions are judged on how well they capture this size descent, and on their efficiency.

The following are some common ways to reason about the "descent" observed in an execution of the subject program.

1. Characterize the descent observed in *every* function-call transition of a self-recursion. For example, a certain *tuple* of argument sizes is descending lexically in every such transition, or a certain *sum* of argument sizes is decreasing in every such transition.

2. Consider any descent in simple indirect recursion.

3. Incorporate information about the results of function calls.

The most precise methods in this work subsume the above reasoning, and can detect very complex descent besides. Through our investigations, we have found that these methods, as well as many methods described in the literature for program termination and variable boundedness, are *intrinsically* hard. Constructively, we propose efficient polytime approaches that are demonstrated to be sufficiently precise (by considering the forms of descent that are handled).

Prototype supporting analyses are sketched. These collect useful size and dependency information for program expressions. They allow us to describe more accurately the argument relations at potential function-call transitions. Their precision is critical to the success of deciding program termination or ensuring variable boundedness in practice. The given analyses form a good basis for the actual realization of the polytime methods described in this work. However, a number of issues (both theoretical and practical) have to be tackled before there can be implementations suitable for "black-box" application.

9.1.1 Experimentation

Experiments need to be conducted to test the applicability of the SCP termination criterion in practice. We expect to find most of the test suites for program termination [LS97a, CT97, SSS97] amenable to the SCP approach.

Experiments should be designed to test the applicability of SCPB for the general application of offline partial evaluation, rather than the specialization of programs manipulated to specialize well (e.g., the interpreters). However, note that even in the context of specializing interpreters, where it would be natural to manually ensure variable boundedness, our analysis serves to *confirm* the safety of any specialization. Other uses of variable boundedness analysis, e.g., to check for finite resource consumption by a non-terminating process, could be motivation for further experimentation.

9.1.2 Framework for applying analyses

Variable boundedness analysis. The success of variable boundedness analysis is measured by the generalization that is *not* caused. In other words, unnecessary generalization should be minimized. In practice, offline partial evaluation is used as a semi-automatic tool. A certain amount of manual re-coding is extremely helpful towards its success.

In this context, the user may be encouraged to specify manually which expressions or variables to generalize in order to achieve boundedness for the program variables. In the same vein, variable boundedness information can be provided manually, in case automatic analysis causes generalization that is known to be unnecessary. As long as the analyzer is sufficiently fast and precise, the feedback that it provides about any generalization performed due to variable boundedness considerations can be inspected by the user, who can then modify the subject program, or provide information that allows better specialization.

Program termination analysis. Program termination analysis is frequently part of the program verification process. Typically, deductive methods are used to show partial correctness, i.e., the input-output specification is satisfied *when* the program terminates. This is followed by a separate proof of program termination [Hoa69]. The purpose of automatic termination analysis is to assist in the termination proof by automatically classifying as much of the subject program as possible.

A framework is needed in which the user can run the SCP termination algorithm, and obtain a report as to the "trouble areas" of the program where non-termination may occur (the SCP termination algorithm is easily adapted for this purpose). In this context, the algorithm is a program verification *tool*. For functions that cannot be handled by SCP, other automatic methods can be tried. Occasionally, termination is dependent on information that is not easily available to static analysis. In such circumstances, a manual proof of termination may be the only way to proceed. The design of a framework in which the user can carry out termination testing conveniently requires care. At any rate, as an aid to program testing, termination analysis must be more "verbose" than simply declare success or failure for the subject program.

9.1.3 Towards implementation

Efficient abstract interpretation. The proposed SRG analysis can generate exponentially large descriptions of possible function-call transitions and their argument size relations. In practice, it may be sensible to merge all the abstract closures with the same target function and the same number of arguments, for any single deduction. This would achieve polynomial complexity, while still capturing useful size information for closure arguments.

Polyvariance of abstract interpretation. Even as we contemplate ways to approximate the abstract function-call transitions, it is clear that the common functionals, such as `map` and `fold`, require special considerations. Such functionals are convenient for programming, but they easily confuse monovariant static analyses. This argues for a degree of polyvariance; it may be practical to single out the most frequently used library functionals (like `map`) for polyvariant treatment.

Issues related to the form of the SRG. An explicit representation of the SRG contains redundant information. For any function-call transition from f to f', only the parameters of f' are updated. Thus, only the argument size relations with target parameters in $\mathit{Param}(f')$ need to be represented. The "bridging arcs" of the form $x \overset{\overline{\downarrow}}{\to} x$, for $x \notin \mathit{Param}(f')$, do not have to recorded explicitly. We have assumed for the SCP analysis of an SCC S of Tr that the SRGs are represented "explicitly." Although it has been mentioned that the parameters of functions not referred to by S elements can be ignored, it makes sense to avoid handling the redundant arcs as far as possible.

For instance, the analysis of first-order parts of the subject program does not require the redundant arcs at all. These parts of the program could be treated differently from the higher-order parts, using efficient procedures that ignore the redundant arcs, as well as TP-parameters $f \cdot y$ where $y \notin \mathit{Param}(f)$. The procedures for treating the higher-order parts of the subject program should also take advantage of the form of the SRG, where possible. For example, the following are obvious improvements.

1. In the procedure for determining whether a given variable y is dominated by BV variables at the in-neighbour functions, depth-first search is used to traverse a tree with vertices of the form $h \cdot z$ where $h \in \mathit{Fun}$ and $z \in \mathit{Var}$. The tree is rooted at $f \cdot y$. Recall that the arcs of a successfully traversed tree indicate how (whether) a complete $\overset{\triangleright}{\to}$-thread reaching y in a multipath ending at f may be extended back, if the initial function of the multipath is in the same call-graph SCC as f, and an abstract transition is prepended to the multipath.

   Suppose that a vertex of the form $h' \cdot z$ is encountered, where h' is in the same call-graph SCC as f, and let $z \in \mathit{Param}(h)$. If h and h' are different functions in the same call-graph SCC, it is only necessary to ask how (whether) $h \cdot z$ may be extended back, since a transition to h is needed to overwrite the value of z. It is valid to regard $h \cdot z$ as the sole descendant of $h' \cdot z$ in the traversal.

If h and h' are not even in the same call-graph SCC, then it is valid to regard $h' \cdot z$ as a leaf node in the traversal.

2. For the calculation of TP-parameters, if $z \in Param(h)$ and $h \cdot z$ is eliminated, then all elements of the form $h' \cdot z$ may be eliminated at once. Argue as follows. Suppose that $z \in Param(h)$ and $h \cdot z$ is eliminated, *but* $h' \cdot z$ is TP, where h' and h are strongly-connected in the call graph. Now consider a multipath from h to h' without any cycles. By definition, this multipath has a thread of "bridging arcs" from z to z. As this is the only thread that extends $h' \cdot z$ leftwards, and all TP-parameters are extendible leftwards to other TP-parameters, $h \cdot z$ must be TP, which is a contradiction.

9.1.4 Special program classes

Consider the `append` program in CPS, seen at the end of the last chapter.

```
append xs ys = (⁰append-k xs ys id)

append-k as bs k = if (as = nil) (¹k bs)
                      (²append-k (tl as) bs (cont k (hd as)))

cont c a r = (³c (a : r))

id x = x
```

To handle the termination of this program (or allow unfolding transitions to `cont` when specializing it with `xs` static), the analysis must recognize that any function-call transition at callsite 3 is such that the value of the continuation variable `c` after the transition is a smaller closure than the current value of `c` (a suitable size function here is the total number of function *and* list constructors in a value). Thus the SRG for the abstract function-call transition at 3 may be given the arc $\mathtt{c} \stackrel{\downarrow}{\rightarrow} \mathtt{c}$. If the target parameter (`c` in this case) can take a closure value, then the size analysis $\mathcal{S}$ does not deduce any size relation for it anyway, so the addition of the arc from `c` to `c` does not introduce any unwanted fan-ins. In general, $\mathcal{G}[\![x]\!]$ may be modified as follows: If $\langle\!\langle f, n, G\rangle\!\rangle \in \mathcal{G}[\![x]\!]$, then include in G the arc $x \stackrel{\downarrow}{\rightarrow} f^{(i)}$, for $i \leq n$ and $\mathcal{C}l[\![f^{(i)}]\!] \neq \{\}$.

We have omitted this simple modification in the main body of the exposition for the following reason. We have assumed from the start that higher-order facilities are used in restricted ways. Accordingly, size analysis has not been concerned with descent in closure sizes. The CPS programs form a special class of interest. Their treatment is aptly regarded as an extension of the main approach.

To continue with the CPS-append example, let the abstract transitions for callsites 2 and 3 be a_2 and a_3 respectively. The possible strongly-connected subsets are $\{a_2\}$ and $\{a_3\}$. For $\{a_2\}$, the SRG of a_2 has the TP-arc $\mathtt{as} \stackrel{\downarrow}{\rightarrow} \mathtt{as}$, and for $\{a_3\}$, the SRG of a_3 (now) has the TP-arc $\mathtt{c} \stackrel{\downarrow}{\rightarrow} \mathtt{c}$. By the SCP termination criterion, the program is terminating.

We considered the specialization of the CPS-append program with `xs` static and `ys` dynamic at the end of the last chapter. It was observed that `r` was BV because `cont` was marked for memoization. The new termination analysis does not require `cont` to be marked for memoization, so a different argument is needed. Now, the companion set of `r` is $\{\mathtt{r}\}$. The only set of abstract transitions constructive for it consists of the abstract transition for callsite 3. This set has the decreasing TP-arc $\mathtt{c} \stackrel{\downarrow}{\rightarrow} \mathtt{c}$. The in-neighbours to `r` are `a` and `bs`, which are already known to be BV. By SCPB, `r` is BV. The success of the SCPB criterion here should come as no surprise, since it is due to the same descent that has allowed unfolding transitions to `cont` in the first place. This is the same descent that allows the SCP termination criterion to deduce that any function-call transition to `cont` must terminate.

We leave open the question of what other specific programming styles deserve special considerations, or can be accommodated easily.

9.2 Contributions of this work

In this work, we have studied the program termination and variable boundedness problems from a mainly theoretical perspective, placing particular emphasis on the formulation and study of conditions for judging program termination and variable boundedness. For the effort, we have been rewarded with formal PSPACE-hardness results for many approaches to these problems in the literature [Sag91, Hol91, GJ96, AH96, LSS97, CT97, Gle99, LJBA01]. This indicates to us the need to consider approximate criteria, not just ad hoc approximations of existing methods.

In this work, we have proposed polytime criteria for both program termination and variable boundedness, which are demonstrated to be sufficiently precise by considering the type of descent subsumed by them. Our criteria are based on general principles, centered around impossible size descent that would arise in some argument values, should termination/ boundedness be violated. Our conditions deduce this size descent from the size relations among argument values at potential function-call transitions. From our study, we have achieved a better understanding of the limitations and strengths of this form of reasoning.

The connection between the program termination and variable boundedness problems, mentioned in the introductory chapter, is evidenced by a clear analogy among the program termination and variable boundedness criteria. The ISD termination criterion has been inspired by the variable boundedness analyses in the literature, while SCPB is an adaptation of the SCP termination criterion to variable boundedness. In this way, we have capitalized on the connection between the two problems.

Of course, we do not deny that many orthogonal concerns have to be addressed for the success of program termination or variable boundedness analysis in practice. For this reason, we have sketched a suite of prototype supporting analyses, sufficiently precise to allow straightforward higher-order programs to be handled. These analyses should be considered as preparation for the implementation and experimentation phase of this research. We now briefly review what has been achieved in the present study.

1. An intuitive abstract representation is proposed for recording the argument size relations at potential function-call transitions (in terms of which conditions for program termination and variable boundedness can be formulated). The SRG captures the relevant information for reasoning about size descent.

2. The SRG representation directly determines what size-descent deductions are possible. In this work, we have strived to balance precision and efficiency.

   For precision, the semantics of the subject language and the SRG analysis have been carefully designed to make it possible to express size relations with respect to the latest value on the call stack for *any* variable x. In the (instrumented) L semantics, such size relations at a function-call transition are considered "local" to the transition. It is hoped that this is sufficient to handle any straightforward use of higher-order facilities.

   Otherwise, the SRG is an intuitive representation of a set of size relations. It gives an intuitive modelling of the size relations among a closure's arguments and the variable values in environments where the closure may occur. Abstract closures are "single-level" in the sense that embedded closures are not modelled directly. Any closures embedded at the i-th argument position of all f closures are modelled by elements tagged to $f^{(i)}$. This is a form of context-insensitivity.

3. A suite of prototype supporting analyses are sketched, ranging from size analysis to analysis of abstract closures carrying size and dependency information about the modelled closures' arguments. This is in preparation for a practical implementation.

4. Different termination criteria are stated in terms of the abstract function-call transitions collected for the subject program. These criteria include one that has been implemented [LSS97, CT97], and two approximations. The type of descent reasoning subsumed by each criterion is explored with the help of examples. The investigation provides some insight into the capability of each criterion.

5. Formal PSPACE-hardness results are proved for the SCT and ISD termination criteria. The complexity results extend to analyses described in the literature [Sag91, LSS97, CT97], and point to the need for good approximate criteria.

6. A PTIME criterion is proposed that handles "permuted argument" descent and lexical descent. We believe that the criterion is sufficiently precise in practice, and that it is suitable for efficient implementation. Its decision method has a practical time complexity, and uses only simple graph techniques.

7. Interesting applications of termination analysis are discussed, in the context of program transformation. Particularly relevant is the application of termination analysis to ensure finite unfolding during specialization. It has been pointed out that finite unfolding is directly related to program termination, rather than variable boundedness.

8. Variable boundedness criteria are derived by analogy with the termination criteria, establishing the close relationship between the problems of program termination and variable boundedness.

   One of our criteria corresponds to an approach to variable boundedness analysis studied over the past decade [Hol91, JGS93, GJ96, AH96, Gle99]. We call this criterion ISDB. A comparison of ISDB and the ISD termination criterion has motivated us to define SCTB and SCPB, variable boundedness criteria analogous to the SCT and SCP termination criteria.

9. The PSPACE-hardness results for the SCT and ISD termination criteria are extended to SCTB and ISDB. Thus, all the analyses of variable boundedness from [Hol91] to [Gle99] are PSPACE-hard. This points to the need for polytime conditions that can decide the boundedness of a variable.

10. In addition to SCPB, another PTIME condition, called the domination condition, is proposed for deciding the boundedness of a variable. The condition implies that values of the variable are "dominated" by values of other bounded variables at in-neighbour functions. The condition is based on size-relation information only.

    We have demonstrated an approach based on the 2 conditions above on programs that make straightforward use of higher-order programming (even programs in CPS), and believe such an approach to be sufficiently precise in practice. We also believe that the conditions are suitable for efficient implementation. Their decision methods have practical time complexities, and use only simple graph techniques.

In summary, the basis has been laid for

- an analysis tool to help the user probe for program non-termination;
- finite offline partial evaluation, which performs *finite* and *effective* program specialization.

Some adaptation of the techniques discussed here may be suitable for incorporation in an offline partial evaluator such as Similix. As has been pointed out, there remains work to be done to achieve this goal, but we hope to have taken a step in the general direction.

Appendix A

Automata

A.1 Finite state automata, and deciding ISD

Definition A.1.1 For any finite set A, define A^* to be the set of all finite sequences over A. A subset of A^* is called a *language over* A.

Definition A.1.2 A *finite state automaton* (FSA) is a tuple $\mathcal{A} = (\Sigma, S, S_0, \rho, F)$, where Σ is a finite set called the *alphabet*, S is a finite set called *states*, S_0 is a subset of S called the *initial states*, and F is a subset of S called the *final states*. The *state transition relation* $\rho \subseteq S \times \Sigma \times S$ is a set of *transition triples*.

Definition A.1.3 (Behaviour of an FSA) Let $\mathcal{A} = (\Sigma, S, S_0, \rho, F)$ be any FSA.

1. A *run* of $\mathcal{A}$ on a finite word, $a_1 \ldots a_T \in \Sigma^*$, is a sequence $r = s_0 a_1 s_1 \ldots a_T s_T$, such that $s_0 \in S_0$, and for $0 \leq t < T$, $(s_t, a_{t+1}, s_{t+1}) \in \rho$.
2. The run r is *accepting* if $s_T \in F$.
3. $L(\mathcal{A}) = \{w \in \Sigma^* \mid \text{ some run } r \text{ on } w \text{ is accepting}\}$ is the *language accepted by* $\mathcal{A}$.

Definition A.1.4 A language A over Σ is called *regular* if $A = L(\mathcal{A})$ for some FSA $\mathcal{A}$.

Lemma A.1.5 Recall the definitions of *LOOP* and *DOWN* from Ch. 4. *Tr* are abstract function-call transitions of the subject program p. Each element of *Tr* is labelled distinctly by $a \in \Sigma$, where Σ is finite. The abstract transition labelled by a is denoted $tr(a)$. Then,

$$\begin{array}{lcl} \mathit{LOOP} & = & \{a_1, \ldots, a_T \mid tr(a_1), \ldots, tr(a_T) \text{ is a } \mathit{Tr}\text{-loop}\} \\ \mathit{DOWN} & = & \{a_1, \ldots, a_T \mid tr(a_1), \ldots, tr(a_T) \\ & & \quad \text{has a complete thread from } x \text{ to } x \text{ with a } \downarrow\text{-arc}\} \end{array}$$

Claim: *LOOP* and *DOWN* are regular subsets of Σ^*.

Proof Define $\rho = \{((f', b), a, (f'', 1)) \mid b \in \{0,1\},\ tr(a) = f' \xrightarrow{G} f''\}$. Here, b is a boolean flag (to indicate that the multipath is non-empty). For FSA $\mathcal{A}_f = (\Sigma, \mathit{Fun} \times \{0,1\}, \{(f,0)\}, \rho, \{(f,1)\})$, a run r is accepting iff $r = (f_0, 0) a_1 (f_1, 1) \ldots a_T (f_T, 1)$ where $f_0 = f_T = f$ and $tr(a_1) \ldots tr(a_T)$ is a *Tr*-loop (around f). Thus, the words corresponding to *Tr*-loops around f is a regular subset of Σ^*. Now, *LOOP* is the union of $L(\mathcal{A}_f)$ for $f \in \mathit{Fun}$. By standard a technique, *LOOP* can be shown to be a regular subset of Σ^*. Explicitly, for each $\mathcal{A}_f$, we create copies of the states $S = \mathit{Fun} \times \{0,1\}$, indexed by f. Define:

$$\begin{array}{lcl} \rho' & = & \{((f', b, f), a, (f'', 1, f)) \mid tr(a) = f' \xrightarrow{G} f'',\ b \in \{0,1\},\ f \in \mathit{Fun}\} \\ S' & = & \mathit{Fun} \times \{0,1\} \times \mathit{Fun} \\ S'_0 & = & \{(f, 0, f) \mid f \in \mathit{Fun}\} \\ F' & = & \{(f, 1, f) \mid f \in \mathit{Fun}\} \end{array}$$

Let $\mathcal{A}' = (\Sigma, S', S_0', \rho', F')$. Then $\mathcal{A}'$ accepts any word corresponding to a call-graph loop, i.e., it accepts *LOOP*. Intuitively, an $\mathcal{A}'$-run tracks the functions in a multipath, while "remembering" the initial function.

For an automaton to accept *DOWN*, a run needs to track the parameters and size relations of a thread, so the states have to record the initial and current parameters, as well as any $\downarrow$ size relation observed. Define:

$$\begin{array}{rcl}
\rho'' & = & \{((x', r', x), a, (x'', r'', x)) \mid x \in \mathit{Var},\ r', r'' \in \{\downarrow, \overset{=}{\downarrow}\}, \\
 & & \quad tr(a) = f' \overset{G}{\to} f'', x' \overset{r}{\to} x'' \in G,\ (r = \downarrow \text{ or } r' = \downarrow) \Rightarrow (r'' = \downarrow)\} \\
S'' & = & \mathit{Var} \times \{\downarrow, \overset{=}{\downarrow}\} \times \mathit{Var} \\
S_0'' & = & \{(x, \overset{=}{\downarrow}, x) \mid x \in \mathit{Var}\} \\
F'' & = & \{(x, \downarrow, x) \mid x \in \mathit{Var}\}
\end{array}$$

Let $\mathcal{A}'' = (\Sigma, S'', S_0'', \rho'', F'')$. It is not difficult to prove that for $T > 0$, there is an $\mathcal{A}''$-run on $a_1 \ldots a_T$ ending in state (x_T, r, x_0) iff $tr(a_1) \ldots tr(a_T)$ has a thread from x_0 to x_T. Further, there is an $\mathcal{A}''$-run on $a_1 \ldots a_T$ ending in state $(x_T, \downarrow, x_0)$ iff $T > 0$ and $tr(a_1) \ldots tr(a_T)$ has a thread from x_0 to x_T containing at least one $\downarrow$-arc. (Note that $tr(a_1) \ldots ar(a_T)$ may not be flow-legal.) It follows that there is an accepting $\mathcal{A}''$-run on $a_1 \ldots a_T$ iff for some x, $tr(a_1) \ldots tr(a_T)$ has a thread from x to x with a $\downarrow$ arc. By definition, the language accepted by $\mathcal{A}''$ is *DOWN*.

A.2 Infinite automata, and deciding SCT

An infinite automaton is like a finite state automaton, except that it accepts a language of infinite words over a finite alphabet.

Definition A.2.1 For any finite set A, define A^ω to be the set of all *infinite* sequences over A. A subset of A^ω is called an *infinitary language over* A.

Definition A.2.2 A *Büchi automaton* is a tuple $\mathcal{A} = (\Sigma, S, S_0, \rho, F)$, where Σ is a finite set called the *alphabet*, S is a finite set called *states*, S_0 is a subset of S called the *initial states*, and F is a subset of S called the *accepting states*. The *state transition relation* $\rho \subseteq S \times \Sigma \times S$ is a set of *transition triples*.

Definition A.2.3 (Behaviour of a Büchi automaton) Let $\mathcal{A} = (\Sigma, S, S_0, \rho, F)$ be a Büchi automaton.

1. A *run* of $\mathcal{A}$ on a infinite word, $a_1 a_2 \ldots \in \Sigma^\omega$, is a sequence $r = s_0 a_1 s_1 \ldots$, such that $s_0 \in S_0$, and for $t \geq 0$, $(s_t, a_{t+1}, s_{t+1}) \in \rho$.

2. The run r is *accepting* if for infinitely many t, $s_t \in F$.

3. $L_\omega(\mathcal{A}) = \{w \in \Sigma^\omega \mid \text{ some run } r \text{ on } w \text{ is accepting}\}$ is the *infinitary language accepted by* $\mathcal{A}$.

Definition A.2.4 An infinitary language A over Σ is called ω*-regular* if $A = L_\omega(\mathcal{A})$ for some Büchi automaton $\mathcal{A}$.

Lemma A.2.5 Recall the definitions of *FLOW* and *DESC* from Ch. 4. *Tr* are abstract function-call transitions of the subject program p. Each element of *Tr* is labelled distinctly by $a \in \Sigma$, where Σ is finite. The abstract transition labelled by a is denoted $tr(a)$. Then,

$$\begin{array}{rcl}
\mathit{FLOW} & = & \{a_1 a_2 \ldots \mid tr(a_1), tr(a_2), \ldots \text{ is an infinite } \mathit{Tr}\text{-multipath}\} \\
\mathit{DESC} & = & \{a_1 a_2 \ldots \mid \exists t : tr(a_t), tr(a_{t+1}), \ldots \text{ has a complete thread with infinite descent}\}
\end{array}$$

Claim: *FLOW* and *DESC* are ω-regular subsets of Σ^ω.

Proof Define $\mathcal{A}' = (\Sigma, Fun, Fun, \rho', Fun)$, where $\rho' = \{(f, a, f') \mid tr(a) = f \overset{G}{\to} f'\}$. Then, $\mathcal{A}$ accepts words corresponding to (flow-legal) infinite Tr-multipaths. By definition, $FLOW = L_\omega(\mathcal{A}')$ is an ω-regular subset of Σ^ω.

Next, we define a Büchi automaton to accept any word corresponding to a sequence of SRGs (not necessarily a multipath) with a complete thread of infinite descent. In order to track size relations, the states are pairs (x, r) where x is a parameter, and $r \in \{\downarrow, \overset{=}{\downarrow}\}$. Define

$$
\begin{array}{lcl}
\rho'' & = & \{((x,r), a, (x', r')) \mid r \in \{\downarrow, \overset{=}{\downarrow}\},\ tr(a) = f \overset{G}{\to} f',\ x \overset{r'}{\to} x' \in G\} \\
S'' & = & Var \times \{\downarrow, \overset{=}{\downarrow}\} \\
S''_0 & = & \{(x, \overset{=}{\downarrow}) \mid x \in Var\} \\
F'' & = & \{(x, \downarrow) \mid x \in Var\}
\end{array}
$$

Let $\mathcal{A}'' = (\Sigma'', S'', S''_0, \rho'', F'')$. Then, for any infinite word $w = a_1 a_2 \ldots$, there exists an accepting run of $\mathcal{A}''$ on w iff $tr(a_1) tr(a_2) \ldots$ contains a complete thread with infinite descent. Clearly, we have $DESC = \Sigma^* \cdot L_\omega(\mathcal{A}'')$. It follows that $DESC$ is an ω-regular subset of Σ^ω.

Appendix B

Correctness of specialization of *2L* programs

B.1 Lengthy proofs

This appendix collects together lengthy inductive proofs for the correctness the specialization of *2L* programs. For succinctness, we will often write $\sigma[f^{(i)} \mapsto u_i]$ and $\tau[f^{(i)} \mapsto \xi_i]$ when the range of i is clear from context; usually, $i = 1, \ldots, n+1$ or $i = 1, \ldots, ar(f)$. The expression "$\sigma'; \zeta \Downarrow v'$ agreeing with v" means "$\sigma'; \zeta \Downarrow v'$ where v' agrees with v."

Theorem B.1.1 (Preserving the results of successful computations) Let p' be a specialization of tp, a well-annotated version of p. Let te be a subexpression of te^g, and $\phi(te) = e$. Suppose that (τ, σ') corresponds to σ on g, and $\sigma; e \Downarrow v \neq \texttt{Err}$. If $\tau; te \downarrow \zeta$, then (ζ, σ') corresponds to v.

Proof Suppose that (τ, σ') corresponds to σ on g, and we have $\sigma; e \Downarrow v \neq \texttt{Err}$ and $\tau; te \downarrow \zeta$. We must prove that (ζ, σ') corresponds to v. The proof is by induction on the size of the evaluation tree for $\sigma; e$. We proceed by case analysis of the form of te. In each case, we may assume, by the induction hypothesis, that each subcomputation state of $\sigma; e$ satisfies the theorem. No subcomputation state aborts or diverges, otherwise there would not be any v such that $\sigma; e \Downarrow v$. So if a subcomputation state has environment σ and the annotated version of its expression is specialized with τ, then supposing that the specialization yields $\zeta 1$, we determine immediately that $(\zeta 1, \sigma')$ corresponds to the result of the subcomputation.

A note on `lift`*s*: Let $te \equiv (\texttt{lift}\ te1)$. Put $e1 = \phi(te1)$. Suppose that we can show (by induction) that for $\sigma; e1 \Downarrow v1 \neq \texttt{Err}$ and $\tau; te1 \downarrow \zeta 1$, we have that $(\zeta 1, \sigma')$ corresponds to $v1$. By well-annotatedness and Lemma 6.6.3, $v1 \in \mathit{Baseval}$. So $\zeta 1 = v1$. This means that $\tau; te \downarrow \zeta \in \mathit{Expr}$ such that ζ always evaluates to $v1$. It follows that ζ matches $v1$ (trivially), and for the unmatched pair $(\zeta, v1)$, we have $\sigma'; \zeta \Downarrow v'$ agreeing with $v1$ (as $v' = v1$). By definition, (ζ, σ') corresponds to $v1$. This reasoning allows us to ignore `lift` operations in the following case analysis. The same reasoning should be applied to $(\texttt{lift}\ te1)$ for each possible $te1$ from below.

- $te \equiv x$: $\zeta = \tau(x)$, $v = \sigma(x)$. Since (τ, σ') corresponds to σ on g and $x \in \mathit{Param}(g)$, $(\tau(x), \sigma')$ corresponds to $\sigma(x)$, i.e., (ζ, σ') corresponds to v, as required.

- $te \equiv f^{\mathtt{S}}$: $\zeta = v = \langle f \rangle$, so (ζ, σ') corresponds to v.

- $te \equiv f^{\mathtt{M}}$: Let $\sigma'; \zeta \Downarrow v'$. Then $v' = \langle f' \rangle$ where $f' = \mathit{nextFun}(f, \tau)$. Now, $v = \langle f \rangle$, at the same level of instantiation as v' (both require $ar(f)$ arguments to complete). By definition, v' and v agree, so

(ζ, σ') corresponds to v, as required. *Note*: For $\zeta \in \mathit{Expr}$, (ζ, σ') corresponds to v iff $\sigma'; \zeta$ evaluates to some v' agreeing with v.

- $te \equiv (\mathtt{if}\ te1\ te2\ te3)^{\mathsf{S}}$: Let $\tau; te1 \downarrow \zeta1$, $\tau; te2 \downarrow \zeta2$ and $\tau; te3 \downarrow \zeta3$. Put $e1 = \phi(te1)$, $e2 = \phi(te2)$ and $e3 = \phi(te3)$, and let $\sigma; e1 \Downarrow v1$, $\sigma; e2 \Downarrow v2$ and $\sigma; e3 \Downarrow v3$ (if they exist). By the induction hypothesis, $\zeta1$ matches $v1$. By well-annotatedness and Lemma 6.6.3, $\zeta1 \in \mathit{Baseval}$. Therefore $\zeta1 = v1$. We deduce that either $\zeta = \zeta2$ and $v = v2$, or $\zeta = \zeta3$ and $v = v3$. In each case, it follows from the induction hypothesis that (ζ, σ') corresponds to v, as required.

- $te \equiv (\mathtt{if}\ te1\ te2\ te3)^{\mathsf{D}}$: Let $\tau; te1 \downarrow \zeta1$, $\tau; te2 \downarrow \zeta2$ and $\tau; te3 \downarrow \zeta3$. Put $e1 = \phi(te1)$, $e2 = \phi(te2)$ and $e3 = \phi(te3)$, and let $\sigma; e1 \Downarrow v1$, $\sigma; e2 \Downarrow v2$ and $\sigma; e3 \Downarrow v3$ (if they exist). Further, let $\sigma'; \zeta1 \Downarrow v1'$, $\sigma'; \zeta2 \Downarrow v2'$ and $\sigma'; \zeta3 \Downarrow v3'$ (if they exist). Now, $\zeta = \mathit{makeIf}(\zeta1, \zeta2, \zeta3)$. By the induction hypothesis, if $v1 \notin \{\mathtt{Err}, \mathtt{False}\}$, then $v1'$ agrees with $v1$, in particular, $v1' \notin \{\mathtt{False}, \mathtt{Err}\}$. Further, $v = v2$, and it follows from the induction hypothesis that $v2'$ and $v2$ agree. Since $v1' \notin \{\mathtt{Err}, \mathtt{False}\}$, we deduce that $v' = v2'$, so v' agrees with v. Therefore (ζ, σ') corresponds to v, as required. The case for $v1 = \mathtt{False}$ is similar.

- $te \equiv (op\ te1\ \ldots)^{\mathsf{S}}$: The following reasoning for $te1$ applies to every argument of the operation. Let $\tau; te1 \downarrow \zeta1$ and $\sigma; e1 \Downarrow v1$, where $e1 = \phi(te1)$. Since $v \neq \mathtt{Err}$, we have that $v1 \neq \mathtt{Err}$. It follows from the induction hypothesis that $\zeta1$ matches $v1$. By well annotatedness and Lemma 6.6.3, $\zeta1 \in \mathit{Baseval}$, so $\zeta1 = v1$. We deduce that $\zeta = \mathcal{O}[\![op]\!](\zeta1, \ldots) = \mathcal{O}[\![op]\!](v1, \ldots) = v$. Therefore (ζ, σ') corresponds to v.

- $te \equiv (op\ te1\ \ldots)^{\mathsf{D}}$: The following reasoning for $te1$ applies to every argument of the operation. Let $\tau; te1 \downarrow \zeta1$, $\sigma; e1 \Downarrow v1$ where $e1 = \phi(te1)$, and $\sigma'; \zeta1 \Downarrow v1'$. Since $v \neq \mathtt{Err}$, we must have $v1 \in \mathit{Baseval}$ and $v1 \neq \mathtt{Err}$. It follows from the induction hypothesis that $v1' = v1$. Now, since $\zeta = \mathit{makeOp}_{op}(\zeta1, \ldots)$, $\sigma'; \zeta \Downarrow v'$ implies that $v' = \mathcal{O}[\![op]\!](v1', \ldots) = \mathcal{O}[\![op]\!](v1, \ldots) = v$. We conclude that (ζ, σ') corresponds to v.

- $te \equiv (te1\ te2)^{\mathsf{S}}$: Let $\sigma; e1 \Downarrow \langle f, u_1, \ldots, u_n\rangle$, and $\sigma; e2 \Downarrow u_{n+1} \neq \mathtt{Err}$, where $e1 = \phi(te1)$ and $e2 = \phi(te2)$. It follows from the induction hypothesis and Lemma 6.6.3 that $\tau; te1 \downarrow \langle f, \xi_1, \ldots, \xi_n\rangle$, and $\tau; te2 \downarrow \xi_{n+1} \neq \mathtt{Err}$. In addition, each (ξ_i, σ') corresponds to u_i for $i = 1, \ldots, n+1$.

  Suppose that $n < \mathit{ar}(f) - 1$. Then since $v = \langle f, u_1, \ldots, u_{n+1}\rangle$ and $\zeta = \langle f, \xi_1, \ldots, \xi_{n+1}\rangle$, it follows that (ζ, σ') corresponds to v. Now suppose $n = \mathit{ar}(f) - 1$. Then writing σ_1 for $\sigma[f^{(i)} \mapsto u_i]_{i=1,n+1}$ and τ_1 for $\tau[f^{(i)} \mapsto \xi_i]_{i=1,n+1}$, we have $\sigma_1; e^f \Downarrow v$, and $\tau_1; te^f \downarrow \zeta$. Now, (τ_1, σ') corresponds to σ_1 on f, so by the induction hypothesis, (ζ, σ') corresponds to v.

- $te \equiv (te1\ te2)^{\mathsf{D}}$: Let $\sigma; e1 \Downarrow \langle f, u_1, \ldots, u_n\rangle$ and $\sigma; e2 \Downarrow u_{n+1} \neq \mathtt{Err}$, where $e1 = \phi(te1)$ and $e2 = \phi(te2)$. Let $\tau; te1 \downarrow \zeta1$ and $\tau; te2 \downarrow \zeta2$. It follows from the induction hypothesis that $(\zeta1, \sigma')$ corresponds to $\langle f, u_1, \ldots, u_n\rangle$ and $(\zeta2, \sigma')$ corresponds to u_{n+1}, i.e., $\sigma'; \zeta1 \Downarrow \langle f', u_1', \ldots, u_m'\rangle$, which agrees with $\langle f, u_1, \ldots, u_n\rangle$, and $\sigma'; \zeta2 \Downarrow u'_{m+1}$, which agrees with u_{n+1}. Note that we have $\zeta = \mathit{makeApp}(\zeta1, \zeta2)$.

  If $n < \mathit{ar}(f) - 1$, then $\sigma'; \zeta \Downarrow \langle f', u_1', \ldots, u'_{m+1}\rangle$, which is at the same level of instantiation as $\langle f, u_1, \ldots, u_{n+1}\rangle$, because $\langle f', u_1', \ldots, u_m'\rangle$ and $\langle f, u_1, \ldots, u_n\rangle$ are at the same level of instantiation. Let $f' = \mathit{nextFun}(f, \tau_1)$. For $i = 1, \ldots, n$, the value of $f^{(i)}$ used for the specialization $\overline{\tau}_1(f^{(i)})$ matches u_i and for every unmatched pair (t, u) and σ'' such that $\sigma''(f'^{(k)}) = u_k'$, $k = 1, \ldots, m+1$, $\sigma''; t \Downarrow u'$ agreeing with u, because $(\zeta1, \sigma')$ corresponds to $\langle f, u_1, \ldots, u_n\rangle$. Now, by definition, $\overline{\tau}_1(f^{(n+1)}) = f^{(n+1)} \in \mathit{Expr}$, so $\overline{\tau}_1(f^{(n+1)})$ matches u_{n+1} with unmatched pair $(f^{(n+1)}, u_{n+1})$. But $f^{(n+1)} = f'^{(m+1)}$ by design. So for σ'' such that $\sigma''(f'^{(k)}) = u_k'$, $k = 1, \ldots, m+1$, we have $\sigma''; f'^{(m+1)} \Downarrow u'_{m+1}$, which agrees with u_{n+1}, because $(\zeta 2, \sigma')$ corresponds to u_{n+1} (from the induction hypothesis). By definition, $\langle f', u'_1 \ldots, u'_{m+1}\rangle$ and $\langle f, u_1, \ldots, u_{n+1}\rangle$ agree. It follows that (ζ, σ') corresponds to $\langle f, u_1, \ldots, u_{n+1}\rangle = v$.

Now suppose that $n = ar(f) - 1$. By the foregoing, each $\overline{\tau}_1(f^{(i)})$ matches u_i, for $i = 1, \ldots, ar(f)$, and for any unmatched (t, u) and σ'' such that $\sigma''(f'^{(k)}) = u'_k$, $k = 1, \ldots, ar(f')$, $\sigma''; t \Downarrow u'$ agreeing with u. Write σ'' for $\sigma'[f'^{(k)} \mapsto u'_k]_{k=1,ar(f')}$ and σ_1 for $\sigma[f^{(i)} \mapsto u_i]_{i=1,ar(f)}$. Then, by definition, $(\overline{\tau}_1, \sigma'')$ corresponds to σ_1 on f. Since $\overline{\tau}_1; te^f \downarrow e^{f'}$ and $\sigma_1; e^f \Downarrow v$, the induction hypothesis now gives $\sigma''; e^{f'} \Downarrow v'$ agreeing with v. We have $\sigma'; \zeta \Downarrow v'$ (recall that $\zeta = \mathit{makeApp}(\zeta 1, \zeta 2)$), so ζ matches v with unmatched pair (ζ, v), and $\sigma'; \zeta \Downarrow v'$ agreeing with v. Thus, (ζ, σ') corresponds to v.

- $te \equiv (te1\ \ te2)^{\mathtt{M}}$: Let $\sigma; e1 \Downarrow \langle f, u_1, \ldots, u_n\rangle$ and $\sigma; e2 \Downarrow u_{n+1} \neq \mathtt{Err}$, where $e1 = \phi(te1)$ and $e2 = \phi(te2)$. It follows from the induction hypothesis that $\tau; te1 \downarrow \langle f, \xi_1, \ldots, \xi_n\rangle$ and $\tau; te2 \downarrow \xi_{n+1}$ such that each (ξ_i, σ') corresponds to u_i for $i = 1, \ldots, n+1$ (because $(\langle f, \xi_1, \ldots, \xi_n\rangle, \sigma')$ corresponds to $\langle f, u_1, \ldots, u_n\rangle$ and (ξ_{n+1}, σ') corresponds to u_{n+1}). By definition, each ξ_i matches u_i, and for every unmatched pair (t, u), we have $\sigma'; t \Downarrow u'$ agreeing with u.

  Suppose that $n < ar(f) - 1$. Then $\zeta = \mathit{makeMem}(f, \tau_1, n+1) =$ "$f'\ f^{(k_1)}\ \ldots f^{(k_a)}\ \xi_{j_1}\ \ldots\ \xi_{j_d}$", where $f^{(k_1)}, \ldots, f^{(k_a)}$ are all the free variables of $\tau_1(f^{(i)})$ for $i \in I_S$, ordered in the standard way, $f' = \mathit{nextFun}(f, \tau_1)$ and $\tau_1(f^{(i)}) = \xi_i$ for $i = 1, \ldots, n+1$, in particular, for $i \in I_S$. Now, ζ matches $\langle f, u_1, \ldots, u_{n+1}\rangle$ because $\zeta \in \mathit{Expr}$. The only unmatched pair is $(\zeta, \langle f, u_1, \ldots, u_{n+1}\rangle)$, so if $\sigma'; \zeta \Downarrow v'$ agreeing with $\langle f, u_1, \ldots, u_{n+1}\rangle = v$, then (ζ, σ') corresponds to v as required. Therefore, consider the value of v'.

  Since each ξ_{j_k} evaluates in σ' to a value agreeing with u_{j_k} (unravelling the definition of "(ξ_i, σ') corresponds to u_i"), each argument of the compound application ζ evaluates successfully in σ', thus v' exists and is an f' closure $\langle f', u'_1, \ldots, u'_m\rangle$, at the same level of instantiation as $\langle f, u_1, \ldots, u_{n+1}\rangle$ (by design). Now, for $i \in I_S$, we already noted that $\tau_1(f^{(i)}) = \xi_i$, which matches u_i, and for any unmatched (t, u), t evaluates in σ' to some u' agreeing with u (because (ξ_i, σ') corresponds to u_i). We claim that $\overline{\tau}_1(f^{(i)}) = \mathit{SubV}(\tau_1(f^{(i)}), \vec{fv}, \vec{nv})$ (the alpha-converted version of $\tau_1(f^{(i)})$), continues to match u_i. Any unmatched pair has the form (t_1, u), such that there is an unmatched pair (t, u) of $\tau_1(f^{(i)})$ and u_i, with $t_1 = t[\mathit{newvar}_r / f^{(k_r)}]_{r=1,a}$. The evaluation of t_1 in σ'' where $\sigma''(f'^{(k)}) = u'_k$ (implying that $\sigma''(\mathit{newvar}_r) = \sigma'(f^{(k_r)})$) coincides with the evaluation of t in σ', since by design the $f^{(k_r)}$'s are all the variables occurring in t. (In other words, alpha-conversion is correct for L.) In particular, the evaluation of t_1 in σ'' yields a value agreeing with u.

  As for $j \in I_D$, $\overline{\tau}_1(f^{(j)}) = f^{(j)}$, which matches u_j with the unmatched pair $(f^{(j)}, u_j)$. It has been noted that the evaluation of ξ_j in σ' agrees with u_j (see start of last paragraph), so in any σ'' where $\sigma''(f'^{(k)}) = u'_k$, $k = 1, \ldots, m$, the value of $\sigma''(f^{(j)})$, which is the evaluation of ξ_j in σ' (by design of *makeMem* and *nextMem*) agrees with u_j. Putting everything together, for each $i = 1, \ldots, n+1$, $\overline{\tau}_1(f^{(i)})$ matches u_i, and for any unmatched pair (t, u), and σ'' where $\sigma''(f'^{(k)}) = u'_k$, $k = 1, \ldots, m$, it holds that $\sigma''; t \Downarrow u'$ agreeing with u. This, and the fact that $v' = \langle f', u'_1, \ldots, u'_m\rangle$ and $v = \langle f, u_1, \ldots, u_{n+1}\rangle$ are at the same level of instantiation, establish that v' agrees with v, as desired.

  Finally, suppose that $n = ar(f) - 1$. Then v is the evaluation of e^f in $\sigma_1 = \sigma[f^{(i)} \mapsto u_i]_{i=1,ar(f)}$. And $\zeta = \mathit{makeMem}(f, \tau_1, ar(f)) =$"$f'\ f^{(k_1)}\ \ldots\ f^{(k_a)}\ \xi_{j_1}\ \ldots\ \xi_{j_d}$", where $\tau_1(f^{(i)}) = \xi_i$, $i = 1, \ldots, ar(f)$, as before. Since $\zeta \in \mathit{Expr}$, it matches v with unmatched pair (ζ, v), and we have to prove that $\sigma'; \zeta \Downarrow v'$ agreeing with v, for then (ζ, σ') corresponds to v.

  By design, ζ is a call. For $n < ar(f) - 1$, the evaluation of ζ in σ' was an f' closure of the form $\langle f', u'_1, \ldots, u'_m\rangle$, where $f' = \mathit{nextFun}(f, \tau_1)$, at the same level of instantiation as $\langle f, u_1, \ldots, u_n\rangle$. For

$n = ar(f) - 1$, the evaluation of ζ in σ' is the evaluation of $e^{f'}$ in the updated environment of $\sigma'' = \sigma'[f'^{(k)} \mapsto u'_k]_{k=1,ar(f')}$. We can argue as before that for $i = 1, \ldots, ar(f)$, $\overline{\tau}_1(f^{(i)})$ matches u_i, and for any unmatched pair (t, u), and σ''' where $\sigma'''(f'^{(k)}) = u'_k$, we have $\sigma'''; t \Downarrow u'$ agreeing with u. In particular, $\sigma''; t \Downarrow u'$ agreeing with u. In other words, $(\overline{\tau}_1(f^{(i)}), \sigma'')$ corresponds to u_i for $i = 1, \ldots, ar(f)$. By definition, $(\overline{\tau}_1, \sigma'')$ corresponds to σ_1 on f. As $\overline{\tau}_1; te^f \downarrow e^{f'}$ and $\sigma_1; e^f \Downarrow v$, it follows from the induction hypothesis that $\sigma''; e^{f'} \Downarrow v'$ agreeing with v. Since the evaluation of ζ in σ' is the evaluation of $e^{f'}$ in σ'', we have established that $\sigma'; \zeta \Downarrow v'$ where v' agrees with v, as desired.

Special treatment for residual functions needing no arguments: Suppose that $a = d = 0$. Then f' should not take any argument at all. This is of course not allowed in L. By design, $\zeta = (f'\ \texttt{nil})$ and f' has a single unused parameter. Thus, ζ still leads to a function-call transition, and the rest of the reasoning is not affected.

Very useful observation: Let (τ, σ') correspond to σ on g, and te be an application subexpression of te^g, annotated `M` or `D`. Suppose that for $e = \phi(te)$, we have $\sigma; e$ leading to the evaluation of $\sigma_1; e^f$. If $\tau; te \downarrow \zeta$, we have seen that $\sigma'; \zeta$ leads to the evaluation of $\sigma''; e^{f'}$, where for some τ', the pair (τ', σ'') corresponds to σ_1 on f and $\tau'; te^f \downarrow e^{f'}$. This observation will save us considerable *pain* in further correctness proofs.

Theorem B.1.2 Let p' be a specialization of tp, a well-annotated version of p *without memoization points*. Suppose that (τ_0, σ'_0) corresponds to σ_0 on $f_{initial}$, and $\tau_0; te^{f_{initial}} \downarrow \zeta_0$. If $\sigma_0; e^{f_{initial}}$ diverges, i.e., there is an infinite $\rightsquigarrow$ chain starting with $\sigma_0; e^{f_{initial}}$, then either $\sigma'_0; \zeta_0$ diverges or $\sigma'_0; \zeta_0 \Downarrow \texttt{Err}$.

Proof We do not want to consider an application $te \equiv (te1\ te2)^{\mathtt{S}}$ if it is dangerous to unfold te some number of arguments. By design, an expression such as te always occurs as a rator within an `M`-annotated application. In the proof, te will be considered at the `M`-annotated application containing it. Refer to expressions like te as *internal*. By design, internal expressions do not cause any unfolding transitions. Note that te^f is definitely *not* internal.

Suppose that (τ, σ') corresponds to σ on g, and $\sigma; e$ is reachable and divergent. Let te be a non-internal subexpression of te^g, and put $e = \phi(te)$. We claim that for some non-internal subexpression te_1 of te^f, with $e_1 = \phi(te_1)$, $\sigma; e \rightsquigarrow^+ \sigma_1; e_1$ such that $\sigma_1; e_1$ is divergent, and one of the following is true.

(A) $bt(te) = bt(te_1) = D$, and $\tau; te \overset{S}{\hookrightarrow}^+ \tau_1; te_1$ such that for some ζ, both $\tau; te$ and $\tau_1; te_1$ evaluate to ζ, and (τ_1, σ') corresponds to σ_1 on f.

This corresponds to a simplification step at specialization-time, which preserves the divergent subcomputation in the residual program. Call $((\sigma_1; e_1), (\tau_1; te_1), (\sigma'; \zeta))$ the *successor triple*.

(B) $bt(te) = bt(te_1) = D$, $\tau; te \downarrow \zeta$, and $\sigma'; \zeta \rightsquigarrow^+ \sigma''; \zeta_1$ such that for some τ_1, $\tau_1; te_1 \downarrow \zeta_1$, and (τ_1, σ'') corresponds to σ_1 on f.

This case corresponds to the residual program making a transition to ζ_1. (*Point*: this should happen every so often.) Let the successor triple be $((\sigma_1; e_1), (\tau_1; te_1), (\sigma''; \zeta_1))$.

(C) $bt(te_1) = S$, and $\tau; te \overset{S}{\hookrightarrow}^+ \tau_1; te_1$ such that (τ_1, σ') corresponds to σ_1 on f.

In this case, the divergent subcomputation has static binding-time. Let the successor triple be $((\sigma_1; e_1), (\tau_1; te_1), (\sigma'; \bullet))$.

(D) $bt(te) = D$, $\tau; te \downarrow \zeta$, and $\sigma'; \zeta \Downarrow \texttt{Err}$.

In this case, the residual program evaluates erroneously instead of diverging. There is no transition triple.

These cases indicate how a divergent computation of the source program can manifest itself in the specialization, or in the computation of the residual program.

By assumption, (τ_0, σ_0') corresponds to σ_0 on $f_{initial}$, and $\sigma_0; e^{f_{initial}}$ is divergent. By design, $e^{f_{initial}}$ is non-internal. So the premise of the above claim is satisfied initially. Starting with the following triple:

$$((\sigma_0; e^{f_{initial}}), (\tau_0; te^{f_{initial}}), (\sigma_0'; \zeta_0)),$$

we find that each of cases (A), (B), (C) defines a next triple $((\sigma_1; e_1), (\tau_1; te_1), (\sigma''; \zeta_1))$ such that (τ_1, σ'') corresponds to σ_1 on the function of e_1, the evaluation state $\sigma_1; e_1$ is divergent, and te_1 is non-internal. Applying the claim inductively, we deduce a sequence of triples:

$$((\sigma_0; e_0), (\tau_0; te_0), (\sigma_0'; \zeta_0)), ((\sigma_1; e_1), (\tau_1; te_1), (\sigma_1'; \zeta_1)), \ldots ((\sigma_t; e_t), (\tau_t; te_t), (\sigma_t'; \zeta_t)), \ldots$$

Suppose that the chain is finite. Then (D) eventually applies, i.e., for some t, we have $\sigma_t'; \zeta_t \Downarrow \texttt{Err}$. Each $\sigma_t'; \zeta_t$ is either connected to $\sigma_{t+1}'; \zeta_{t+1}$ by $\rightsquigarrow$ transitions, or they are equal. Thus, $\sigma_0'; \zeta_0 \rightsquigarrow^* \sigma_t'; \zeta_t$. It is not difficult to show that $\sigma_0'; \zeta_0 \Downarrow \texttt{Err}$ must hold when $\sigma_0'; \zeta_0 \rightsquigarrow^* \sigma_t'; \zeta_t$ and $\sigma_t'; \zeta_t \Downarrow \texttt{Err}$.

Suppose that the chain is infinite, but ζ_t is eventually $\bullet$. Only (C) applies thereafter, since (A) and (B) are for expressions with dynamic binding-time. (It is obvious that the subcomputation states of states with static binding-times have static binding-times.) Let $\zeta_{t-1} \neq \bullet$ but $\zeta_t = \zeta_{t+1} = \ldots = \bullet$. We deduce an infinite $\overset{S}{\hookrightarrow}$ chain from $\tau_{t-1}; te_{t-1}$. We claim, without proof, that this is impossible for the successful specialization of tp.

Finally, suppose that the chain is infinite, and $\zeta_t \neq \bullet$ for all t. Define a residual program $\rightsquigarrow$ chain by starting with $\sigma_0'; \zeta_0$, and extending it whenever case (B) applies. For each $\sigma_t'; \zeta_t$ equal to $\sigma_{t+1}'; \zeta_{t+1}$, correspondingly $\tau_t; te_t \overset{S}{\hookrightarrow}^+ \tau_{t+1}; te_{t+1}$ (case (A)). Since any $\overset{S}{\hookrightarrow}$ chain must be finite for the $(\zeta)_t$ to exist, the $\sigma_t'; \zeta_t$ chain is always extendible. An infinite $\rightsquigarrow$ chain starting with $\sigma_0'; \zeta_0$ is therefore deduced. In other words, $\sigma_0'; \zeta_0$ is divergent. This completes the proof of the theorem.

The proof of the claim proceeds by case analysis on the form of te. In the following, te of the form ($\texttt{lift}$ $te1$) will not be considered. The rest of the proof will show that case (C) is applicable for any te with static binding-time. Since the binding-time of $te1$ in ($\texttt{lift}$ $te1$) must be static, it follows that case (C) is applicable for any ($\texttt{lift}$ $te1$).

- $te \equiv (\texttt{if}\ te1\ te2\ te3)^{\texttt{S}}$ where $bt(te) = S$: Then $bt(te1) = bt(te2) = bt(te3) = S$. By design, $te1$, $te2$ and $te3$ are *not* internal. Let $e1 = \phi(te1)$, $e2 = \phi(te2)$ and $e3 = \phi(te3)$. If $\sigma; e \rightsquigarrow \sigma; e1$ starts the infinite $\rightsquigarrow$ chain, then case (C) holds with $\sigma_1; e_1 = \sigma; e1$ and $\tau_1; te_1 = \tau; te1$. Now if $\sigma; e \rightsquigarrow \sigma; e2$ starts the infinite chain, then by Theorem 6.7.3 and well-annotatedness, $\tau; te1 \downarrow \zeta1$ where $\zeta1 \in \mathit{Baseval} \setminus \{\texttt{False}, \texttt{Err}\}$, so (C) holds with $\sigma_1; e_1 = \sigma; e2$ and $\tau_1; te_1 = \tau, te2$. The case with $\sigma; e \rightsquigarrow \sigma; e3$ starting the infinite chain is similar.

- $te \equiv (\texttt{if}\ te1\ te2\ te3)^{\texttt{S}}$ where $bt(te) = D$: Then $bt(te1) = S$ and $bt(te2) = bt(te3) = D$. By design, $te1$, $te2$ and $te3$ are *not* internal. Let $e1 = \phi(te1)$, $e2 = \phi(te2)$ and $e3 = \phi(te3)$. If $\sigma; e \rightsquigarrow \sigma; e1$ starts the infinite chain, argue as above that (C) holds. If $\sigma; e2$ starts the infinite chain, by Theorem 6.7.3 and well-annotatedness, $\tau; te1 \downarrow \zeta1$ where $\zeta1 \in \mathit{Baseval} \setminus \{\texttt{False}, \texttt{Err}\}$, so $\tau; te \downarrow \zeta2$ where $\tau; te2 \downarrow \zeta2$. Thus case (A) holds with $\sigma_1; e_1 = \sigma; e2$ and $\tau_1, te_1 = \tau; te2$. The case of $\sigma; e \rightsquigarrow \sigma; e3$ starting the infinite chain is similar.

- $te \equiv (\texttt{if}\ te1\ te2\ te3)^{\texttt{D}}$: Then $bt(te1) = bt(te2) = bt(te3) = D$. By design, $te1$, $te2$ and $te3$ are *not* internal. Let $e1 = \phi(te1)$, $e2 = \phi(te2)$ and $e3 = \phi(te3)$. Now suppose that $\sigma; e \rightsquigarrow \sigma; e1$ starts the infinite chain. Then as $\tau; te \downarrow \zeta = \mathit{makeIf}(\zeta1, \zeta2, \zeta3)$ where $\tau; te1 \downarrow \zeta1$, we have $\sigma'; \zeta \rightsquigarrow \sigma'; \zeta1$. In this case, (B) holds with $\sigma''; \zeta_1 = \sigma'; \zeta1$, $\tau_1; te_1 = \tau; te1$ and $\sigma_1; e_1 = \sigma; e1$. If $\sigma; e \rightsquigarrow \sigma; e2$ starts the infinite chain, then by Theorem 6.7.3, $\sigma'; \zeta1 \Downarrow v1'$ agreeing with $v1$, where $\sigma; e1 \Downarrow v1$. In particular $v1' \notin \{\texttt{False}, \texttt{Err}\}$. So $\sigma'; \zeta \rightsquigarrow \sigma'; \zeta2$, and (B) holds with $\sigma''; \zeta_1 = \sigma'; \zeta2$, $\tau_1; te_1 = \tau; te2$ and $\sigma_1; e_1 = \sigma; e2$. The case for $\sigma; e \rightsquigarrow \sigma; e3$ starting the infinite chain is similar.

- $te \equiv (op\ te1\ te2\ \ldots)^{\mathtt{S}}$: Then $bt(te1) = bt(te2) = \ldots = S$. By design, $te1, te2, \ldots$ are *not* internal. Let $e1 = \phi(te1)$. Suppose that $\sigma; e \rightsquigarrow \sigma; e1$ starts the infinite chain. (The same reasoning applies for any other actual argument of the operation.) Since $\tau; te \overset{S}{\hookrightarrow} \tau; te1$, case (C) holds with $\sigma_1; e_1 = \sigma; e1$ and $\tau_1; te_1 = \tau; te1$.

- $te \equiv (op\ te1\ te2\ \ldots)^{\mathtt{D}}$: Then $bt(te1) = bt(te2) = \ldots = D$. By design, $te1, te2, \ldots$ are *not* internal. Let $e1 = \phi(te1)$. Suppose that $\sigma; e \rightsquigarrow \sigma; e1$ starts the infinite chain. (The same reasoning applies for any other argument.) Now, $\tau; te \downarrow \zeta = \mathit{makeOp}_{op}(\zeta1, \ldots)$ where $\tau; e1^D \downarrow \zeta1$. Since $\sigma'; \zeta \rightsquigarrow \sigma'; \zeta1$, case (B) holds with $\sigma''; \zeta_1 = \sigma'; \zeta1$, $\tau_1; te_1 = \tau; te1$ and $\sigma_1; e_1 = \sigma; e1$.

- $te \equiv (te1\ te2)^{\mathtt{S}}$: Then $bt(te1) = S$. Now, te is non-internal implies that $te1$ is non-internal. By design, $te2$ is non-internal. Let $e1 = \phi(te1)$ and $e2 = \phi(te2)$. If $\sigma; e \rightsquigarrow \sigma; e1$ starts the infinite chain, case (C) holds with $\sigma_1; e_1 = \sigma; e1$ and $\tau_1; te_1 = \tau; te1$. If $\sigma; e \rightsquigarrow \sigma; e2$ starts the infinite chain with $bt(te) = S$, then by Theorem 6.7.3, $\tau; te1 \downarrow \zeta1$ such that $(\zeta1, \sigma')$ corresponds to $v1$, where $\sigma; e1 \Downarrow v1$, i.e., $\zeta1$ is a closure. Thus $\tau; te \overset{S}{\hookrightarrow} \tau; te2$ and (C) holds with $\sigma_1; e_1 = \sigma; e2$ and $\tau_1; te_1 = \tau; te2$. If $\sigma; e \rightsquigarrow \sigma; e2$ starts the infinite chain with $bt(te2) = D$, then te would be internal (a static application for which it is dangerous to unfold some number of arguments), so there is nothing to do here, since te is non-internal by assumption.

  Next, suppose that $\sigma; e1 \Downarrow \langle f, u_1, \ldots, u_{ar(f)-1}\rangle$ and $\sigma; e2 \Downarrow u_{ar(f)} \neq \mathtt{Err}$, and the function-call transition $\sigma; e \rightsquigarrow \sigma[f^{(i)} \mapsto u_i]; e^f$ starts the infinite chain. Let $\sigma_1 = \sigma[f^{(i)} \mapsto u_i]$ and $e_1 = e^f$. By Theorem 6.7.3, $\tau; te1 \downarrow \langle f, \xi_1, \ldots, \xi_{ar(f)-1}\rangle$ and $\tau; te2 \downarrow \xi_{ar(f)} \neq \mathtt{Err}$, where each (ξ_i, σ') corresponds to u_i. Therefore, if we let $\tau_1 = \tau[f^{(i)} \mapsto \xi_i]$, then (τ_1, σ') corresponds to σ_1 on f. Put $te_1 = te^f$. Supposing that $bt(te) = S$, (C) is satisfied with the states $\sigma_1; e_1$ and $\tau_1; te_1$. If $bt(te) = D$, (A) is satisfied with the same states, as $\tau; te \overset{S}{\hookrightarrow} \tau_1; te^f$, and both states evaluate to the same result.

- $te \equiv (te1\ te2)^{\mathtt{D}}$: Then $bt(te1) = bt(te2) = D$. By design, $te1$ and $te2$ are *not* internal. Let $e1 = \phi(te1)$ and $e2 = \phi(te2)$. Suppose that $\sigma; e \rightsquigarrow \sigma; e1$ starts the infinite chain. As $\tau; te \downarrow \zeta$ where ζ has the form $\mathit{makeApp}(\zeta1, \zeta2)$, we have $\sigma'; \zeta \rightsquigarrow \sigma'; \zeta1$ where $\tau; te1 \downarrow \zeta1$. Thus, (B) holds with $\sigma''; \zeta_1 = \sigma'; \zeta1$, $\tau_1; te_1 = \tau; te1$ and $\sigma_1; e_1 = \sigma; e1$. If it is $\sigma; e \rightsquigarrow \sigma; e2$ that starts the infinite chain, the state $\sigma; e1$ must evaluate successfully to a closure $v1$, so $\sigma'; \zeta1 \Downarrow v1'$ agreeing with $v1$ by Theorem 6.7.3. It follows that $\sigma'; \zeta \rightsquigarrow \sigma'; \zeta2$ where $\tau; te2 \downarrow \zeta2$. In this case, (B) is satisfied with $\sigma''; \zeta_1 = \sigma'; \zeta2$, $\tau_1; te_1 = \tau; te2$ and $\sigma_1; e_1 = \sigma; e2$.

  Next, suppose that $\sigma; e1 \Downarrow \langle f, u_1, \ldots, u_n\rangle$ with $n = ar(f) - 1$ and $\sigma; e2 \Downarrow u_{n+1} \neq \mathtt{Err}$, and $\sigma; e \rightsquigarrow \sigma[f^{(i)} \mapsto u_i]; e^f$ starts the infinite chain. Let $\sigma_1 = \sigma[f^{(i)} \mapsto u_i]$ and $e_1 = e^f$. By the observation made after the proof of B.1.1, for $\tau; te \downarrow \zeta$, we have $\sigma'; \zeta \rightsquigarrow \sigma''; e^{f'}$, where for some τ', it holds that $\tau'; te^f \downarrow e^{f'}$. Put $\tau_1 = \tau'$ and $\zeta_1 = e^{f'}$. In the updated notation: $\tau; te \downarrow \zeta$, $\sigma'; \zeta \rightsquigarrow \sigma''; \zeta_1$, $\tau_1; te_1 \downarrow \zeta_1$, and (τ_1, σ'') corresponds to σ_1 on the function of e_1. Further, we have $bt(te_1) = D$, te_1 is non-internal, and $\sigma; e \rightsquigarrow \sigma_1; e_1$ starts an infinite chain. Therefore (B) is satisfied.

- $te \equiv (te1\ te2)^{\mathtt{M}}$: Write te as $(te_{(0)}\ te_{(1)}\ \ldots\ te_{(k)})^{\mathtt{M}}$ such that each $te_{(i)}$ is non-internal, omitting the annotations on the *intermediate applications* $(te_{(0)}\ te_{(1)})^{\mathtt{S}}$, $((te_{(0)}\ te_{(1)})^{\mathtt{S}}\ te_{(2)})^{\mathtt{S}}$, ... Observe that $bt(te_{(0)}) = S$. Put $e = \phi(te)$, and $e_{(i)} = \phi(te_{(i)})$ for each i.

  If it is $\sigma; e \rightsquigarrow^+ \sigma; e_{(0)}$ that diverges, then as $\tau; te \overset{S}{\hookrightarrow}{}^+ \tau; te_{(0)}$, (C) is satisfied with $\sigma_1; e_1 = \sigma; e_{(0)}$ and $\tau_1; e_1 = \tau; te_{(0)}$. If an intermediate application $(e_{(0)}\ \ldots e_{(q)})$ $(q < k)$ causes a function-call transition that diverges, i.e., $\sigma; (e_{(0)}\ \ldots\ e_{(q-1)}) \Downarrow \langle f, u_1, \ldots, u_{ar(f)-1}\rangle$, $\sigma; e_{(q)} \Downarrow u_{ar(f)} \neq \mathtt{Err}$, and $\sigma; e \rightsquigarrow^+ \sigma[f^{(i)} \mapsto u_i]; e^f$ starts the infinite chain, then reason as follows. Let $\sigma_1 = \sigma[f^{(i)} \mapsto u_i]$ and

$e_1 = e^f$. By Theorem 6.7.3, $\tau; (te_{(0)} \ \ldots \ te_{(q-1)})^{\mathtt{S}} \downarrow \langle f, \xi_1, \ldots, \xi_{ar(f)-1}\rangle$ and $\tau; te_{(q)} \downarrow \xi_{ar(f)} \neq \mathtt{Err}$ such that each (ξ_i, σ') corresponds to u_i. Let $\tau_1 = \tau[f^{(i)} \mapsto \xi_i]$. Then (τ_1, σ') corresponds to σ_1 on f. Put $te_1 = te^f$, and note that te_1 is non-internal. Since the binding-time of the intermediate application is static, case (C) is satisfied with the states $\sigma_1; e_1$ and $\tau_1; te_1$.

Otherwise, it may be one of the actual arguments that diverges. Without any loss of generality, assume that $\sigma; e \rightsquigarrow^+ \sigma; e_{(1)}$ starts the infinite chain. Consider the rator of the application with this argument, here $e_{(0)}$. By Theorem 6.7.3, if $\sigma; e_{(0)} \Downarrow v_{(0)}$, then $\tau; te_{(0)} \downarrow \zeta_{(0)}$ such that $(\zeta_{(0)}, \sigma')$ corresponds to $v_{(0)}$. In particular, as $v_{(0)}$ is a closure, so is $\zeta_{(0)}$ ($v_{(0)}$ is a closure for evaluation to reach $e_{(1)}$). If $bt(te_{(1)}) = S$, then as $\tau; te \overset{S}{\hookrightarrow}{}^+ \tau; te_{(1)}$, (C) is satisfied with $\sigma_1; e_1 = \sigma; e_{(1)}$ and $\tau_1; te_1 = \tau; te_{(1)}$. If $bt(te_{(1)}) = D$, the reasoning is much more involved.

Suppose that $bt(te_{(1)}) = D$. Now, $v_{(0)}$ has the form $\langle f, u_1, \ldots, u_n\rangle$. Argue that $n \leq ar(f) - k$ as follows: It is dangerous to unfold $(te_{(0)} \ te_{(1)})^{\mathtt{S}}$ more than $ar(f){-}n{-}1$ arguments, so if $n > ar(f){-}k$, there would be a memoization point among the intermediate applications of te. (Strictly, we require a correctness result for the determination of memoization points.) This implies that $\zeta_{(0)}$, which has the form $\langle f, \xi_1, \ldots, \xi_n\rangle$, *cannot* lead to any unfolding transition given the available arguments. Let $\tau; te_{(1)} \downarrow \zeta_{(1)}$. It follows that $\zeta_{(1)}$ appears in the residual application, *unless* a static error occurs among $te_{(2)}, \ldots, te_{(k)}$, in which case $\tau; te \downarrow error_D$, and (D) is satisfied. (Consult the *2L* semantics.) Suppose that $\tau; te$ does *not* evaluate to $error_D$. Then for $\tau; te \downarrow \zeta$, ζ is a residual application. As $(\zeta_{(0)}, \sigma')$ corresponds to $\langle f, u_1, \ldots, u_n\rangle$, it can be deduced that any actual argument occurring earlier than $\zeta_{(1)}$ in ζ evaluates successfully with σ'. Thus $\sigma'; \zeta \rightsquigarrow^+ \sigma'; \zeta_{(1)}$. Letting $\sigma''; \zeta_1 = \sigma'; \zeta_{(1)}$, $\tau_1; te_1 = \tau; te_{(1)}$ and $\sigma_1; e_1 = \sigma; e_{(1)}$, all the conditions for (B) are satisfied.

Finally, let $\sigma; e1 \Downarrow \langle f, u_1, \ldots, u_n\rangle$ with $n = ar(f) - 1$ and $\sigma; e2 \Downarrow u_{n+1} \neq \mathtt{Err}$, and suppose that $\sigma; e \rightsquigarrow \sigma[f^{(i)} \mapsto u_i]; e^f$ starts the infinite chain. Let $\sigma_1 = \sigma[f^{(i)} \mapsto u_i]$ and $e_1 = e^f$. By the observation made after the proof of B.1.1, for $\tau; te \downarrow \zeta$, we have $\sigma'; \zeta \rightsquigarrow \sigma''; e^{f'}$, where for some τ', it holds that $\tau'; te^f \downarrow e^{f'}$. Let $\tau_1 = \tau'$, $te_1 = te^f$ and $\zeta_1 = e^{f'}$. In the updated notation: $\tau; te \downarrow \zeta$, $\sigma'; \zeta \rightsquigarrow \sigma''; \zeta_1$, $\tau_1; te_1 \downarrow \zeta_1$, and (τ_1, σ'') corresponds to σ_1 on the function of e_1. Further, we have that $bt(te_1) = D$, te_1 is non-internal, and $\sigma; e \rightsquigarrow \sigma_1; e_1$ starts an infinite chain. Therefore (B) is satisfied.

Theorem B.1.3 Let p' be a specialization of tp, a well-annotated version of p *without memoization points.* Consider a subexpression te of te^g such that either te does *not* have the form $(te1 \ te2)^{\mathtt{S}}$, or it is *not* dangerous to unfold te any number of arguments. In particular, te may be $te^{f_{initial}}$. For the proof, refer to expressions like te as *non-internal.* Note that by design, every function body is non-internal.

Suppose that (τ, σ') corresponds to σ on g, and for $e = \phi(te)$: $\sigma; e$ is reachable and $\sigma; e \Downarrow \mathtt{Err}$. If $\tau; te \downarrow \zeta$, then ζ is *erroneous with* σ': If $bt(te) = S$ then $\zeta = \mathtt{Err}$, and if $bt(te) = D$ then $\sigma'; \zeta \Downarrow \mathtt{Err}$.

Proof Suppose, as stated in the theorem, that (τ, σ') corresponds to σ on g, the function of te, and for $e = \phi(te)$: $\sigma; e$ is reachable and $\sigma; e \Downarrow \mathtt{Err}$. Let $\tau; te \downarrow \zeta$. We must prove that ζ is erroneous with σ'. This is accomplished by induction on the size of the proof tree for $\sigma; e$. We proceed by case analysis on the form of te. In each case, we may assume, by the induction hypothesis, that the theorem holds for any subcomputation of $\sigma; e$. In particular, if an erroneous subcomputation has environment σ, and the annotated version of its expression is specialized with τ, then the result of this specialization is erroneous with σ'.

A note on `lift`*s*: Let $te \equiv (\mathtt{lift}\ te1)$. Suppose that we can show, by induction, that for $\tau; te1 \downarrow \zeta1$ and $\sigma; e \Downarrow \mathtt{Err}$, the specialization result $\zeta1$ is erroneous with σ', i.e., $\zeta1 = \mathtt{Err}$. Then for $\tau; te \downarrow \zeta$, it follows from the *2L* semantics that ζ always evaluates to `Err`. Thus ζ is erroneous with σ'. This reasoning allows us to ignore `lift` operations in the following case analysis. The same reasoning should be applied to $(\mathtt{lift}\ te1)$, for each possible $te1$ from below.

- $te \equiv (\mathtt{if}\ te1\ te2\ te3)^{\mathtt{S}}$: By design, $te1, te2, te3$ are non-internal, and $bt(te1) = S$. Let $e1 = \phi(te1)$, $e2 = \phi(te2)$ and $e3 = \phi(te3)$. If $\sigma; e1 \Downarrow \mathtt{Err}$, it follows from the induction hypothesis that $\tau; te1 \downarrow \zeta1$, where $\zeta 1$ is erroneous with σ'. Since $bt(te1) = S$, we have $\zeta 1 = \texttt{Err}$. So $\zeta = error_{bt}$, for $bt = bt(te)$. It follows that ζ is erroneous with *any* environment.

Next, suppose that $\sigma; e1 \Downarrow v1$ where $v1 \notin \{\texttt{False}, \texttt{Err}\}$ and $\sigma; e2 \Downarrow \texttt{Err}$. It follows from Theorem 6.7.3 and well-annotatedness that for $\tau; te1 \downarrow \zeta 1$, we have $\zeta 1 = v1 \in \mathit{Baseval} \setminus \{\texttt{False}, \texttt{Err}\}$. Thus, ζ is the result of specializing $\tau; te2$, which is erroneous with σ' by the induction hypothesis. The case for $\sigma; e1 \Downarrow \texttt{False}$ and $\sigma; e3 \Downarrow \texttt{Err}$ is similar.

- $te \equiv (\texttt{if}\ te1\ te2\ te3)^{\mathsf{D}}$: Then $\zeta = \mathit{makeIf}(\zeta 1, \zeta 2, \zeta 3)$ where $\tau; te1 \downarrow \zeta 1$, $\tau; te2 \downarrow \zeta 2$ and $\tau; te3 \downarrow \zeta 3$. By design, $te1, te2, te3$ are non-internal, and $bt(te1) = bt(te2) = bt(te3) = D$. Let $e1 = \phi(te1)$, $e2 = \phi(te2)$ and $e3 = \phi(te3)$. Now suppose that $\sigma; e1 \Downarrow \texttt{Err}$. It follows from the induction hypothesis that $\sigma'; \zeta 1 \Downarrow \texttt{Err}$, so $\sigma'; \zeta \Downarrow \texttt{Err}$. If $\sigma; e1 \Downarrow v1 \notin \{\texttt{False}, \texttt{Err}\}$, and $\sigma; e2 \Downarrow \texttt{Err}$, argue as follows. By Theorem 6.7.3, $\sigma'; \zeta 1 \Downarrow v1'$ agreeing with $v1$. In particular, $v1' \notin \{\texttt{False}, \texttt{Err}\}$. Thus, $\sigma'; \zeta$ evaluates the same as $\sigma'; \zeta 2$, whose result is error from the induction hypothesis. The case for $\sigma; e1 \Downarrow \texttt{False}$ and $\sigma; e3 \Downarrow \texttt{Err}$ is similar.

- $te \equiv (op\ te1\ te2\ \ldots)^{\mathsf{S}}$: By design, $te1, te2, \ldots$ are non-internal, and $bt(te1) = bt(te2) = \ldots = S$. Let $e1 = \phi(te1)$, $e2 = \phi(te2)$, ... Now suppose that $\sigma; e1 \Downarrow v1$ and $\tau; te1 \downarrow \zeta 1$. If $v1 = \texttt{Err}$, it follows from the induction hypothesis that $\zeta 1 = \texttt{Err}$. By an assumption about $\mathcal{O}$, the value of $\mathcal{O}[\![op]\!](\zeta 1, \ldots)$ is undefined. By the *2L* semantics, $\zeta = \texttt{Err}$. If $v1 \neq \texttt{Err}$, then by Theorem 6.7.3 and well-annotatedness, $\zeta 1 = v1 \in \mathit{Baseval} \setminus \{\texttt{Err}\}$. The same comments apply to other actual arguments $e2, \ldots$ of the operation. Since we would have $\zeta = \texttt{Err}$ if any actual argument evaluates in σ to $\texttt{Err}$, let us assume that each one evaluates in σ to a non-error base value. In this case, we deduce that $\mathcal{O}[\![op]\!](v1, \ldots)$ is undefined. It follows that $\mathcal{O}[\![op]\!](\zeta 1, \ldots)$ is undefined. By the *2L* semantics, $\zeta = \texttt{Err}$.

- $te \equiv (op\ te1\ te2\ \ldots)^{\mathsf{D}}$: By design, $te1, te2, \ldots$ are non-internal, and $bt(te1) = bt(te2) = \ldots = D$. Let $e1 = \phi(te1)$, $e2 = \phi(te2)$, ... The following reasoning about $e1$ applies to *every* actual argument of the operation. Suppose that $\sigma; e1 \Downarrow v1$ and $\tau; te1 \downarrow \zeta 1$. From the induction hypothesis, if $v1 = \texttt{Err}$, then $\sigma'; \zeta 1 \Downarrow \texttt{Err}$. If $v1$ is a closure, then by Theorem 6.7.3, $\sigma'; \zeta 1 \Downarrow v1'$, where $v1'$ is a closure agreeing with $v1$. Finally, if $v1$ is a non-error base value, we deduce that $\sigma'; \zeta 1 \Downarrow v1$.

  Since $\zeta = \mathit{makeOp}_{op}(\zeta 1, \ldots)$, the only way for ζ *not* to be erroneous with σ' would be for each of the actual arguments $e1, e2, \ldots$ to evaluate in σ to a non-error base value (using an assumption about $\mathcal{O}$). In this case, we deduce that $\mathcal{O}[\![op]\!](v1, \ldots)$ is undefined. It follows that $\sigma'; \zeta \Downarrow \texttt{Err}$.

- $te \equiv (te1\ te2)^{\mathsf{S}}$: By design, $te1$ is non-internal, and $bt(te1) = S$. Let $e1 = \phi(te1)$ and $e2 = \phi(te2)$. If $\sigma; e1 \Downarrow \texttt{Err}$, it follows from the induction hypothesis that $\tau; te1 \downarrow \texttt{Err}$, so $\zeta = error_{bt}$ for $bt = bt(te)$. It follows that ζ is erroneous with σ'. If $\sigma; e1 \Downarrow v1$ such that $v1 \neq \texttt{Err}$ but $v1 \notin \mathit{Clo}$, then by Theorem 6.7.3, $\tau; te1 \downarrow \zeta 1$ such that $\zeta 1 \notin \mathit{2Clo}$. Once again, $\zeta = error_{bt}$ for $bt = bt(te)$. Now let $\sigma; e1$ evaluate to a closure and $\sigma; e2 \Downarrow \texttt{Err}$. By Theorem 6.7.3, $\tau; te1 \downarrow \zeta 1 \in \mathit{2Clo}$. From the induction hypothesis, if $bt(te2) = S$, then $\tau; te2 \downarrow \texttt{Err}$, so $\zeta = error_{bt}$ for $bt = bt(te)$. And if $bt(te2) = D$, then te would *not* be non-internal, so there is nothing to prove.

  Another possibility for getting an error is to have $\sigma; e1 \Downarrow \langle f, u_1, \ldots, u_n \rangle$ where $n = ar(f) - 1$ and $\sigma; e2 \Downarrow u_{n+1} \neq \texttt{Err}$, but $\sigma[f^{(i)} \mapsto u_i]; e^f \Downarrow \texttt{Err}$. In this case, by Theorem 6.7.3, we have $\tau; te1 \downarrow \langle f, \xi_1, \ldots, \xi_n \rangle$ and $\tau; te2 \downarrow \xi_{n+1} \neq \texttt{Err}$, such that each (ξ_i, σ') corresponds to u_i. Now, $\tau[f^{(i)} \mapsto \xi_i]; te^f \downarrow \zeta$, and since $(\tau[f^{(i)} \mapsto \xi_i], \sigma')$ corresponds to $\sigma[f^{(i)} \mapsto u_i]$ on f, by the induction hypothesis, ζ is erroneous with σ'.

- $te \equiv (te1\ te2)^{\mathsf{D}}$: Then $\zeta = \mathit{makeApp}(\zeta 1, \zeta 2)$, where $\tau; te1 \downarrow \zeta 1$ and $\tau; te2 \downarrow \zeta 2$. By design, $te1, te2$ are non-internal, and $bt(te1) = bt(te2) = D$. Let $e1 = \phi(te1)$ and $e2 = \phi(te2)$. If $\sigma; e1 \Downarrow \texttt{Err}$, it follows from the induction hypothesis that $\zeta 1$ is erroneous with σ', so ζ is erroneous with σ'. If $\sigma; e1 \Downarrow v1$ such that $v1 \neq \mathtt{Err}$ but $v1 \notin \mathit{Clo}$, then by Theorem 6.7.3, $\sigma'; \zeta 1 \Downarrow v1' \notin \mathit{Clo}$. Once again, ζ is erroneous with σ'. If $v1 \in \mathit{Clo}$ but $\sigma; e2 \Downarrow \mathtt{Err}$, then by Theorem 6.7.3, $\sigma'; \zeta 1 \Downarrow v1'$ agreeing with $v1$, but from the induction hypothesis, $\sigma'; \zeta 2 \Downarrow \mathtt{Err}$, so ζ is erroneous with σ'.

Another possibility for getting an error is to have $\sigma; e1 \Downarrow \langle f, u_1, \ldots, u_n \rangle$ where $n = ar(f) - 1$ and $\sigma; e2 \Downarrow u_{n+1} \neq \mathtt{Err}$, but $\sigma[f^{(i)} \mapsto u_i]; e^f \Downarrow \mathtt{Err}$. Let $\sigma_1 = \sigma[f^{(i)} \mapsto u_i]$. Then by the observation made after the proof of B.1.1, $\sigma'; \zeta$ evaluates to the same result as $\sigma''; e^{f'}$, where for some τ', (τ', σ'') corresponds to σ_1 on f, and $\tau'; te^f \downarrow e^{f'}$. By the induction hypothesis, the result is $\mathtt{Err}$.

- $te \equiv (te1\ te2)^{\mathtt{M}}$: Write te as $(te_{(0)}\ te_{(1)}\ \ldots\ te_{(k)})^{\mathtt{M}}$ such that each $te_{(i)}$ is non-internal, omitting the annotations on the *intermediate applications* $(te_{(0)}\ te_{(1)})^{\mathtt{S}}$, $((te_{(0)}\ te_{(1)})^{\mathtt{S}}\ te_{(2)})^{\mathtt{S}}, \ldots$. Observe that $bt(te_{(0)}) = S$. Put $e = \phi(te)$, and $e_{(i)} = \phi(te_{(i)})$ for each i.

  There are many ways for $\sigma; e$ to evaluate to $\mathtt{Err}$. If $\sigma; e_{(0)} \Downarrow \mathtt{Err}$, it follows from the induction hypothesis that $\tau; te_{(0)} \downarrow \mathtt{Err}$, so ζ is $error_D$, erroneous with *any* environment. If $\sigma; e_{(0)} \Downarrow v_{(0)}$ such that $v_{(0)} \neq \mathtt{Err}$ but $v_{(0)} \notin \mathit{Clo}$, then by Theorem 6.7.3, $\tau; te_{(0)} \downarrow \zeta_{(0)} \notin \mathit{2Clo}$, so once again, ζ is $error_D$.

  If an intermediate application $(e_{(0)}\ \ldots e_{(q)})$ $(q < k)$ causes a function-call transition, and the result is $\mathtt{Err}$, i.e., $\sigma; (e_{(0)}\ \ldots\ e_{(q-1)}) \Downarrow \langle f, u_1, \ldots, u_{ar(f)-1} \rangle$, $\sigma; e_{(q)} \Downarrow u_{ar(f)} \neq \mathtt{Err}$, and $\sigma_1; e^f \Downarrow \mathtt{Err}$ for the updated environment $\sigma_1 = \sigma[f^{(i)} \mapsto u_i]$, the reasoning is more involved. By Theorem 6.7.3, we have $\tau; (te_{(0)}\ \ldots\ te_{(q-1)})^{\mathtt{S}} \downarrow \langle f, \xi_1, \ldots, \xi_{ar(f)-1} \rangle$ and $\tau; te_{(q)} \downarrow \xi_{ar(f)} \neq \mathtt{Err}$ such that each (ξ_i, σ') corresponds to u_i. Let $\tau_1 = \tau[f^{(i)} \mapsto \xi_i]$. Since (τ_1, σ') corresponds to σ_1 on f, and te^f is non-internal, it follows from the induction hypothesis that the result of evaluating τ_1, te^f is $\mathtt{Err}$. We deduce that ζ is $error_D$ again in this case. The remaining ways for $\sigma; e$ to evaluate to $\mathtt{Err}$ are limited. Either one of the actual arguments $e_{(i)}$ evaluates to $\mathtt{Err}$ with σ, or e causes a function-call transition whose result is $\mathtt{Err}$. We consider each of the possibilities below.

  Suppose that an actual argument $e_{(i)}$ evaluates to $\mathtt{Err}$ with σ. Without loss of generality, let $\sigma; e_{(1)} \Downarrow \mathtt{Err}$. Consider the rator of the application with this argument, here $e_{(0)}$. Let it evaluate to $v_{(0)}$ with σ, i.e., let $\sigma; e_{(0)} \Downarrow v_{(0)}$. By assumption, $v_{(0)} \in \mathit{Clo}$. By Theorem 6.7.3, $\tau; te_{(0)} \downarrow \zeta_{(0)}$ such that $(\zeta_{(0)}, \sigma')$ corresponds to $v_{(0)}$. In particular, $\zeta_{(0)} \in \mathit{2Clo}$. If $bt(te_{(1)}) = S$, then from the induction hypothesis, $\tau; te_{(1)} \downarrow \mathtt{Err}$, so ζ is $error_D$. If $bt(te_{(1)}) = D$, the reasoning is much more involved.

  Suppose that $bt(te_{(1)}) = D$. Now, $v_{(0)}$ has the form $\langle f, u_1, \ldots, u_n \rangle$. Argue that $n \leq ar(f) - k$ as follows: It is dangerous to unfold $(te_{(0)}\ te_{(1)})^{\mathtt{S}}$ more than $ar(f) - n - 1$ arguments, so if $n > ar(f) - k$, there would be a memoization point among the intermediate applications of te. (Strictly, we require a correctness result for the determination of memoization points.) This implies that $\zeta_{(0)}$, which has the form $\langle f, \xi_1, \ldots, \xi_n \rangle$, *cannot* lead to any unfolding transition given the available arguments. Let $\tau; te_{(1)} \downarrow \zeta_{(1)}$. It follows that $\zeta_{(1)}$ appears in the residual application, *unless* a static error occurs among $te_{(2)}, \ldots, te_{(k)}$, in which case $\zeta = error_D$. (Consult the *2L* semantics.) Suppose that ζ is *not* $error_D$. Then ζ is a residual application. As $(\zeta_{(0)}, \sigma')$ corresponds to $\langle f, u_1, \ldots, u_n \rangle$, it can be deduced that any actual argument occurring earlier than $\zeta_{(1)}$ in ζ evaluates successfully with σ'. Thus, $\sigma'; \zeta \leadsto^+ \sigma'; \zeta_{(1)}$. From the induction hypothesis, we have $\sigma'; \zeta_{(1)} \Downarrow \mathtt{Err}$. It follows that $\sigma'; \zeta \Downarrow \mathtt{Err}$.

  Finally, suppose that $\sigma; e1 \Downarrow \langle f, u_1, \ldots, u_n \rangle$ with $n = ar(f) - 1$ and $\sigma; e2 \Downarrow u_{n+1} \neq \mathtt{Err}$, but $\sigma[f^{(i)} \mapsto u_i]; e^f \Downarrow \mathtt{Err}$. Let $\sigma_1 = \sigma[f^{(i)} \mapsto u_i]$. By the observation made after the proof of B.1.1, $\sigma'; \zeta$ evaluates to the same result as $\sigma''; e^{f'}$, where for some τ', (τ', σ'') corresponds to σ_1 on f, and $\tau'; te^f \downarrow e^{f'}$. By the induction hypothesis, the result is $\mathtt{Err}$.

Appendix C

Publications by the author

1. Partial Evaluation of the Euclidean Algorithm, Revisited. In *Higher-Order and Symbolic Computation* 12(2), September 1999.

   The usual formulation of the Euclidean Algorithm is not well-suited to be specialized with respect to one of its arguments, at least when using offline partial evaluation. This has led Danvy and Goldberg to reformulate it using bounded recursion. In this article, we show how *The Trick* can be used to obtain a formulation of the Euclidean Algorithm with good binding-time separation. This formulation of the Euclidean Algorithm specializes effectively using standard offline partial evaluation.

2. The Size-Change Principle for Program Termination. In *Proceedings of the ACM Symposium on Principles of Programming Languages* (POPL), January 2001. (Joint work with Neil Jones and Amir Ben-Amram.)

   The "size-change termination" principle for a first-order functional language with well-founded data is: a program terminates on all inputs if *every infinite call sequence* (following program control flow) would cause an infinite descent in some data values.
   Size-change analysis is based only on local approximations to parameter size changes derivable from program syntax. The set of infinite call sequences that follow program flow and can be recognized as causing infinite descent is an ω-regular set, representable by a Büchi automaton. Algorithms for such automata can be used to decide size-change termination. We also give a direct algorithm operating on "size-change graphs" (without the passage to automata).

   Compared to other results in the literature, termination analysis based on the size-change principle is surprisingly simple and general: lexical orders (also called lexicographic orders), indirect function calls and permuted arguments (descent that is not *in-situ*) are all handled *automatically and without special treatment*, with no need for manually supplied argument orders, or theorem-proving methods not certain to terminate at analysis time.

   We establish the problem's *intrinsic complexity.* This turns out to be surprisingly high, complete for PSPACE, in spite of the simplicity of the principle. PSPACE-hardness is proved by a reduction from Boolean program termination. An interesting consequence: The same hardness result applies to many other analyses found in the termination and quasi-termination literature.

Bibliography

[AA99] Andreas Abel and Thorsten Altenkirch. A semantical analysis of structural recursion. In *Abstracts of the Fourth International Workshop on Termination WST'99*, pages 24–25, May 1999.

[ADH$^+$98] Harold Abelson, R. Kent Dybvig, Christopher T. Haynes, Guillermo Juan Rozas, N. I. Adams IV, Daniel P. Friedman, Eugene E. Kohlbecker, Guy L. Steele Jr., David H. Bartley, Robert H. Halstead Jr., Don Oxley, Gerald J. Sussman, G. Brooks, Chris Hanson, K. M. Pitman, and Mitchell Wand. Revised report on the algorithmic language scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105, 1998.

[AG97] Thomas Arts and Jürgen Giesl. Proving innermost termination automatically. In *Proceedings Rewriting Techniques and Applications RTA'97*, volume 1232 of *Lecture Notes in Computer Science*, pages 157–171. Springer, 1997.

[AH96] 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.

[AHU75] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1975.

[And96] Peter Holst Andersen. Termination analysis for offline partial evaluation of a higher order functional language. Master's thesis, DIKU, University of Copenhagen, Denmark, Aug 1996.

[Art97] Thomas Arts. *Automatically Proving Termination and Innermost Normalisation of Term Rewriting Systems.* PhD thesis, Universiteit Utrecht, 1997.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffery D. Ullman. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, 1986.

[BD77] R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, 1977.

[BD91] Anders Bondorf and Olivier Danvy. Automatic autoprojection of recursive equations with global variables and abstract data types. *Science of Computer Programming*, 16(2):151–195, 1991.

[BJ93] Anders Bondorf and Jesper Jørgensen. Efficient analyses for realistic offline partial evaluation: Extended version. Technical Report 93/4, DIKU, University of Copenhagen, Denmark, 1993.

[Bon91a] Anders Bondorf. *Self-applicable Partial Evaluation (DIKU Report 90/17).* PhD thesis, DIKU, University of Copenhagen, Denmark, 1991.

[Bon91b] Anders Bondorf. Similix manual. Technical Report 91/9, DIKU, University of Copenhagen, Denmark, 1991.

[Bon92] Anders Bondorf. Improving binding-times without explicit CPS conversion. In *Proceedings of the 1992 ACM Conference on LISP and Functional Programming, 1992*, pages 1–10. ACM, 1992.

[CC77] Patrick Cousot and Radhia Cousot. A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the Fourth ACM Symposium on Principles of Programming Languages, January 1977*, pages 238–252. ACM, 1977.

[CC79] Patrick Cousot and Radhia Cousout. Systematic design of program analysis frameworks. In *Proceedings of the Sixth ACM Symposium on Principles of Programming Languages, January 1979*, pages 269–282. ACM, 1979.

[CC95] Cliff Click and Keith D. Cooper. Combining analyses, combining optimizations. *ACM Transactions on Programming Languages and Systems*, 17(2):181–196, Mar 1995.

[CFG+97] Agostino Cortesi, Gilberto Filé, Roberto Giacobazzi, Catuscia Palamidessi, and Francesco Ranzato. Complementation in abstract interpretation. *ACM Transactions on Programming Languages and Systems*, 19(1):7–47, Jan 1997.

[Chi91] Wei-Ngan Chin. Generalizing deforestation to all first-order functional programs. In *Workshop on Static Analysis of Equational, Functional and Logic Programming Languages, BIRGE 74*, pages 173–181, 1991.

[CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

[CMB+95] Michael Codish, Anne Mulkers, Maurice Bruynooghe, Maria Garcìa de la Banda, and Manuel V. Hermenegildo. Improving Abstract Interpretations by Combining Domains. *ACM Transactions on Programming Languages and Systems*, 17(1):28–44, Jan 1995.

[Cou81] Patrick Cousot. Semantic foundations of program analysis. In S.S. Muchnick and N.D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 10, pages 303–342. Prentice-Hall, Inc, Englewood Cliffs, New Jersey, 1981.

[CT97] Michael Codish and Cohavit Taboch. A semantic basis for termination analysis of logic programs and its realization using symbolic norm constraints. In Michael Hanus, Jan Heering, and Karl Meinke, editors, *Algebraic and Logic Programming, 6th International Joint Conference, ALP '97–HOA '97, Southampton, U.K., September 3–5, 1997*, volume 1298 of *Lecture Notes in Computer Science*, pages 31–45. Springer, 1997.

[DDF93] Stefaan Decorte, Danny De Schreye, and Massimo Fabris. Automatic inference of norms: A missing link in automatic termination analysis. In D. Miller, editor, *Proceedings of the 1993 International Symposium on Logic Programming*, pages 420–436. MIT Press, 1993.

[DG97] Olivier Danvy and Mayer Goldberg. Partial evaluation of the euclidean algorithm. *LISP and Symbolic Computation*, 10:101–111, 1997.

[DH93] Nachum Dershowitz and Charles Hoot. Topics in termination. In Claude Kirchner, editor, *Rewriting Techniques and Applications, Proceedings of the 5th International Conference, RTA-93, Montreal, Canada, Jun 16–18, 1993*, volume 690 of *Lecture Notes in Computer Science*, pages 198–212. Springer, 1993.

[DLSS99] Nachum Dershowitz, Naomi Lindenstrauss, Yehoshua Sagiv, and Alexander Serebrenik. Automatic termination analysis of programs containing arithmetic predicates. In *International Conference on Logic Programming: Workshop on Verification of Logic Programs, Las Cruces, New Mexico, USA, December 1, 1999*, volume 30(1) of *Electronic Notes in Theoretical Computer Science*. Elsevier Science B. V., Dec 1999.

[DMP96] Olivier Danvy, Karoline Malmkjær, and Jens Palsberg. Eta-expansion does the trick. Technical Report 96/17, BRICS, 1996.

[EJ89] E. Allen Emerson and Charanjit S. Jutla. On simultaneously determinizing and complementing omega-automata. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science, Pacific Grove, California, Jun 5–8, 1989*, pages 333–342. IEEE, 1989.

[FSDF93] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation (PLDI), Albuquerque, New Mexico, June 23-25, 1993*, pages 237–247. ACM, 1993.

[Gie95] Jürgen Giesl. Termination analysis for functional programs using term orderings. In Alan Mycroft, editor, *Proc. 2nd Int'l Static Analysis Symposium (SAS), Glasgow, Scotland*, volume 983 of *Lecture Notes in Computer Science*, pages 154–171. Springer-Verlag, Sep 1995.

[GJ91] Carsten K. Gomard and Neil D. Jones. A partial evaluator for the untyped lambda calculus. *Journal of Functional Programming*, 1(1):21–69, Jan 1991.

[GJ96] 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, Akademgorodok, Novosibirsk, Russia, Jun 25–28, 1996*, volume 1181 of *Lecture Notes in Computer Science*, pages 273–284. Springer, 1996.

[Gle99] Arne J. Glenstrup. Terminator II: Stopping partial evaluation of fully recursive programs. Master's thesis, DIKU, University of Copenhagen, Denmark, 1999.

[Glü91] Robert Glück. Towards multiple self-application. In *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 309–320. ACM, 1991.

[Gom99] Carsten K. Gomard. Higher order partial evaluation– HOPE for the lambda calculus. Master's thesis, DIKU, University of Copenhagen, Denmark, Sep 1999.

[Ham96] G. Hamilton. Higher order deforestation. In H. Kuchen and S. D. Swierstra, editors, *Programming Languages: Implementations, Logics and Programs*, volume 1140 of *Lecture Notes in Computer Science*, pages 213–227. Springer, 1996.

[Hen90] Matthew Hennessy. *The Semantics of Programming Languages*. Wiley, 1990.

[Hen91] Fritz Henglein. Efficient type inference for higer-order binding-time analysis. In *Proc. Conf. Functional Programming Languages and Computer Architecture (FPCA)*, volume 523 of *Lecture Notes in Computer Science*, pages 448–472. Springer, Aug 1991.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM (CACM)*, 12(10):576–580, Oct 1969.

[Hoa78] C. A. R. Hoare. Some properties of predicate transformers. *Journal of the Association of Computing Machinary*, 25(3):461–480, Jul 1978.

[Hol91] Carsten Kehler Holst. Finiteness analysis. In John Hughes, editor, *Functional Programming Languages and Computer Architecture, Cambridge, Massachusetts, August 1991*, volume 523 of *Lecture Notes in Computer Science*, pages 473–495. Springer, 1991.

[JGS93] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.

[JLM96] Jesper Jørgensen, Michael Leuschel, and Bern Martens. Conjunctive partial deduction in pratice. In J. Gallagher, editor, *Proceedings LOPSTR '96, Stockholm, Sweden*, volume 1207 of *Lecture Notes in Computer Science*, pages 59–82. Springer-Verlag, Aug 1996.

[Joh85] Thomas Johnsson. Lambda lifting. In *Proceedings of Conference on Functional Programming Languages and Computer Architecture '85, Nancy, France*, volume 201 of *Lecture Notes in Computer Science*. Springer, 1985.

[Jon88] Neil D. Jones. Automatic program specialization: A re-examination from basic principles. In D. Bjørner, A. P. Ershov, and N. D. Jones, editors, *Partial evaluation and Mixed Computation*, pages 225–282, 1988.

[Jon97] Neil D. Jones. *Computability and Complexity From a Programming Perspective*. Foundations of Computing Series. MIT Press, 1997.

[Jon99] Neil D. Jones. The expressive power of higher-order types. *Journal of Functional Programming*, 1999.

[KU77] John B. Kam and Jeffrey D. Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7:305–317, 1977.

[Lee99] Chin Soon Lee. Partial evaluation of the euclidean algorithm, revisited. *Higher-Order and Symbolic Computation*, 12(2):203–212, Sep 1999.

[Leu96] Michael Leuschel. The ECCE partial deduction system and the DPPD library of benchmarks. In *(http://www.kuleuven.ac.be/~lpai/)*. Unpublished, 1996.

[Leu98] Michael Leuschel. On the power of homeomorphic embedding for online termination. In *Proceedings of the Static Analysis Symposium '98*, volume 1503 of *Lecture Notes in Computer Science*, pages 230–245. Springer, 1998.

[LJBA01] 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, January 2001*. ACM, 2001.

[LS91] J. W. Llyod and J. C. Shepherdson. Partial evaluation in logic programming. *Journal of Logic Programming*, 11:217–242, 1991.

[LS97a] Naomi Lindenstrauss and Yehohua Sagiv. Automatic termination analysis of logic program (with detailed experimental results). In *(http://www.cs.huji.ac.il/~naomil/)*. Unpublished, 1997.

[LS97b] Naomi Lindenstrauss and Yehoshua Sagiv. Automatic termination analysis of Prolog programs. In Lee Naish, editor, *Proceedings of the Fourteenth International Conference on Logic Programming*, pages 64–77, Leuven, Belgium, Jul 1997. MIT Press.

[LSS97] Naomi Lindenstrauss, Yehoshua Sagiv, and Alexander Serebrenik. Termilog: A system for checking termination of queries to logic programs. In Orna Grumberg, editor, *Computer Aided Verification, 9th International Conference, CAV '97, Haifa, Israel, Jun 22–25, 1997*, volume 1254 of *Lecture Notes in Computer Science*, pages 444–447. Springer, 1997.

[McN66] Robert McNaughton. Testing and generating infinite sequences by a finite automaton. *Information and Control*, 9(5):521–530, 1966.

[NNH00] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer, 2000.

[OCM00] Enno Ohlebusch, Claus Claves, and Claude Marche. TALP: A tool for the termination analysis of logic programs. In *11th International Conference on Rewriting Techniques and Applications*, 2000.

[Pap94] Christos H. Papadimitiriou. *Computational Complexity*. Addison-Wesley, 1994.

[Pau96] Lawrence C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 1996.

[Plü90] Lutz Plümer. *Termination Proofs for Logic Programs*, volume 446 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 1990.

[Ram30] Frank P. Ramsey. On a problem of formal logic. In *Proceedings of the London Mathematical Society*, pages 264–286. The London Mathematical Society, Dec 1930.

[Rom90] Sergei A. Romanenko. Arity raiser and its use in program specialization. In N. D. Jones, editor, *ESOP '90, 3rd European Symposium on Programming, Copenhagen, Denmark, May 1990*, volume 432 of *Lecture Notes in Computer Science*, pages 341–360. Springer-Verlag, May 1990.

[Saf88] Shmuel Safra. On the complexity of omega-automata. In *29th Annual Symposium on Foundations of Computer Science*, pages 319–327, White Plains, New York, 1988. IEEE.

[Sag91] Yehoshua Sagiv. A termination test for logic programs. In Vijay Saraswat and Kazunori Ueda, editors, *Logic Programming, Proceedings of the 1991 International Symposium, San Diego, California, USA, Oct 28–Nov 1, 1991*, pages 518–532. MIT Press, 1991.

[Sei96] H. Seidl. Integer constraints to stop deforestation. In *European Symposium on Programming*, volume 1058 of *Lecture Notes in Computer Science*, pages 326–340. Springer, 1996.

[SGJ94] Morten Heine Sørensen, Robert Glück, and Neil D. Jones. Towards unifying partial evaluation, deforestation, supercompilation, and gpc. In *European Symposium on Programming '94*, volume 788 of *Lecture Notes in Computer Science*, pages 483–500. Springer, 1994.

[Sør94] Morten Heine Sørensen. A grammar-based data-flow analysis to stop deforestation. In *Proceedings CAAP '94*, volume 787 of *Lecture Notes in Computer Science*, pages 335–351. Springer, 1994.

[Sør98] Morten Heine Sørensen. Convergence of program transformers in the metric space of trees. In *Proceedings MPC '98*, volume 1422 of *Lecture Notes in Computer Science*, pages 315–337. Springer, 1998.

[SS97] Helmut Seidl and Morten Heine Sørensen. Constraints to stop higher-order deforestation. In *Proceedings of the Symposium on Principles of Programming Languages, 1997*. ACM, 1997.

[SSS97] Chris Speirs, Zoltan Somogyi, and Harald Søndergaard. Termination analysis for Mercury. In Pascal Van Hentenryck, editor, *Static Analysis, Proceedings of the 4th International Symposium, SAS '97, Paris, France, Sep 8–19, 1997*, volume 1302 of *Lecture Notes in Computer Science*, pages 160–171. Springer, 1997.

[SVW87] A. Prasad Sistla, Moshe Y. Vardi, and Pierre Wolper. The complementation problem for Büchi automata with applications to temporal logic. *Theoretical Computer Science*, 49:217–237, 1987.

[Tho96] Simon Thompson. *The Craft of Functional Programming*. Addison-Wesley, 1996.

[TI 87] TI Inc. *TI Scheme Language Reference Manual, Revision B*. Texas Instruments Incorporated, 1987.

[Tou98] Hélène Touzet. Encoding the hydra battle as a rewrite system. In *MFCS 1998*, volume 1450 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.

[Tur86] V. F. Turchin. The concept of a supercompiler. *TOPLAS*, 8(3):292–325, 1986.

[Var96] Moshe Vardi. An automata-theoretic approach to linear temporal logic. In *Logics for Concurrency: Structure versus Automata*, volume 1043 of *Lecture Notes in Computer Science*, pages 238–266. Springer, 1996.

[Wad88] P. L. Wadler. Transforming programs to eliminate trees. In *European Symposium on Programming '88*, volume 300 of *Lecture Notes in Computer Science*, pages 344–358. Springer, 1988.

Index