Terminator II:
Stopping Partial Evaluation of Fully Recursive Programs

Master's Thesis

Arne John Glenstrup

June 13, 1999

Technical Report DIKU-TR-99/8
Department of Computer Science
University of Copenhagen
Universitetsparken 1
DK-2100 Copenhagen Ø
DENMARK
http://www.diku.dk/~panic/TerminatorII/

This report has been typeset by the author using the LaTeX system with the following packages: alltt, amssymb, amstext, anysize, apacite, floatflt, fontenc, index, inputenc, latexsym, listings, mathligs, shortvrb, theorem, varioref and xy.

The main text is set in 10 point Palatino. Sans serif and typewriter fonts are the standard Computer Modern typefaces of the TEX system. Small text is set at 9 points, and footnotes are set at 8 points. Special math symbols, Greek and calligraphic letters are from the standard TEX math typefaces.

Abstract

This paper presents a novel way of detecting termination for a large class of programs written in a functional language. The method includes, but is not restricted to, detection of decreasing values under lexicographic ordering, and can thus prove the termination of Ackermann's function.

Assuming the existence of a well-founded ordering of a value domain we present an algorithm, utilising efficient graph operations, that first detects variables of *bounded variation*, and then uses information about successively decreasing values assigned to some of these variables to establish termination.

We show how an extension of this technique to partial evaluation can be used to ensure that specialisation terminates when all variables are of *bounded static variation* and appropriate specialisation points are inserted. This can pose a tricky problem when the result of nested calls cannot be classified as static due to specialisation points. We solve this in an elegant way by recording sufficient information necessary to undo previous boundedness classifications without recomputing costly graph operations. In contrast to previous methods, our insertion of specialisation points is not based on a general heuristic (like e.g. "insert specialisation points at all dynamic conditionals"), but rather on a safe approximation and an "insert-by-need" strategy. Experiments have shown that the method works well on interpreters, which are of prime interest for partial evaluation.

The algorithm for detecting termination has a worst-case complexity of $O(p^3)$, where p is the program size, but for typical programs it is expected to be at most $O(p^2)$. The insertion of a small set of specialisation points during the binding-time analysis has exponential worst-case complexity, due to the need for considering all combinations of recursive call sites. This indicates that future research should address this problem in more depth.

Preface

This thesis is submitted in partial fulfillment of the requirements for a Danish Master's Degree. The supervisor for the project has been Professor Neil D. Jones at the Institute of Computer Science of the University of Copenhagen (DIKU). Part of the work was done while visitng the Hitachi Advanced Research Laboratory (HARL) in Japan in 1997, although the main body of the text was produced in the period July 1998–March 1999. All work except the prototype implementation has been made by the author, although many ideas and formulations have constantly flowed back and forth between Neil D. Jones, Jesper Jørgensen and the author.

Acknowledgements

I would like to thank Professor *Neil Jones* for devoting a lot of time to supervising this project. His sense of direction in partial evaluation is invaluable, and his indefatigable insistence on expositional clarity is the main reason you may find the following text comprehensible.

Peter Sestoft was the external censor of the project, and I am grateful for his detailed comments which pinpointed several small mistakes and at least one major blunder on my part.

Thanks also go to Professor *Masami Hagiya* at the University of Tokyo and *Akihiko Takano* at Hitachi, Japan, for their hospitality and great help, both in professional and practical matters, during my stay in Japan 1996–1997.

Special thanks to *Jesper Jørgensen* for undertaking the arduous task of implementing the ideas in a prototype, for swift responses on bug reports, and for deep and insightful technical and theoretical discussions.

Thanks to *Peter Holst Andersen* for pointing out when matters that I thought were trivial were not so trivial, and for commenting on drafts of the paper. Also *Manuvir Das* has been kind in answering a lot of technical questions concerning his PhD work.

I greatly appreciate that *Tommy Højfeld Olesen* took time out to cut his finger (!) and rigorously read the introduction. It is to him, *Martin Koch* and *Finn Schiermer Andersen* you should send your kind thoughts if you find the introduction particularly clear and readable.

Although all these people helped me remove errors and obscurities, those that remain are solely my responsibility.

June 1999
Arne John Glenstrup

Contents

Chapter 1

Introduction

In this paper we present an algorithm to detect termination of deterministic programs written in a functional language. The algorithm is itself terminating and sound, and must therefore be conservative, yet it is able to detect a fair class of terminating programs, including, but not restricted to, programs that terminate by virtue of a simple lexicographic ordering of function arguments.

We will assume the reader has a basic knowledge of computer science, and in parts concerning partial evaluation a basic knowledge of this subject (similar to that presented by Jones, Gomard, & Sestoft, 1993) is also assumed.

1.1 Informal introduction to the algorithm

Later, we will formally develop the termination detection analyses, but as an example of how the algorithm works, we will first consider the following

Example 1.1 (Termination of a `LOOP` program interpreter)
Given a small functional language, `LOOP`, with constants, variables, let-expressions, conditionals and function calls, where a program consists of a list of functions taking exactly one argument, including a start function `main`,

we construct an interpreter for this language as shown in Figure 1.1 on the following page, where `cons` and `[]` are list constructors, `car` and `cdr` return the head and tail of a list, `cadar x` is short for `car (cdr (car x))` etc. The recursive calls to `eval` have be split into three classes according to how the arguments increase or decrease:

In $\texttt{eval}^a$ calls, parameter `l` decreases
In $\texttt{eval}^b$ calls, parameter `l` is unchanged; parameters `ns` and `vs` increase
In $\texttt{eval}^c$ calls, parameters `l`, `ns` and `vs` are unchanged

The interpreter, written in a dialect of LISP, is started by calling the function `run` with the program to be interpreted (`p`), a list whose length limits the call depth (`l`)[1] and the interpreted program's input data (`d`). For example, to reverse the list [42,17,17,4,2,42] one can invoke the interpreter by

```
run (main xs = call reverse xs
     reverse xs = if null xs then [ ]
                  else call append (cons (call reverse (cdr xs)) [car xs])
     append xs ys = if null xs then ys
                    else cons (car xs) (call append (cdr xs) ys) )
     [1,1,1,1,1,1] [42,17,17,4,2,42]
```

As the call depth is limited by the length (here 6) of `l`, the interpreted programs compute primitive recursive functions. Furthermore, as the interpreted programs terminate, so does the *interpreter*, and we can detect this automatically by examining the data flow of the interpreter.

First we can compute a "safe approximation" of the size (i.e. the number of cons nodes) of the return values of functions in $\texttt{int}_{\text{LOOP}}$, compared to those of their parameters. Nontermination stems from parameters that take on unbounded values during evaluation, and termination is guaranteed by decreasing parameters, so if we view an expression as a "value transformer" operating on the values of its free variables, we are interested in two kinds of effects it can have on them:

[1] Having a call depth limit is not natural for an interpreter, but for the sake of this example it ensures termination.

```
run p l d = eval (lkbody main p)
                 (list (lkparm main p)) (list d) l p

eval e ns vs l p =
  case e of
    c                  : c
    x                  : lkvar x ns vs
    basefn e_1 e_2     : apply basefn (eval^c e_1 ns vs l p)
                                      (eval^c e_2 ns vs l p)

    let x = e_1 in e_2 : eval^b e_2 (cons x ns)
                           (cons (eval^c e_1 ns vs l p) vs) l p

    if e_1 then e_2 else e_3 : eval^c (if (eval^c e_1 ns vs l p) then e_2
                                  else e_3)
                                 ns vs l p

    call f e_1         : if null l then []
                         else eval^a (lkbody f p)
                                     (list (lkparm f p))
                                     (list (eval^c e_1 ns vs l p))
                                     (cdr l) p

lkparm f p    = if caar p = f then cadar p
                else lkparm f (cdr p)

lkbody f p    = if caar p = f then cddar p
                else lkbody f (cdr p)

lkvar x ns vs = if car ns = x then car vs
                else lkvar x (cdr ns) (cdr vs)
```

Figure 1.1: int$_{\mathsf{LOOP}}$, an interpreter for a small call-depth bounded functional language, LOOP. `e` is the expression, `ns` the list of variable names, `vs` the list of corresponding values, `l` the call depth limit and `p` the program. The `lkparm` and `lkbody` functions look up function parameters and function bodies, and `lkvar` looks up the value of a variable

A non-bounding effect: As constants are lifted directly out of the interpreted expression e, the return value of eval *might* be greater than (i.e. contain more cons nodes than) the value of e, vs or p. Further, the return value of lkparm/lkbody (or lkvar) might be unboundedly large if the value of p (or vs) is unboundedly large.

A decreasing effect: The return value of eval cannot be guaranteed *always* to be less than any of its input variables. However, for lkparm/lkbody (or lkvar), the return value is *always* less than p (or vs).

These safe approximations can be computed automatically and we can represent them in compact form thus ('↓' reads "always less than," '↑' "possibly greater than," and '↥' "related to the size of"):

f	Non-bounding	Decreasing
eval	{↑(e),↑(vs),↑(p)}	{}
lkparm	{↥(p)}	{↓(p)}
lkbody	{↥(p)}	{↓(p)}
lkvar	{↥(vs)}	{↓(vs)}

Note that ↑(x) overrides ↥(x), i.e. ↥(x) is not included in a set containing ↑(x).

Using these descriptions for estimating the values of nested calls we build two graphs describing the increasing and decreasing data flow between function variables, shown in Figures 1.2 and 1.3. Each node represents a function variable, and the edges indicate that the value of one variable can depend on that of another. For instance, there is an edge from eval$_{\mathtt{e}}$ to lkparm$_{\mathtt{f}}$ because the value of f in lkparm depends on the value of e in eval. In the $\mathrm{SDG}^{\uparrow}$, an edge label ↑ indicates that the value of a variable *might sometimes* become greater from one function call to the next, in $\mathrm{SDG}^{\downarrow}$, an edge label ↓ indicates that the value *must always* become smaller. For readability, we have merged all eval calls of class *c* (i.e. calls of the form eval e$_x$ ns vs l p) into one loop, the lower right one on eval nodes. Unlabeled edges correspond roughly speaking to "no change in size."

Bounded variation We now attempt to detect automatically which function variables only take finitely many different values during any run of

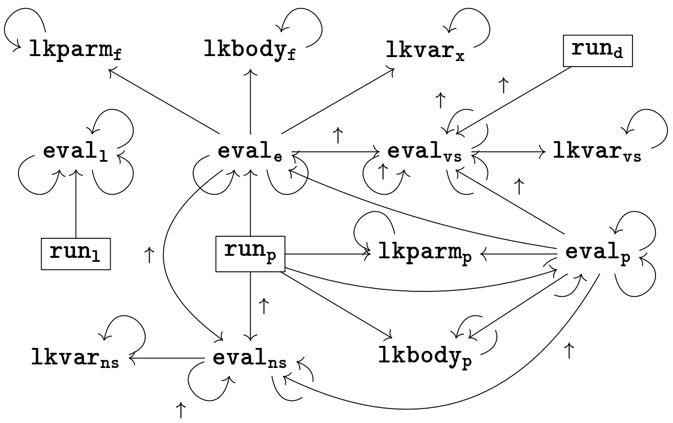


Figure 1.2: Increasing size dependency graph (SDG$^\uparrow$) for `int`$_{\mathsf{LOOP}}$

`int`$_{\mathsf{LOOP}}$—these variables are of *bounded variation* (BV). Clearly, as function `run` is not called from within the interpreter, its variables are all of BV.

Bounded domination For a variable to take unboundedly many different values requires either that it is involved in a loop where its value increases, or it receives values produced in such a loop. We therefore only concern ourselves with the loops (i.e. strongly connected components, SCC) in the SDG$^\uparrow$ graph: for each SCC, if the values that can flow into the SCC are of BV and there are no increasing edges in the SCC, its nodes will be of BV. Using this observation, called *bounded domination*, we can see that `eval`$_\mathtt{l}$, `eval`$_\mathtt{p}$, `eval`$_\mathtt{e}$, `lkparm`$_\mathtt{p}$, `lkbody`$_\mathtt{p}$, `lkparm`$_\mathtt{f}$, `lkbody`$_\mathtt{f}$ and `lkvar`$_\mathtt{x}$ are all BV.

Bounded anchoring The two nodes involved in increasing loops, `eval`$_\mathtt{ns}$ and `eval`$_\mathtt{vs}$, are in fact also BV. To see this, we draw up a *loop dependency graph* (LDG). In Figures 1.2 and 1.3 there are three kinds of loops (from three sets of call sites) for the `eval` nodes. We call them a,b,c (for SDG$^\uparrow$ loops) and

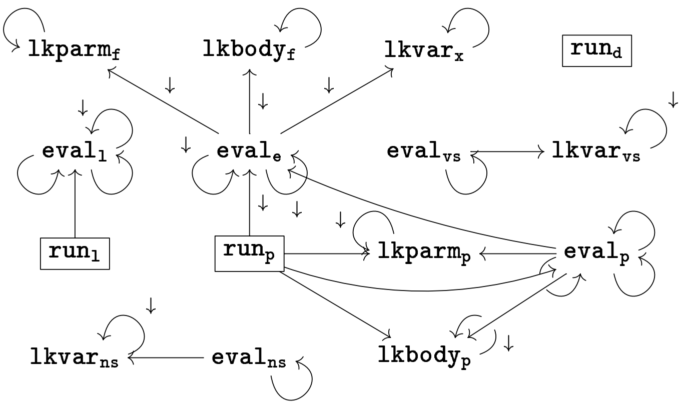
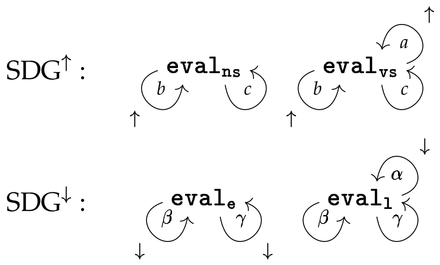
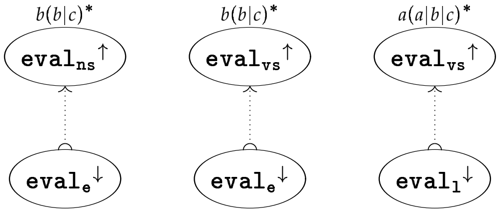

$\texttt{lkparm}_\texttt{f}$
$\texttt{lkbody}_\texttt{f}$
$\texttt{lkvar}_\texttt{x}$
$\texttt{run}_\texttt{d}$
$\texttt{eval}_\texttt{l}$
$\texttt{eval}_\texttt{e}$
$\texttt{eval}_\texttt{vs}$
$\texttt{lkvar}_\texttt{vs}$
$\texttt{run}_\texttt{l}$
$\texttt{run}_\texttt{p}$
$\texttt{lkparm}_\texttt{p}$
$\texttt{eval}_\texttt{p}$
$\texttt{lkvar}_\texttt{ns}$
$\texttt{eval}_\texttt{ns}$
$\texttt{lkbody}_\texttt{p}$

Figure 1.3: Decreasing size dependency graph (SDG$^\downarrow$) for $\texttt{int}_{\textsf{LOOP}}$

their respective *sibling loops*, loops with an identical list of call sites, in SDG$^\downarrow$ are called α, β, γ:


SDG$^\uparrow$:
$\texttt{eval}_\texttt{ns}$
$\texttt{eval}_\texttt{vs}$
a
b
c
SDG$^\downarrow$:
$\texttt{eval}_\texttt{e}$
$\texttt{eval}_\texttt{l}$
α
β
γ

We now consider the possible ways of making loops, using each SDG$^\uparrow$ edge 0, 1 or more times in a loop. For example, an increasing loop only consisting of b and c edges will always run "in parallel" with a decreasing loop consisting of β and γ edges—we say these bc-loops are *anchored* in the corresponding $\beta\gamma$-loops. This is captured in the LDG for $\texttt{int}_{\textsf{LOOP}}$, shown in Figure 1.4. Nodes represent SDG-loops, and edges show the relationship between *increasing* loops in the SDG$^\uparrow$ and *decreasing* loops in the SDG$^\downarrow$. Note



$b(b|c)^*$
$b(b|c)^*$
$a(a|b|c)^*$
eval$_{\texttt{ns}}{}^\uparrow$
eval$_{\texttt{vs}}{}^\uparrow$
eval$_{\texttt{vs}}{}^\uparrow$
eval$_{\texttt{e}}{}^\downarrow$
eval$_{\texttt{e}}{}^\downarrow$
eval$_{\texttt{l}}{}^\downarrow$

Figure 1.4: Loop dependency graph (LDG) for **int**$_{\texttt{LOOP}}$. The regular expressions indicate what kind of loop the nodes represent, i.e. which SDG$^\uparrow$ edges they are composed of (the order of the edges is irrelevant).

that since neither **ns** nor **vs** increase along *c* loops, loop nodes representing only *c* calls are uninteresting and are not included in the LDG.

The rightmost LDG connection represents loops consisting of at least one *a* edge; the increasing variable **eval**$_{\texttt{vs}}$ is anchored in the corresponding decreasing variable **eval**$_{\texttt{l}}$ along the sibling loops (that consist of at least one α edge). The other two connections represent loops consisting of at least one *b* edge and no *a* edges, anchored in corresponding loops consisting of at least one β edge and no α edges. The reason for splitting up the **eval**$_{\texttt{vs}}$ loops according to whether they contain an *a* edge is that they are anchored in loops of two different variables, **eval**$_{\texttt{e}}$ and **eval**$_{\texttt{l}}$.

Note that, given e.g. the loop *abb*, where *every* SDG$^\uparrow$ edge on **eval**$_{\texttt{vs}}$ is marked ↑, it is sufficient that the sibling loop $\alpha\beta\beta$ has *at least one* decreasing edge (in this case α). On the other hand, this must hold for *all* SDG$^\uparrow$ loops (e.g *abb*, *bb*, *bbb*, *bbbb*, ...).

Now there remain only two points to check; all the values that can flow into the **eval**$_{\texttt{ns}}$ and **eval**$_{\texttt{vs}}$ nodes come from BV nodes, and the anchors (**eval**$_{\texttt{e}}$ and **eval**$_{\texttt{l}}$) have already been detected to be BV—then **eval**$_{\texttt{vs}}$ and **eval**$_{\texttt{ns}}$ are BV by what we will call *bounded anchoring*. Summing up, we see that all variables have been detected to be BV.

Result of the analyses. To determine whether the interpreter terminates we now conceptually add an extra parameter, `dp`, to every function definition, changing

```
run p l d        = eval ...
eval e ns vs l p = ...lkvar...
                   ...eval...
lkparm f p       = ...lkparm...
lkbody f p       = ...lkbody...
lkvar f p        = ...lkvar...
```

into

```
run dp p l d        = eval 0...
eval dp e ns vs l p = ...lkvar 0...
                      ...eval (dp + 1)...
lkparm dp f p       = ...lkparm (dp + 1)...
lkbody dp f p       = ...lkbody (dp + 1)...
lkvar dp f p        = ...lkvar (dp + 1)...
```

This extra parameter records the call depth, and in all (mutually) recursive calls, `dp` is incremented. This is related to termination: as we have no iteration constructs, the only source of nontermination is recursive calls, but if all the `dp` parameters can be detected to be BV, the call depth is bounded, and thus the program will terminate.

The subgraph of the SDG$^{\uparrow}$ containing the `dp` nodes is disjoint from the rest and is shown in Figure 1.5. As it contains nothing but increasing edges we cannot use bounded domination, but bounded anchoring can still be applied. The loop dependency graph for these loops is shown in Figure 1.6. As all the anchor nodes have already been shown to be of BV, all the `dp` loops are well anchored and we have thus detected that `int`$_{\mathsf{LOOP}}$ terminates on any input.

Example 1.2 (Termination of a WHILE program interpreter)
Now consider making a new interpreter, `int`$_{\mathsf{WHILE}}$, from `int`$_{\mathsf{LOOP}}$ by replacing the **if**-expression in the last **case**-branch by its **else** branch and removing all `l` parameters:

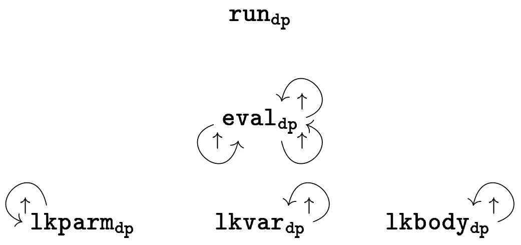


Figure 1.5: Subgraph of SDG$^\uparrow$ for $\texttt{int}_{\mathsf{LOOP}}$ containing `dp`

```
case e of
    ⋮
  call f e_1 : eval^a (lkbody f p)
                      (list (lkparm f p))
                      (list (eval^c e_1 ns vs p))
                      p
```

Then the interpreted programs can compute all partial recursive functions, and $\texttt{int}_{\mathsf{WHILE}}$ may thus fail to terminate for some input.

We are able to detect this automatically: The SDG graphs are as before except for nodes $\texttt{run}_1$ and $\texttt{eval}_1$ being removed. As the α edge no longer exists, $\texttt{eval}_{\texttt{vs}}$-loops in the LDG containing *a* edges are no longer anchored in any decreasing BV loop, so $\texttt{eval}_{\texttt{vs}}$, and consequently $\texttt{lkvar}_{\texttt{vs}}$, are not BV. This can easily be understood, as $\texttt{eval}_{\texttt{vs}}$ is the variable that would contain the value of a diverging parameter in an interpreted function.

When it comes to anchoring the `dp` variables of $\texttt{int}_{\mathsf{WHILE}}$, the SDG$^\uparrow$, shown in Figure 1.7, now contains a loop for $\texttt{eval}_{\texttt{dp}}$ that is not anchored in any decreasing BV loop. This loop represents loops consisting of at least one *a* edge, and as they are not anchored, we cannot guarantee termination of $\texttt{int}_{\mathsf{WHILE}}$.

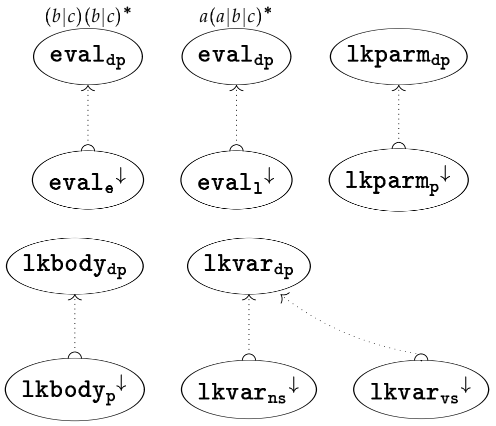


Figure 1.6: Loop dependency graph for `dp` variables added to $\texttt{int}_{\mathsf{LOOP}}$

1.2 Termination of off-line partial evaluation

In partial evaluation we are given a program p and some of its input data s, called the *static data*, and we produce a specialised program (residual program) p_s such that running p_s on the remaining data d, called the *dynamic data*, yields the same output (hopefully faster) as running p on both s and d. A well-known example is specialising the power function `pow n x =` **if** `n = 0` **then** `1` **else** `x * pow (n - 1) x`. Specialising with $\mathtt{n} = 3$ proceeds as follows:

$$\begin{array}{rcl}
\mathtt{pow}_3\;\mathtt{x} & = & \textbf{if}\ 3=0\ \textbf{then}\ \mathtt{1}\ \textbf{else}\ \mathtt{x\ *\ pow}\,(3-1)\,\mathtt{x} \\
 & = & \mathtt{x\ *}\ \textbf{if}\ 2=0\ \textbf{then}\ \mathtt{1}\ \textbf{else}\ \mathtt{x\ *\ pow}\,(2-1)\,\mathtt{x} \\
 & = & \mathtt{x\ *\ x\ *}\ \textbf{if}\ 1=0\ \textbf{then}\ \mathtt{1}\ \textbf{else}\ \mathtt{x\ *\ pow}\,(1-1)\,\mathtt{x} \\
 & = & \mathtt{x\ *\ x\ *\ x\ *}\ \textbf{if}\ 0=0\ \textbf{then}\ \mathtt{1}\ \textbf{else}\ \mathtt{x\ *\ pow}\,(0-1)\,\mathtt{x} \\
 & = & \mathtt{x\ *\ x\ *\ x\ *\ 1}
\end{array}$$

In each step it must be decided what to reduce—these are called the *static* expressions—and what to leave, i.e. generate residual code for, called the *dynamic* expressions. *Off-line* partial evaluation stages the process in two stages:

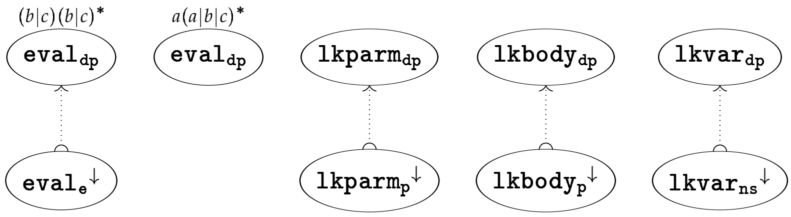

$(b|c)(b|c)^*$
$a(a|b|c)^*$
$\mathtt{eval}_{\mathtt{dp}}$
$\mathtt{eval}_{\mathtt{dp}}$
$\mathtt{lkparm}_{\mathtt{dp}}$
$\mathtt{lkbody}_{\mathtt{dp}}$
$\mathtt{lkvar}_{\mathtt{dp}}$
$\mathtt{eval}_{\mathtt{e}}{}^{\downarrow}$
$\mathtt{lkparm}_{\mathtt{p}}{}^{\downarrow}$
$\mathtt{lkbody}_{\mathtt{p}}{}^{\downarrow}$
$\mathtt{lkvar}_{\mathtt{ns}}{}^{\downarrow}$

Figure 1.7: Loop dependency graph for `dp` variables added to $\mathtt{int}_{\mathsf{WHILE}}$

first a *binding time analysis* (BTA) analyses the program and decides which expressions are to be considered static and which dynamic, and also which calls are to be unfolded. Following this, a *specialisation phase* reduces the program according to the output of the BTA. In `pow`, expressions `n = 0` and `(n - 1)` are static.

It can also be the case that partial evaluation encounters a function call with *the same static arguments* several times during specialisation—think e.g. of an interpreter, interpreting while-loops with a dynamic condition. This does not lead to infinite recursion, however, if the function call is annotated as a *specialisation point*. The specialisation point annotation instructs the specialiser to remember the values of the static arguments and create a specialised, residual version of the function (Jones et al., 1993). Thus, instead of unfolding at specialisation time, code for calling a new specialised, residual function is generated in the residual program. Later calls with the same static arguments will then just generate code for calling this residual function. If a specialisation point was added to the recursive call in `pow`, the specialisation above would yield four residual functions:

$$\begin{array}{l}\mathtt{pow}_3\ \mathtt{x\ =\ x\ *\ pow}_2\ \mathtt{x;}\ \ \mathtt{pow}_2\ \mathtt{x\ =\ x\ *\ pow}_1\ \mathtt{x;}\\ \mathtt{pow}_1\ \mathtt{x\ =\ x\ *\ pow}_0\ \mathtt{x;}\ \ \mathtt{pow}_0\ \mathtt{x\ =\ 1}\end{array}$$

Note that the return value of a function call annotated as a specialisation point is not computed in the specialisation phase, so adding specialisation points can change expressions from static to dynamic.

There is a so-called *congruence* restriction (Jones et al., 1993) on how to choose the binding times (also called the *binding-time division*), namely that static expressions must not require the values of dynamic expressions to be computed, but apart from this restriction there is some freedom of choice. Often one would like as many static expressions as possible, so that only few computations remain in the residual program, but if too many expressions are considered static, the specialisation phase (and thus the entire partial evaluation) might not terminate, due to progressively larger and larger static expressions. This can *not* be avoided by inserting specialisation points, which only shifts the problem to the generation of infinitely many residual functions.

However, if we can guarantee that each static function parameter in the program will only be supplied with *finitely* many *different* argument values during specialisation, that phase can be brought to terminate by inserting specialisation points: If each potentially infinite loop contains a specialisation point, specialisation will stop eventually, because residualising will prevent the unfolding of two function calls with identical arguments. A function parameter with the "finitely many different values" property is said to be of *bounded static variation* (BSV).

Example 1.3 (Termination of partial evaluation of $\texttt{int}_{\texttt{LOOP}}$)
Consider the LOOP interpreter, with `p` and `l` static and `d` dynamic. Using the same SDG$^{\uparrow}$ as before we can ensure the congruence condition is satisfied by starting at $\texttt{run}_{\texttt{d}}$ (the dynamic input to $\texttt{int}_{\texttt{LOOP}}$) and marking all nodes reachable from this node as dynamic. Thus, $\texttt{eval}_{\texttt{vs}}$ and $\texttt{lkvar}_{\texttt{vs}}$ must be dynamic.

Using exactly the same technique as before, i.e. bounded domination and anchoring, we determine the remaining variables to be BSV. The SDG$^{\uparrow}$ for `dp` is like Figure 1.6, except that now it does not contain the rightmost connection to $\texttt{lkvar}_{\texttt{vs}}$ as this variable is not BSV (it is dynamic), but all the loops are still anchored, indicating that partial evaluation will terminate.

A variation: inserting specialisation points. Intuitively, it is obvious that partial evaluation of $\texttt{int}_{\texttt{LOOP}}$ will terminate for static `p` and `l` input: constant propagation and unfolding of a primitive recursive function are limited (among other things) by the length of `l`. Now consider what happens if we want to specialise with **p** static and **l** and **d** dynamic; in this case we find that the same nodes, except $\mathtt{eval}_{\mathtt{l}}$, are BSV by bounded anchoring and domination.

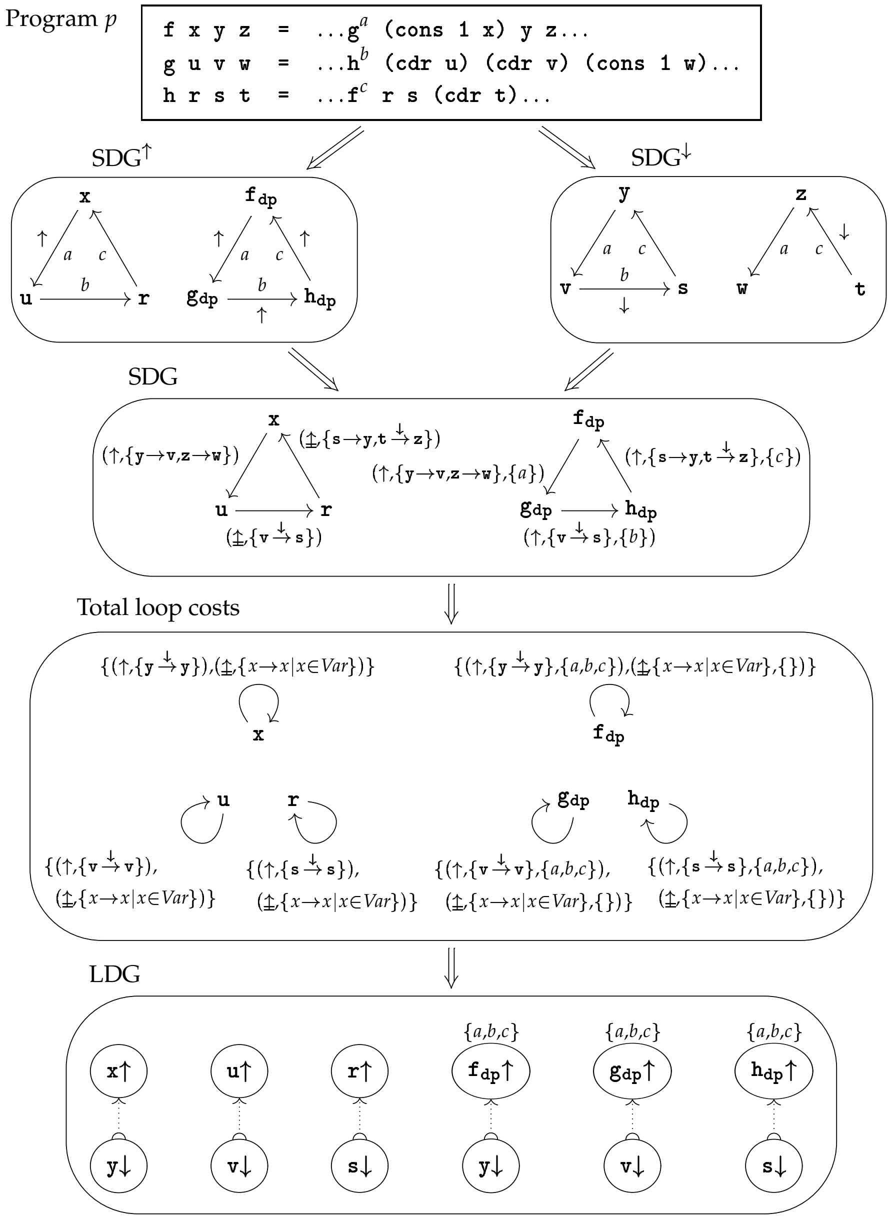
Now the SDG$^{\uparrow}$ for **dp** will resemble that of $\mathtt{int}_{\mathsf{WHILE}}$ (cf. Figure 1.7), where the $\mathtt{eval}_{\mathtt{dp}}$ loops containing an *a* edge are not anchored.

The **dp** loop anchoring observation for guaranteeing termination can be extended to partial evaluation in this way: *If* all **dp** loops that do not contain a specialisation point are anchored in a decreasing BSV loop in the SDG$^{\downarrow}$, *then* specialisation will terminate. For a loop to "contain a specialisation point" means that one of its edges was generated from a call site which is a specialisation point. Conversely, to make specialisation terminate, we must *insert specialisation points* for all non-BSV-anchored loops in the SDG$^{\uparrow}$ for **dp**. In our example we observe that the non-anchored loops all contain the edge *a*, which corresponds to the recursive call to **eval** in the last branch of the **case**-expression; adding a specialisation point here will cause partial evaluation of $\mathtt{int}_{\mathsf{LOOP}}$ (with **p** static and **l** and **d** dynamic) to terminate.

The entire story for partial evaluation termination is slightly more involved, including a dynamic dependency graph, cf. Section 7.2, but the basic principle is as described in the present example. A sketch of how the various graphs are related is shown in Figure 1.8 on the next page; note that we have left out the less interesting parts of the graphs and the program. At the top one can see how the SDG$^{\uparrow}$ and SDG$^{\downarrow}$ graphs conceptually are constructed from the program and then merged into an SDG: The edges of the SDG come from the SDG$^{\uparrow}$ and the second component of the edge labels come from the SDG$^{\downarrow}$. For instance, the SDG edge $\mathtt{u} \xrightarrow{(\underline{\uparrow}, \{\mathtt{v} \xrightarrow{\downarrow} \mathtt{s}\})} \mathtt{r}$ representing the similar edge in SDG$^{\uparrow}$ for call site *b* is labeled with $\{\mathtt{v} \xrightarrow{\downarrow} \mathtt{s}\}$ because $\mathtt{v} \xrightarrow{\downarrow} \mathtt{s}$ is the only edge in SDG$^{\downarrow}$ from call site *b*.

Note that the call sites are only included in the labels of edges between **dp** nodes. The reason for this is that only **dp** nodes are used when determining at which call sites to insert specialisation points.

The total loop costs, i.e. the various kinds of loops, are then computed conceptually by iterating over the SDG, composing the edge labels along all loops. By examining all the total loop costs with '$\uparrow$' in their first component



Program p

```
f x y z  =  ...g^a (cons 1 x) y z...
g u v w  =  ...h^b (cdr u) (cdr v) (cons 1 w)...
h r s t  =  ...f^c r s (cdr t)...
```

SDG↑
x
f_dp
u
r
g_dp
h_dp
a
b
c
SDG↓
y
z
v
s
w
t
SDG
(↑,{y→v,z→w})
(↨,{s→y,t ↓→ z})
(↑,{y→v,z→w},{a})
(↑,{s→y,t ↓→ z},{c})
(↨,{v ↓→ s})
(↑,{v ↓→ s},{b})
Total loop costs
{(↑,{y ↓→ y}),(↨,{x→x|x∈Var})}
{(↑,{y ↓→ y},{a,b,c}),(↨,{x→x|x∈Var},{})}
{(↑,{v ↓→ v}), (↨,{x→x|x∈Var})}
{(↑,{s ↓→ s}), (↨,{x→x|x∈Var})}
{(↑,{v ↓→ v},{a,b,c}), (↨,{x→x|x∈Var},{})}
{(↑,{s ↓→ s},{a,b,c}), (↨,{x→x|x∈Var},{})}
LDG
{a,b,c}
x↑
u↑
r↑
f_dp↑
g_dp↑
h_dp↑
y↓
v↓
s↓

Figure 1.8: Sketch of how the graphs used are related. Letters a, b and c represent call sites, and $Var = \{\texttt{r},\texttt{s},\texttt{t},\texttt{u},\texttt{v},\texttt{w},\texttt{x},\texttt{y},\texttt{z}\}$

we construct the LDG which is used to detect anchoring and specialisation points.

1.3 What is new?

In this paper we present a fully automatic way of detecting termination. It can detect termination of e.g. Ackermann's function and other functions governed by lexicographically ordered decreasing value tuples, but is even more general than this.

We show that similar techniques are applicable both to termination of ordinary evaluation and of partial evaluation, and show the importance of considering *bounded static variation*, *bounded domination* and *bounded anchoring* in partial evaluation.

For termination in partial evaluation we solve the complex problem caused by the combination of nested recursive calls and specialisation point insertion, where the latter can force function parameters to become dynamic. This is solved in the novel way sketched in the previous section by first collecting information about parameter size dependencies, *and then* collecting information about how these dependencies behave and depend on each other along loops in the parameter size dependency graphs.

Using this aggregate information we are able to insert sufficiently many specialisation points for specialisation to be guaranteed to terminate, whilst keeping track of which BSV parameters are affected by the dynamic expressions that are thereby introduced. As an added benefit, we also get a separation of concerns by representing size, dynamic and loop dependencies in separate data structures, thus obtaining, we hope, a more clear understanding of the problem.

The algorithm for detecting termination has a worst-case complexity of $O(p^3)$, where p is the program size, but for typical programs it is expected to be at most $O(p^2)$ and often linear. Inserting specialisation points during the binding-time analysis has exponential worst-case complexity, due to the number of different sets of call sites that constitute one type of `dp`-loop. In the preceding example, $\texttt{eval}_{\texttt{dp}}$-loops were anchored in 2 kinds of loops (cf. Figure 1.6), but when also the call sites are used to distinguish different loops, this example contains 7 kinds of loops, with call sites {`b`},{`c`},{`b`,`c`} (for $\texttt{eval}_\texttt{e}$-anchors) and {a},{a,b},{a,c},{a,b,c} (for $\texttt{eval}_\texttt{1}$-anchors). However, we present an optimisation which will make specialisation point insertion polynomial for typical programs.

The overall structure of the paper is as follows:

- First we give a brief background in Chapter 2 on termination analysis, and outline the focus of this paper.
- For the technical part, we introduce the basics in Chapter 3: the syntax and semantics of the language we use. We use a small-step semantics, which lends itself better to correctness proofs of the conditions that guarantee bounded variation.
- Next, in Chapter 4, we define the notion of a *call path* that is a potential "evaluation trace," as a link between the graph algorithms that perform a kind of symbolic comparison and the evaluation operator that operates on concrete values.
- Then, in Chapter 5, we introduce size dependency based on a size dependency function. By factoring out this part of the analysis we also obtain a more clear understanding of what constitutes a "correct" or "safe" size function. Specifically, a problem present in the work of Andersen and Holst (1996) where **if**-expressions and functions that can return constants are approximated conservatively finds a clear and satisfactory solution.
- Following that, we define the concept of *bounded variation* in Chapter 6 and define the central data structures of the algorithm: the *size* and *loop dependency* graphs. The latter are generated by extracting information collected by an algorithm for computing closed semi-ring values, applied to the size dependency graphs.
- These graphs are then used to detect which parameters can be guaranteed to be of BV: we state the *bounded domination* and *bounded anchoring* conditions that supply these guarantees.
- Next, potentially diverging loops are detected, conceptually by observing the boundedness of an extra parameter dp in every function.

- The extension to partial evaluation then follows in Chapter 7, mainly extending the call loop anchoring by showing how to choose places to add specialisation points, and we tackle the problems arising from the dynamic effects they introduce. This is elegantly solved by combining dynamic dependency with the loop dependency graph and performing simple *dynamic cascading*.
- An overall view of the algorithm for both problems is given in Chapter 8, where we also discuss its correctness and some general properties it enjoys.
- This is followed by a brief presentation in Chapter 9 of the prototype implementation of the analyses, where we show examples of programs that are handled well and some that are treated pessimistically by our analyses. We also discuss the complexity of the analyses, and suggest a small optimisation for specialisation-point insertion.
- Finally, in Section 10.4, we give a short discussion on the current state of the research in this area and round off with a conclusion.

Readers mainly interested in the *algorithms,* will want to concentrate on Sections 5.2, 5.4, 5.5 and Chapters 6 (up to Condition 6.8), 7 and 9. Readers mainly interested in the *correctness,* will want to concentrate on Chapters 3, 4, 5, 6, 7 and Section 8.1.

Chapter 2

Background

This section presents a brief review of the main directions of termination analysis and outline the focus of the present paper. A more technical comparison of related work and our work appears in Chapter 10 on page 138.

This paper follows a line of work on constructing the BTA to ensure termination of partial evaluation, starting with Jones' discussion of *static domination* and of building up static data under dynamic control (Jones, 1988). His concept of static domination is identical to the present *bounded domination*, but values built by increasing operators in loops were always considered dangerous and generalised if they tested on dynamic data. There was, in other words, no concept of *anchoring* one variable in another.

Holst (1991) introduced the notion of *quasitermination*, where the program does not necessarily terminate, but only runs through a finite number of *different* states; this corresponds to all variables being of BV. In fact, it can readily be seen that we have for any program p in our language that

$$\begin{tikzcd}
p \text{ terminates} \arrow[r, Rightarrow] \arrow[d, Leftrightarrow] & p \text{ quasiterminates} \arrow[d, Leftrightarrow] \\
\text{All } \mathtt{dp} \text{ variables are BV} & \text{All ordinary variables are BV}
\end{tikzcd}$$

Further, instead of looking explicitly at dynamic control, he considered *in-situ* parameter growth and decrease in *endotransitions* which are function calls that result in further calls to the same function. The term "in-situ" simply means that the value of a parameter is constructed from previous values of the same parameter.

The concept of *bounded static variation* was first introduced by Jones et al. (1993), and captures precisely the requirements for termination of specialisation.

Holst's quasitermination approach was extended to the higher-order case in joint work with Andersen (1996), where they represent size approximations of higher order values as possibly infinite trees that in turn are approximated by tree grammars. They found that to achieve good results in the higher order case they had to also employ closure and single-threaded analyses. Using these techniques they performed experiments automatically proving that partial evaluation of various higher-order versions of a lambda calculus interpreter terminates.

At the same time, we concentrated on giving an efficient and clearly understandable graph algorithm for the first-order tail recursive case (Glenstrup & Jones, 1996), utilising *bounded static variation*, *bounded domination* and *bounded anchoring*, the latter corresponding to the in-situ condition for termination originally stated by Holst. With this approach we were able to prove termination of partial evaluation of a simple interpreter written in tail recursive style. For the tail recursive case just one size dependency graph was necessary, because the "is constructed from" edge label (↥) always coincided with either an "is never greater than" (↧) or "is always less than" (↓) edge label. This is not the case for a non-tail expression like in

```
f x y = f (g x y)
g u v = if ... then u else v
```

Furthermore, obviously no size approximations of function return values were necessary.

At the basis of all this work—the in-situ condition and the bounded anchoring conditions—lies König's Lemma (König, 1936; Diestel, 1997), which states that a finitely branching graph with no nontrivial infinite paths is finite.

Nielson and Nielson (1996) present a termination analysis for a higher-order functional programming language with algebraic data types, but without mutual recursion. It is based on a type and effect inference system where the effects on the arrows are the termination properties of the functions. Like us, they use the subexpression ordering as the well-founded domain for guaranteeing termination, but they only look for loops with decreasing properties, considering the remaining ones unsafe. The effect of our operational approach of looking for potentially increasing loops and anchoring them, is to some extent achieved by explicitly requesting a lexicographic ordering on the parameters of the functions, but this ordering is not automatically detected in their work.

They do not employ a size dependency analysis, so their termination analysis relies on being able to recognize the recursive decrease syntactically. This implies that if functions are to be detected as terminating, the recursion argument in recursive calls cannot be a nested call. It is conceivable, though, that their analysis could be extended by e.g. a size dependency analysis to overcome the syntactic restriction.

Although their approach is not as operational as ours, and an implementation would not be immediately obvious, the type and effect framework seems well suited for a higher order analysis and makes correctness proofs simpler and more convincing.

Another type inference based approach given by Hughes, Pareto, and Sabry (1996) can also prove the termination of some non-primitve recursive functions, e.g. the `shuffle` function

```
shuffle xs = if xs = [] then
               []
             else
               cons (car xs) (shuffle (reverse (cdr xs)))
```

Of course, this requires an inference of the fact that the size of the result of `reverse` is identical to that of its input. This termination analysis relies on a constraint solver for Presburger formulas.

2.1 Focus

The present paper is the extension of this work to the fully recursive first order case. Our focus is that

- it is important to give a *solid semantic foundation* for the techniques, so that the termination guarantees can be trusted, which also implies that
- *correctness* of the techniques should be proved, or at least convincingly argued for. Further,
- the techniques should be *automatic* and *general*, i.e. be able to handle a large class of programs, as this is a key property of partial evaluation. However,
- the prime examples that should be handled well are *interpreters*, because they are particularly well suited for partial evaluation. Furthermore,
- the algorithms that implement the techniques must be reasonably *efficient* with acceptable complexity so they can be built into automatic partial evaluation tools. Finally,
- we will restrict ourselves to treating a first-order purely functional language with call-by-value evaluation, considering only totally static and totally dynamic data (i.e not partially static data). We will only consider off-line partial evaluation and monovariant binding-time analysis. While extensions of the termination analyses along these dimensions are interesting and important to partial evaluation, we find that they are not vital for an initial understanding and examination of the problem in question.

Note also that we are only considering termination *in theory:* in practise the computer may run out of memory, or the user may run out of patience, before termination.

2.2 Static-infinite computations

A static-infinite computation is a nonterminating loop that never tests on dynamic data. The simplest examples are loops like `f s = f (s + 1)` and **while** `true` **do** `s = s + 1` that are attempting to compute a static value, but also `f s d = cons d (f (s + 1) d)` is a static-infinite computation; it will (attempt to) produce an infinite residual program. It has been argued that static-infinite computations are unlikely to occur in practise as they represent poor programming style (Das, 1998; Jones, 1988). From this viewpoint only a *conditional* termination guarantee, stating that partial evaluation will terminate *if* there are no static-infinite computations, is necessary. In current partial evaluators like Similix (Bondorf, 1990), Schism (Consel, 1993), PGG (Thiemann, 1998), SML-mix (Birkedal & Welinder, 1993), C-Mix (Andersen, 1994) and TEMPO (Consel, Hornof, Noel, & Noye, 1996), it is custom to accept nontermination in the case of static-infinite computations.

We must, however, remember that partial evaluation is over-strict, which can cause evaluation of static configurations that do not occur under normal evaluation, particularly in machine-generated programs. Even in reasonably well-crafted programs this situation might occur; consider the following C program where *direction* is the only dynamic variable:

```
if ( direction == UP )  start  = 1;   else start  = 10;
if ( direction == UP )  stop   = 10;  else stop   = 1;
if ( direction == UP )  step   = 1;   else step   = −1;
for ( i = start ; i != stop ; i = i + step   ) { . . . };
```

The **for** loop will then be specialised for all combinations of *start*, *stop* and *step*, including (among others) a static infinite loop:

```
for ( i  = 10;  i  != 1;  i = i    + 1) { . . . };
```

Another example is specialising a self-interpreter to a program containing loops without tests: if we do not detect this potential loop *in the evaluation of the interpreter* and break it with a specialisation point, we in effect have a compiler that does not terminate on all input!

For these reasons, we argue that the BTA should provide an unconditional termination guarantee. It also turns out that it is fairly straightforward to pinpoint the loops that the analysis considers nonterminating and inform the user why some variables have been generalised. If the user wants more computations performed at specialisation time, a manual guarantee could subsequently be supplied to prevent generalisation.

The drawback of guaranteeing that even static-infinite loops will not occur is that one might conservatively generalise too much in cases where static computations could have been specialised away. Das writes that this can particularly be a problem in real imperative programming languages, because they use complex features like signed integers, pointers and arbitrary control flow that are hard to reason about and can cause aliasing, but does not give any examples of it.

2.3 Static errors due to over-strictness

Due to the over-strictness of partial evaluation—it evaluates both branches of a dynamic **if**-expression—errors may occur during specialisation that will never occur under normal evaluation. Consider the program

```
f ss ds = if length ss > length ds then f (cdr ss) ds else ss
```

if `ds` is dynamic and `ss` = `[1, 2]`, the condition is dynamic and `f` will repeatedly be specialised for `ss` = `[1, 2]`, `[2]` and `[]`. At this point a call to `cdr []` is made, resulting in a static error. Das and Reps (1996) have suggested that one should look into the conditional expressions to determine which static variables *control* the loop. In the case above the BTA would then conclude that as the condition is dynamic, no variable controls the loop, and changes to variables (like `cdr ss`) cause them to be generalised.

However, the condition can contain irrelevant variables and if there are several conditions in the loop, it may not be easy to find the one containing the controlling variable. One example of the former is the lookup function for finding a known function in a program:

```
lookup f prg = if f = car (car prg) then cdr (car prg)
               else lookup f (cdr prg)
```

We therefore propose a different approach: whenever a static error raises an exception during specialisation, it is intercepted at the appropriate place and lifted into the residual program. Thus, when specialising the call `f (cdr []) ds` above, the residual program will not contain a call to a residual function, but rather the expression `cdr []`:

```
f2 ds = if 2 > length ds then f1 ds else [1, 2]
f1 ds = if 1 > length ds then f0 ds else [2]
f0 ds = if 0 > length ds then cdr [] else []
```

This is the strategy used in the Similix system (Bondorf, 1993).

2.4 Conditional C specialisation termination

Das (1998) has attacked the hard problem of ensuring termination of partial evaluation for the C programming language. His aim is to supply a *conditional termination guarantee:* the BTA will guarantee that the specialisation phase will terminate *if* there are no static-infinite computations.

He uses three variants of *program representation graphs* (Ramalingam & Reps, 1989), that reperesent the data and control flow through the program. Each statement is represented by a node, edges represent def-use chains, and control dependence edges are added from the nodes representing conditional expressions of **while** and **if** statements to those of the statements in their bodies.

One advantage of his approach is that he defines the semantics directly on the graphs, which makes a safety proof for the BTA (which operates on these graphs) particularly clear. Thus, for a given program stream, the meaning of each node is, roughly speaking, a stream of the values producible during evaluation, and as the control dependence edges are included, the meaning of a node is compositional in meanings of its predecessors.

For a specific node and all possible streams of dynamic program input, Das considers the set of lists obtainable at this node:

- if the lists are all $\perp$-terminated prefixes of some (possibly infinite) list, the values produced at the node are independent of the dynamic program input. Consequently, the node is either of BSV or a part of a static-infinite computation, and this is termed a *strongly static* node.
- if the lists are all repetitions of a fixed list (e.g. {[1,2,3], [1,2,3,1,2,3], ...}) or a $\perp$-terminated prefix of an infinite list, the values produced are

either BSV or part of a static-infinite computation, and this is termed a *weakly static* node. This accounts for the case where the dynamic input only determines *how many* times a static loop is executed.

- if the lists are all built from a finite number of different elements, or are a $\perp$-terminated prefix of an infinite list, the values produced are either BSV or part of a static-infinite computation, and this is termed a *statically varying* node. Except for static-infinite computations, this corresponds to BSV as defined in this paper.

A strongly static node is weakly static and a weakly static node is statically varying.

Das defines three BTAs that detect these progressively larger classes of static nodes by abstract interpretation of the graph (recall that the semantics is defined by the graph). For strongly static BTA all statements inside a dynamic conditional, even constant assignments, will be considered dynamic. However, in the case of weakly static BTA this influence on constant assignments is ignored.

Furthermore, a node which uses values defined in the branches of an **if** construction will also be influenced by the binding time of the condition expression. In the case of statically varying BTA this influence at the end of **if** statements (but not **while** statements, as they represent loops) is ignored. Thus, in dynamic **while** statements, e.g.

```
while (d != s[i]) {
  if (i > 0) i = i - 1; else error("item not found!");
}
```

BSV (for `i`) in the **while** body cannot be exploited.

All these BTAs are described intra-procedurally (so the only source of nontermination is from **while** loops), and are then extended to interprocedural BTAs by combining the program representation graphs of the procedures via various parameter enter and exit nodes. It is not quite clear how dynamically controlled increasing recursive calls, e.g.

```
f(s, d) {
  if (s < d)
    s = s + 1;
  else
```

```
        return $s$;
    call $f(s, d)$;
}
```

(with s static and d dynamic) are handled in the statically varying BTA to ensure termination.

Das goes on to define a different dependency graph, a loop dependence graph, and a BTA on it. The advantage of this graph is that it can handle C programs with pointers and arbitrary control flow, at the cost of losing the ability to define the program semantics directly by the graph.

The loop dependence graph consists of the same nodes as the other depenence graphs, but now loops in the data flow are detected, and control dependences to nodes in loops are considered to be *loop dependences*. The BTA is then performed by marking dynamic input and propagating these marks along data flow and loop dependence edges.

2.5 Termination of term rewriting systems

Most work on termination analysis has been done in the area of term rewriting systems, probably due to the fact that one easily creates (often using automated methods) rewriting systems of which termination is not immediately obvious. Term rewriting systems are distinguished from functional programs in effect by being a list of *small-step* transitions on terms where all function symbols represent *constructors*. Functional programs, on the other hand, consist basically of a list of *big-step* transitions where function symbols represent *functions*.

This last fact means that one cannot see directly from the syntax of a functional program what the resulting size, under some appropriate norm[1], of an application is. Conversely, due to matching and the fact that each small-step rule of a rewriting system can accomplish nontrivial rewriting, it is hard to reason about the *control flow* of term rewriting systems.

[1] A *norm* in this context is a total function $||\cdot|| : \mathit{Term} \to D$ where D is some well-founded domain. Common norms are the *term-size* norm which loosely speaking measures the size of its argument viewed as a term tree, and the *list-size* norm, which measures the length of its argument viewed as a list, ignoring the sizes of the elements.

However, the small-step property of these systems implies that proving their termination is—in principle—simple: find an appropriate well-founded ordering on terms and prove that each rule of the system decreases the term size under this ordering. Consequently, much research has concentrated on developing useful orderings for various types of term rewriting systems (Steinbach, 1995; Dershowitz, 1987), and methods for automatic synthesis of such orderings (Arts & Giesl, 1997).

2.6 Termination analysis using term orderings

Giesl (1995) has carried over the term ordering approach to the functional world by looking at all the recursive calls of a function and synthesising an ordering. This is done by first creating some constraints from the arguments of recursive calls, and then transforming them to an equivalent set of constraints. This equivalent set of constraints does not refer to functions, only constructors, and can thus be solved using the automatic methods developed for term rewriting systems. If these methods produce a well-founded term ordering, it is proof that the recursive calls of the functional program cannot go on forever, and the program terminates.

This work is an example of the *generate-and-test* paradigm (Winston, 1984) where constraints are recognized, and possibly further constraints are synthesized, so that a theorem prover finally can be applied. Walther (1988) gave an algorithm for this kind of automatic inference of program termination and was able to prove the termination of all the algorithms listed by Boyer and Moore (1979).

As this method is based on the advanced techniques of the term rewriting world, it is possible automatically to prove termination of a large range of programs, including the minimum sort function shown in Figure 2.1 on the next page which is not entirely trivial (Walther, 1988; Giesl, 1995).

On the other hand, this method requires sophisticated constraint solvers to synthesise the right ordering, and is not immediately applicable to termination analysis for partial evaluation, because if no ordering can be synthesised, it does not pinpoint the problematic variables that could be generalised.

```
minsort xs =
  if xs = [] then
    []
  else
    cons (mins xs) (minsort (rm (mins xs) xs))

rm x xs =
  if xs = [] then
    []
  else if x = car xs then
    rm x (cdr xs)
  else
    cons (car xs) (rm x (cdr xs))

mins xs =
  if cdr xs = [] then car xs else min (car xs) (mins (cdr xs))

min x y = if x < y then x else y
```

Figure 2.1: Minimum sort function

2.7 Termination of logic programs

The major difference from functional program termination analysis when analysing logic programs is that one must deal with *unification* and *backtracking*. A logic program is, at least conceptually, computed by searching for a success path in its SLD-tree using some strategy, e.g. Prolog's leftmost-atom rule. Due to backtracking, *existential termination*, i.e. that the program will produce at least one answer, differs from *universal termination*, i.e. that repeated backtracking will terminate. Most termination analyses of logic programs consider universal termination (Speirs, 1997), and one sufficient criteria for universal termination of a program p is that all possible SLD-trees for p are finite. An extra twist for logic programs is that parameters can act as "input" or "output" variables according to whether they are instantiated or not, and the termination properties depend on the instantiation pattern of the initial goal.

Given a program, a goal instantiation pattern and some user-chosen norm, Lindenstrauss and Sagiv (1996, 1997) handle the unification problem by first performing an *instantiation analysis*, identifying which parameters in the program will be instantiated enough that their norms cannot change by further unification.

The sufficiently instantiated parameters are then considered in an analysis starting with approximations of the "single-steps" of the program, assuming all answers are sought, i.e. that all possible backtracking is attempted. The single steps are represented by small graphs with *both* directed edges representing a decrease in norm size *and* undirected edges that represent equal norm sizes. These single steps can be composed, the closure of this composition operation is computed, and any nondecreasing cycles detected are reported as possible nontermination.

They analysed about 90 programs, handling successfully 80% of the examples, including *quicksort*, *minimum sort* and a variant of *mergesort* (3 large programs lead to memory problems). It took typically less than 10 seconds to perform the analysis, but some of the larger programs took 10 minutes to analyse.

Codish and Taboch (1997) also take as starting point a user-chosen norm, but instead of analysing the program directly, they first transform it into a set of *binary unfoldings*. Each rule in the binary unfoldings consists of one goal and only one subgoal in the body. The binary unfoldings are equivalent to the program with respect to termination properties, and as they have only one subgoal, they represent all the "single-steps" of the program. However, the set of binary unfoldings of a program is in general infinite, so by computing an approximation of them, and performing instantiation analysis and cycle detection similar to Lindenstrauss and Sagiv, termination is detected. The key point of Codish and Taboch's work is that the binary unfoldings can be used as a semantic basis because not only do they exhibit the termination properties of the original program, they can also be used to find the evaluation calls and answers of the original program.

In termination analysis for the Mercury language, the instantiation problem is sidestepped, because all rules are explicitly annotated with so-called *mode information*. In fact, although Mercury syntax is based on Prolog syntax, the semantics is more like functional languages: every parameter is either an input or output parameter, so no partially instantiated variables can occur (Somogyi, Henderson, & Conway, 1995).

Speirs, Somogyi, and Søndergaard (1997) perform termination analysis on Mercury programs in two stages: First, a size dependency analysis is computed by extracting linear constraints on the parameters in the program and solving them with a standard LP-solver. The constraints take the form $\sum_i \|x_i^{input}\| + \gamma \geq \sum_j \|x_j^{output}\|$, so functions returning values greater than linear combinations of their input (e.g. a multiplication of the input parameters) and higher-order functions are treated conservatively by setting $\gamma = \infty$. However, experiments with real-world Mercury programs indicated that this was not a serious problem.

The second stage operates on a labelled call graph of the program, detecting recursive calls where no parameter decreases. The result of the first stage is used when arguments to recursive calls contain nested calls.

The termination analysis was run on a large number of examples—including the Mercury compiler itself—obtaining results comparable to those of Lindenstrauss and Sagiv (1997), and on large programs it was significantly faster. This speedup is possibly due to the clear mode information in Mercury, reducing the number of instantiation combinations that must be considered.

The methods for termination analysis of logic programs described above are not directly applicable to partial evaluation because folding means that infinite SLD trees do not lead to infinite specialisation if their leaves only contain finitely many nodes with different static parts, but they could probably be adapted, including also information about which parameters are static and which dynamic.

2.8 Termination of on-line partial evaluation

In *on-line* partial evaluation the binding times of variables are decided *during* the specialisation, not before. The main advantage of this with respect to termination is greater *precision* because the actual specialisation-time values are known; one does not have to resort to approximations of expression sizes. The greater precision can lead to better specialisation while still retaining termination.

Recently, *homeomorphic embedding* has gained much popularity for ensuring termination in on-line partial evaluation (Sørensen & Glück, 1995; Leuschel, 1998; Lafave & Gallagher, 1997). Roughly speaking, one expression is homeomorphically embedded in another if the first can be obtained by striking out parts of the latter. Before unfolding during specialisation, one checks that the parameters do not homeomorphically embed *any* of the previous parameters.

The main drawback is that all these homeomorphic comparisons are costly and require storing all previous parameters, also in cases where off-line partial evaluation would not insert a specialisation point. Another point is that the homeomorphic embedding may generalise variables in cases where off-line analyses would not, producing conservative results. For instance in interpreters, when interpreting a function call, the new expression argument ($\mathtt{eval}_e$) may not be embedded in any of the previous ones.

Note that it is vital to compare with *all* previous arguments (or a representative subset), because $p([\,],[a])$ and $p([a],[\,])$ do not homeomorphically embed each other. In an alternating sequence like $p([\,],[a]) \to p([a],[\,]) \to p([\,],[a]) \to p([a],[\,]) \to \cdots$, comparing with only the immediately preceding argument will not reveal the danger. For this reason, the homeomorphic embedding is not immediately usable in off-line termination analysis (Leuschel, 1998).

It would be interesting to combine the on-line and off-line analyses: first doing an off-line termination analysis to discover which variables definitely are safe, then doing on-line termination analysis using homeomorphic embedding only on the remaining parameters. This would give the best of both worlds: non-conservative results for BSV parameters that sometimes increase, non-conservative results for non-BSV-detectable parameters that do not increase during specialisation, and finally faster on-line specialisation due to fewer homeomorphic embedding comparisons.

Chapter 3

Language

3.1 Syntax and notation

We consider a first-order functional language L with fully recursive expressions, i.e. not limited to tail recursive functions.

$$\begin{array}{rcrcl} \mathit{Program} & \ni & p & ::= & \mathit{f1}\ x_1 \ldots x_m = e_1; \ldots; \mathit{fn}\ x_1 \ldots x_k = e_n \\ \mathit{Expression} & \ni & e & ::= & x \mid c \mid b\, e_1 \ldots e_n \mid f\, e_1 \ldots e_n \\ & & & \mid & \textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3 \end{array}$$

$$b \in \mathit{Basefuncname} \quad c \in \mathit{Constant} \quad x \in \mathit{Varname} \quad f \in \mathit{Funname}$$

In the following text we will use the symbol $=$ to denote mathematical equality and the symbol $\equiv$ to denote syntactical equality. We assume a strict left-to-right evaluation order, thus both L and the metalanguage of the text are intended to be read as strict languages.

We presume the reader is familiar with standard functions like $\mathit{dom}\ f$, the domain of function f, $\mathit{rg}\ f = \{f\ x \mid x \in \mathit{dom}\ f\}$, the range of function f, $\mathit{fv}\ e$, the set of free variables in expression e, $\mathit{freshname}()$, a function returning an unused function name, etc.

To simplify the presentation, we will assume given a program p and use the following notation in the ensuing text: Generally, if p is defined by

$p \equiv \ldots; fi\ x_1 \ldots x_n = e_i; \ldots$ and we have a function name $f \equiv fi$, then e^f denotes e_i and $f_1, \ldots, f_n$ denote $x_1, \ldots, x_n$. Furthermore, given simple specific function definitions $p \equiv \ldots; f\ x\ y = e^f; g\ x\ y = e^g; \ldots$, f_x and f_y will denote f's parameters, and similarly for g. Finally, for values or parameters $x_1, \ldots, x_n$ we let $\vec{x}$ denote $x_1 \ldots x_n$ or $(x_1, \ldots, x_n)$ or $[x_1, \ldots, x_n]$; the choice in each case will be clear from the context.

Note that unless otherwise stated, all (in)equalities only hold when both the left hand side and the right hand side are defined.

3.2 Semantics

We now define the semantics of our small language, i.e. how programs are to be interpreted to yield meanings. In our case we take the meaning of a program to be a value that is determined by evaluating the program in a recursive manner.

Values $v \in \mathit{Value}$ are first order, and we assume some well-founded order $\leq$ on this domain, e.g. the subexpression-ordering if *Value* is the set of LISP S-expressions. Sometimes, for the sake of readability, we will use integers, '$+$' and '$-$' in the examples as if it was a well-founded domain. A value is either some program input, (the value of) a constant c or the result of applying a base function. We assume base functions to be deterministic and terminating, possibly with a special error value, on all input values.

An *environment* $E \in \mathit{Environment} = \mathit{Varname} \rightharpoonup \mathit{Value}$ is a function on a finite domain, and the notation $\{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\}$ is used as usual to denote environments. We will express environment extension using the operator $+ : \mathit{Environment} \times \mathit{Environment} \rightharpoonup \mathit{Environment}$ defined by $(E_1 + E_2)\ x =$ if $x \in \mathit{dom}\ \ E_2$ then $E_2\ x$ else $E_1\ x$

Our evaluation model will be defined by a small-step semantics using an evaluation operator $\mathcal{E}$, apply operator $\mathcal{A}$ and call operator $\mathcal{C}$. As it is a small-step semantics, we will need some notation for indicating partially evaluated expressions, called *evaluation contexts*, and we will also need a stack for keeping track of pending function calls:

$$\begin{array}{rcl}
\mathit{Environment} & = & \mathit{Varname} \rightarrow \mathit{Value} \\
\mathit{Stack} & = & (\mathit{Context} \times \mathit{Environment})\ \mathit{List} \\
\mathit{Context} & \ni & \mathit{ve} ::= b\, v_1 \ldots v_m \bullet e_1 \ldots e_n \mid f\, v_1 \ldots v_m \bullet e_1 \ldots e_n \\
 & & \quad \mid \ \mathbf{if} \bullet \mathbf{then}\ e_2\ \mathbf{else}\ e_3
\end{array}$$

The operators $\mathcal{E},\mathcal{A},\mathcal{C}$ are defined in Figure 3.1; the meaning of a program is the value of the first function *f1*. Later, it will be convenient wlog. to assume that the program contains no calls to this function.

$$\begin{array}{lcl}
\mathcal{P} & : & \mathit{Program} \rightarrow \mathit{Value\ List} \rightarrow \mathit{Value} \\
\mathcal{A} & : & \mathit{Stack} \rightarrow \mathit{Value} \rightarrow \mathit{Value} \\
\mathcal{C} & : & \mathit{Funname} \rightarrow \mathit{Value\ List} \rightarrow \mathit{Stack} \rightarrow \mathit{Value} \\
\mathcal{E} & : & \mathit{Expression} \rightarrow \mathit{Environment} \rightarrow \mathit{Stack} \rightarrow \mathit{Value}
\end{array}$$

$$\mathcal{P}\ [\![f1\ x_1 \ldots x_n = e_1; \ldots]\!]\ [v_1,\ldots,v_n] \quad \rightarrow \quad \mathcal{C}\ [\![f1]\!]\ [v_1,\ldots,v_n]\ []$$

$$\begin{array}{lll}
(1) & \mathcal{E}\ [\![c]\!]\ \varrho\ s & \rightarrow \mathcal{A}\ s\ (\mathit{value}\ c) \\
(2) & \mathcal{E}\ [\![x]\!]\ \varrho\ s & \rightarrow \mathcal{A}\ s\ (\varrho\ x) \\
(3) & \mathcal{E}\ [\![\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3]\!]\ \varrho\ s & \rightarrow \mathcal{E}\ [\![e_1]\!]\ \varrho\ (\langle \mathbf{if} \bullet \mathbf{then}\ e_2 \\
 & & \qquad\qquad\qquad \mathbf{else}\ e_3, \varrho\rangle : s) \\
(4) & \mathcal{E}\ [\![b\ e_1\ e_2 \ldots e_n]\!]\ \varrho\ s & \rightarrow \mathcal{E}\ [\![e_1]\!]\ \varrho\ (\langle b \bullet e_2 \ldots e_n, \varrho\rangle : s) \\
(5) & \mathcal{E}\ [\![f\ e_1\ e_2 \ldots e_n]\!]\ \varrho\ s & \rightarrow \mathcal{E}\ [\![e_1]\!]\ \varrho\ (\langle f \bullet e_2 \ldots e_n, \varrho\rangle : s) \\
(0') & \mathcal{A}\ s\ \mathsf{error} & \rightarrow \mathsf{error} \\
(1') & \mathcal{A}\ []\ v & \rightarrow v \\
(3') & \mathcal{A}\ (\langle \mathbf{if} \bullet \mathbf{then}\ e_2\ \mathbf{else}\ e_3, \varrho\rangle : s)\ v & \rightarrow \text{if } v \equiv \mathsf{true} \text{ then } \mathcal{E}\ [\![e_2]\!]\ \varrho\ s \\
 & & \qquad\qquad \text{else } \mathcal{E}\ [\![e_3]\!]\ \varrho\ s \\
(4') & \mathcal{A}\ (\langle b\, v_1 \ldots v_m \bullet e_2 \ldots e_n, \varrho\rangle : s)\ v & \rightarrow \mathcal{E}\ [\![e_2]\!]\ \varrho\ (\langle b\, v_1 \ldots v_m\, v \bullet e_3 \ldots e_n, \varrho\rangle : s) \\
(4'') & \mathcal{A}\ (\langle b\ v_1 \ldots v_m \bullet, \varrho\rangle : s)\ v & \rightarrow \mathcal{A}\ s\ (\mathit{apply}\ b\ [v_1,\ldots,v_m,v]) \\
(5') & \mathcal{A}\ (\langle f\, v_1 \ldots v_m \bullet e_2 \ldots e_n, \varrho\rangle : s)\ v & \rightarrow \mathcal{E}\ [\![e_2]\!]\ \varrho\ (\langle f\, v_1 \ldots v_m\, v \bullet e_3 \ldots e_n, \varrho\rangle : s) \\
(5'') & \mathcal{A}\ (\langle f\ v_1 \ldots v_m \bullet, \varrho\rangle : s)\ v & \rightarrow \mathcal{C}\ [\![f]\!]\ [v_1,\ldots,v_m,v]\ s \\
(5''') & \mathcal{C}\ [\![f]\!]\ [v_1,\ldots,v_n]\ s & \rightarrow \mathcal{E}\ [\![e^f]\!]\ \{f_1 \mapsto v_1,\ldots,f_n \mapsto v_n\}\ s
\end{array}$$

Figure 3.1: Rewrite rules defining the small-step semantics of language L

The symbol • is a special symbol distinct from all other objects, and the function *apply* : *Basefuncname* → *Value List* → *Value* applies the base function b to values $v_1, \ldots, v_n$. We assume that p is well-formed, i.e. that calls in p call functions defined in p with the correct number of arguments and that parameter references are in the lexical scope of their definitions. It is straightforward to check that under these conditions, if *f1* takes n arguments then $\mathcal{P}$, $\mathcal{E}$, $\mathcal{A}$ and $\mathcal{C}$ are well-defined when $\mathcal{P}\ p$ is applied to a list of n values.

In these definitions we have captured all the recursive nature of program evaluation in the stack s: in the equations each left hand side can call at most one of the operators $\mathcal{E}$, $\mathcal{A}$ or $\mathcal{C}$. Thus evaluation of an expression e is a linear sequence of steps:

$$\mathcal{E}\ [\![e]\!]\ \varrho\, s \to \cdots \to \mathcal{A}\ s'\ v' \to \cdots$$

$$\to \mathcal{E}\ [\![e'']\!]\ \varrho''\ s'' \to \cdots \to \mathcal{C}\ [\![f]\!]\ [v_1, \ldots, v_n]\ s''' \to \cdots,$$

As every expression, evaluation context and value is matched by one of the left hand sides of Figure 3.1 on the facing page, it can be seen that evaluation cannot "get stuck":

Lemma 3.1
For $e \in$ Expression and $\varrho \in$ Environment with $fv\ e \subseteq$ dom Environment, if rewriting of $\mathcal{E}\ [\![e]\!]\ \varrho\ [\,]$ according to Figure 3.1 on the preceding page terminates, it terminates with a value $v \in$ Value.

We define the *length* of the sequence to be the number of operators occurring in it, and say it is X-operator-free if operator X does not occur in it.

The notion of sequence length naturally leads to

Definition 3.2 (Program evaluation termination)
Given program p and input $\vec{v}$, we say that $\mathcal{P}\ [\![p]\!]\ \vec{v}$ terminates *iff the length of the sequence $\mathcal{P}\ [\![p]\!]\ \vec{v} \to \cdots$ is finite.*

Definition 3.3 (Program termination)
Given program p, we say p terminates *iff $\mathcal{P}\ [\![p]\!]\ \vec{v}$ terminates for all input $\vec{v} \in$ Value^n.*

So if p does not terminate, the sequence can in general be infinite, but the following lemma shows that this requires the call operator $\mathcal{C}$ to occur infinitely often in the sequence:

Lemma 3.4 (Bounded call-free evaluation depth)
For any expression e and environment ϱ with $fv\ e \subseteq dom\ \varrho$, the length of the longest $\mathcal{C}$-operator-free prefix of steps $\mathcal{E}\ [\![e]\!]\ \varrho\ [\,] \to \cdots$ is bounded by a function of e, $b(e)$.

Proof: Define $|\bullet| : (Context + Stack) \to \mathbb{N}$ as

$$|e| = \begin{cases} 3, & \text{if } e \equiv x \text{ or } e \equiv c \\ 1 + n + |e_1| + \cdots + |e_n|, & \text{if } e \equiv b\ e_1 \ldots e_n \text{ or } e \equiv f\ e_1 \ldots e_n \\ n + |e_1| + \cdots + |e_n|, & \text{if } e \equiv b\ v_1 \ldots v_m\ \bullet\ e_1 \ldots e_n \\ n + |e_1| + \cdots + |e_n|, & \text{if } e \equiv f\ v_1 \ldots v_m\ \bullet\ e_1 \ldots e_n \\ 2 + |e_1| + |e_2| + |e_3|, & \text{if } e \equiv \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 \\ |e_2| + |e_3|, & \text{if } e \equiv \textbf{if } \bullet \textbf{ then } e_2 \textbf{ else } e_3 \end{cases}$$

$$|s| = \begin{cases} 0, & \text{if } s \equiv [\,] \\ 1 + |e| + |s'|, & \text{if } s \equiv \langle e, \varrho \rangle : s' \end{cases}$$

In every step, replace $\mathcal{E}\ [\![e]\!]\ \varrho\ s$ with $|s| + |e|$ and $\mathcal{A}\ s\ v$ with $|s|$. By checking for rules 1–5 and 3′–5′ this can be seen to be a strictly decreasing sequence on $\mathbb{N}$, so we can take $b(e) = |e|$ □

For fixed program p, the number of different expressions that occur in the sequence is bounded (they are always subexpressions of p), and thus there exists a K such that $|e| < K$ for all e occurring in the sequence, so we immediately get the following

Corollary 3.5 (Linear compression)
Define the state transition sequence for p on input $\vec{v}$ to be

$$\mathcal{P}\ [\![p]\!]\ \vec{v} \to \mathcal{C}\ [\![f1]\!]\ \vec{v}_1\ s_1 \to \mathcal{C}\ [\![f^2]\!]\ \vec{v}_2\ s_2 \to \cdots \qquad (\star)$$

when

$$\mathcal{P}\ [\![p]\!]\ \vec{v} \to \mathcal{C}\ [\![f1]\!]\ \vec{v}_1\ s_1 \to \mathcal{E}\ [\![e_1]\!]\ \varrho_1\ s_1 \to^* \mathcal{A}\ s_2\ v \to \mathcal{C}\ [\![f^2]\!]\ \vec{v}_2\ s_2 \to \cdots \qquad (\triangle)$$

where '$\to^*$' in $(\triangle)$ are $\mathcal{C}$-operator-free sequences. Now there exists a $K \in \mathbb{N}$ such that for any $\vec{v}$ the length of any prefix of $(\triangle)$ will be bounded by K times the length of the corresponding prefix of $(\star)$.

This allows us to handle the semantics by reasoning on state transition sequences: their lengths are in some sense proportional to the lengths of the evaluations, and they are finite exactly when the evaluations terminate. As we will not be concerned with the details of the stack, we abbreviate state transition sequences $\mathcal{C}\ [\![f^1]\!]\ \vec{v}_1\ s_1 \to \mathcal{C}\ [\![f^2]\!]\ \vec{v}_2\ s_2 \to \cdots$ simply to $(f^1, \vec{v}_1) \to (f^2, \vec{v}_2) \to \cdots$ in the following text.

We can now express Holst's (1991) definition of quasitermination simply:

Definition 3.6 (Program evaluation quasitermination)
Given p and input $\vec{v}$ of length $arity\ f1$, we say that $\mathcal{P}\ [\![p]\!]\ \vec{v}$ quasiterminates *iff $\{(f, \vec{u}) \mid \mathcal{P}\ [\![p]\!]\ \vec{v} \to (f1, \vec{v}_1) \to^* (f, \vec{u})\}$ is finite*

Definition 3.7 (Program quasitermination)
Given program p, we say p quasiterminates *iff $\mathcal{P}\ [\![p]\!]\ \vec{v}$ quasiterminates for all input $\vec{v} \in Value^n$, where $n = arity\ f1$.*

The key property used throughout this paper to detect termination and quasitermination is

Definition 3.8 (Bounded variation)

$$\boxed{f_i \text{ is BV iff } \forall \vec{u} \in Value^a : \{v_i \mid (f1, \vec{u}) \to^* (f, \vec{v})\} \text{ is finite,}}$$

where $a = arity\ f1$.

It is obvious from the preceding definitions that we have

Corollary 3.9
Given program p, p quasiterminates *if all function parameters in p are BV.*

If we substitute expressions $\vec{e}$ for the free variables $\vec{f}'$ of an expression e', evaluation should produce the same result v' as if we had evaluated e' using an environment mapping the free variables to the expression values $\vec{v}$. This is captured by

Lemma 3.10 (Substitution lemma)
Given $e_1,\ldots,e_m$ *and* ϱ*, assume* $v_1,\ldots,v_m$ *exist such that* $\forall s \forall i \in \{1,\ldots,m\}$: $\mathcal{E}\ [\![e_i]\!]\ \varrho\ s \rightarrow^* \mathcal{A}\ s\ v_i$*. Then we have for all* $e', \vec{f'}$ *that a* v' *exists such that*

$$\begin{aligned}
&\forall s' : \mathcal{E}\ [\![e']\!]\ \{f'_1 \mapsto v_1,\ldots,f'_m \mapsto v_m\}\ s' \rightarrow^* \mathcal{A}\ s'\ v' \text{ if and only if}\\
&\forall s'' : \mathcal{E}\ [\![[f'_1 \mapsto e_1,\ldots,f'_m \mapsto e_m]e']\!]\ \varrho\ s'' \rightarrow^* \mathcal{A}\ s''\ v'
\end{aligned}$$

Proof: by induction on the length of the computation of e'.

Case length = 2:

Subcase $e' \equiv c$: OK.

Subcase $e' \equiv f'_i$: We have $\mathcal{E}\ [\![e']\!]\ \{f'_1 \mapsto v_1,\ldots,f'_m \mapsto v_m\}\ s' \rightarrow \mathcal{A}\ s'\ v_i$ and $\mathcal{E}\ [\![[f'_1 \mapsto e_1,\ldots,f'_m \mapsto e_m]e']\!]\ \varrho\ s'' = \mathcal{E}\ [\![e_i]\!]\ \varrho\ s'' \rightarrow^* \mathcal{A}\ s''\ v_i$

Case length > 2: Let $S = [f'_1 \mapsto e_1,\ldots,f'_m \mapsto e_m]$ and $\varrho' = \{f'_1 \mapsto v_1,\ldots,f'_m \mapsto v_m\}$.

Subcase $e' \equiv$ if e_1 then e_2 else e_3: Wlog we assume e_1 evaluates to true, and then for some v_1, v_2 each step of

$$\begin{aligned}
\mathcal{E}\ [\![e']\!]\ \varrho'\ s' \quad &= \quad \mathcal{E}\ [\![\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3]\!]\ \varrho'\ s'\\
&= \quad \mathcal{E}\ [\![e_1]\!]\ \varrho'\ (\langle \textbf{if} \bullet \textbf{then } e_2 \textbf{ else } e_3, \varrho'\rangle : s')\\
&\rightarrow^* \quad \mathcal{A}\ (\langle \textbf{if} \bullet \textbf{then } e_2 \textbf{ else } e_3, \varrho'\rangle : s')\ \mathsf{true}\\
&\rightarrow \quad \mathcal{E}\ [\![e_2]\!]\ \varrho'\ s'\\
&\rightarrow^* \quad \mathcal{A}\ s'\ v_2
\end{aligned}$$

holds iff the corresponding step of the following holds:

$$\begin{aligned}
\mathcal{E}\ [\![S\ e']\!]\ \varrho\ s'' \quad &= \quad \mathcal{E}\ [\![\textbf{if } S\ e_1 \textbf{ then } S\ e_2 \textbf{ else } S\ e_3]\!]\ \varrho\ s''\\
&= \quad \mathcal{E}\ [\![S\ e_1]\!]\ \varrho\ (\langle \textbf{if} \bullet \textbf{then } S\ e_2 \textbf{ else } S\ e_3, \varrho\rangle : s'')\\
\text{(hyp.)} \quad &\rightarrow^* \quad \mathcal{A}\ (\langle \textbf{if} \bullet \textbf{then } S\ e_2 \textbf{ else } S\ e_3, \varrho\rangle : s'')\ \mathsf{true}\\
&\rightarrow \quad \mathcal{E}\ [\![S\ e_2]\!]\ \varrho\ s''\\
\text{(hyp.)} \quad &\rightarrow^* \quad \mathcal{A}\ s''\ v_2
\end{aligned}$$

Subcase $e' \equiv b\ e'_1 \ldots e'_k$: By letting $v' = \mathit{apply}\ b\ v'_1 \ldots v'_k$ we find for some $\vec{v}'$ that each step of

$$
\begin{array}{rcl}
\mathcal{E}\ \llbracket e' \rrbracket\ \varrho'\ s' & = & \mathcal{E}\ \llbracket b\ e'_1 \ldots e'_k \rrbracket\ \varrho'\ s' \\
 & \to & \mathcal{E}\ \llbracket e_1 \rrbracket\ \varrho'\ (\langle b \bullet e_2 \ldots e_k, \varrho' \rangle : s') \\
 & \to^* & \mathcal{A}\ (\langle b\ \bullet\ e_2 \ldots e_k, \varrho' \rangle : s')\ v'_1 \\
 & \vdots & (k \text{ times}) \\
 & \to & \mathcal{E}\ \llbracket e_k \rrbracket\ \varrho'\ (\langle b\, v'_1 \ldots v'_{k-1} \bullet, \varrho' \rangle : s') \\
 & \to^* & \mathcal{A}\ (\langle b\ v'_1 \ldots v'_{k-1} \bullet, \varrho' \rangle : s')\ v'_k \\
 & \to & \mathcal{A}\ s'\ v'
\end{array}
$$

holds iff the corresponding step of the following holds:

$$
\begin{array}{lrcl}
 & \mathcal{E}\ \llbracket S\ e' \rrbracket\ \varrho\ s'' & = & \mathcal{E}\ \llbracket b\ (S\ e'_1) \ldots (S\ e'_k) \rrbracket\ \varrho\ s'' \\
 & & \to & \mathcal{E}\ \llbracket S e_1 \rrbracket\ \varrho\ (\langle b \bullet (S e_2) \ldots (S e_k), \varrho \rangle : s'') \\
\text{(hyp.)} & & \to^* & \mathcal{A}\ (\langle b\ \bullet\ (S\ e_2) \ldots (S\ e_k), \varrho \rangle : s'')\ v'_1 \\
 & & \vdots & (k \text{ times}) \\
 & & \to & \mathcal{E}\ \llbracket S\ e_k \rrbracket\ \varrho\ (\langle b\ v'_1 \ldots v'_{k-1} \bullet, \varrho \rangle : s'') \\
\text{(hyp.)} & & \to^* & \mathcal{A}\ (\langle b\ v'_1 \ldots v'_{k-1} \bullet, \varrho \rangle : s'')\ v'_k \\
 & & \to & \mathcal{A}\ s''\ v'
\end{array}
$$

Subcase $e' \equiv f\ e_1 \ldots e_k$: By letting v' be the value of $f\ e_1 \ldots e_k$ we find for some $\vec{v}'$ that each step of

$$
\begin{array}{rcl}
\mathcal{E}\ \llbracket e' \rrbracket\ \varrho'\ s' & = & \mathcal{E}\ \llbracket f\ e'_1 \ldots e'_k \rrbracket\ \varrho'\ s' \\
 & \to & \mathcal{E}\ \llbracket e_1 \rrbracket\ \varrho'\ (\langle f\ \bullet\ e_2 \ldots e_k, \varrho' \rangle : s') \\
 & \to^* & \mathcal{A}\ (\langle f\ \bullet\ e_2 \ldots e_k, \varrho' \rangle : s')\ v'_1 \\
 & \vdots & (k \text{ times}) \\
 & \to & \mathcal{E}\ \llbracket e_k \rrbracket\ \varrho'\ (\langle f\, v'_1 \ldots v'_{k-1} \bullet, \varrho' \rangle : s') \\
 & \to^* & \mathcal{A}\ (\langle f\ v'_1 \ldots v'_{k-1} \bullet, \varrho' \rangle : s')\ v'_k \\
 & \to & \mathcal{C}\ \llbracket f \rrbracket\ [v'_1, \ldots, v'_k]\ s' \\
 & \to & \mathcal{E}\ \llbracket e^f \rrbracket\ \{f_1 \mapsto v'_1, \ldots, f_n \mapsto v'_k\}\ s' \\
 & \to^* & \mathcal{A}\ s'\ v'
\end{array}
$$

holds iff the corresponding step of the following holds:

$$\begin{array}{lll}
\mathcal{E}\llbracket S\, e' \rrbracket\, \varrho\, s'' & = & \mathcal{E}\llbracket f\,(S\, e'_1)\ldots(S\, e'_k) \rrbracket\, \varrho\, s'' \\
 & \to & \mathcal{E}\llbracket S\, e_1 \rrbracket\, \varrho\,(\langle f \bullet (S e_2)\ldots(S e_k), \varrho\rangle : s'') \\
\text{(hyp.)} & \to^* & \mathcal{A}(\langle f \bullet (S e_2)\ldots(S e_k), \varrho\rangle : s'')\, v'_1 \\
 & \vdots & (k \text{ times}) \\
 & \to & \mathcal{E}\llbracket S e_k \rrbracket\, \varrho\,(\langle f v'_1 \ldots v'_{k-1} \bullet, \varrho\rangle : s'') \\
\text{(hyp.)} & \to^* & \mathcal{A}\,(\langle f\, v'_1 \ldots v'_{k-1} \bullet, \varrho\rangle : s'')\, v'_k \\
 & \to & \mathcal{C}\,\llbracket f \rrbracket\,[v'_1, \ldots, v'_k]\, s'' \\
 & \to & \mathcal{E}\,\llbracket e^f \rrbracket\,\{f_1 \mapsto v'_1, \ldots, f_n \mapsto v'_k\}\, s'' \\
 & \to^* & \mathcal{A}\, s''\, v'
\end{array}$$

□

3.3 Big-step semantics

The small-step semantics just defined is well-suited for reasoning about termination properties, as it flattens the recursive call tree into a linear structure. However, for some proofs performed by induction it is more convenient to use the following recursive definition of expression evaluation, based only on one operator $\mathcal{E} : \mathit{Expression} \to \mathit{Environment} \to \mathit{Value} \cup \{\bot\}$:

$$\begin{array}{ll}
\mathcal{E}\,\llbracket c \rrbracket\, \varrho & = \mathit{value}\ c \\
\mathcal{E}\,\llbracket x \rrbracket\, \varrho & = \varrho\, x \\
\mathcal{E}\,\llbracket \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 \rrbracket\, \varrho & = \text{if } \mathcal{E}\,\llbracket e_1 \rrbracket\, \varrho \text{ then } \mathcal{E}\,\llbracket e_1 \rrbracket\, \varrho \text{ else } \mathcal{E}\,\llbracket e_3 \rrbracket\, \varrho \\
\mathcal{E}\,\llbracket b\, e_1 \ldots e_n \rrbracket\, \varrho & = \mathit{apply}\ b\ [\mathcal{E}\,\llbracket e_1 \rrbracket\, \varrho, \ldots, \mathcal{E}\,\llbracket e_n \rrbracket\, \varrho] \\
\mathcal{E}\,\llbracket f\, e_1 \ldots e_n \rrbracket\, \varrho & = \mathcal{E}\,\llbracket e^f \rrbracket\,\{f_1 \mapsto \mathcal{E}\,\llbracket e_1 \rrbracket\, \varrho, \ldots, f_n \mapsto \mathcal{E}\,\llbracket e_n \rrbracket\, \varrho\}
\end{array}$$

The two definitions of $\mathcal{E}$ are equivalent, i.e. $\mathcal{E}_{\text{small-step}}\llbracket e \rrbracket\ \varrho\ [\,] = \mathcal{E}_{\text{big-step}}\llbracket e \rrbracket\ \varrho$ for all e, ϱ with $\mathit{fv}\ e \subseteq \mathit{dom}\ \varrho$, and the small-step evaluation terminates exactly when the big-step semantics terminates. Proofs of equivalence of big-step and small-step semantics can be found in the literature (Henglein & Tofte, 1993).

Chapter 4

Call Paths

As we are trying to determine termination properties and this in general is undecidable, we must introduce some approximation into the program analysis. To capture the approximate behaviour of the program we introduce the *call path* which represents a potential call tree of the program in a linear fashion. An example of a program and call tree fragment is

```
f x y  =  if x > 0 then g (x-1) else g (h (y-1))
g u    =  if u > 3 then f (u-2) (u-3) else u
h s    =  if s < 42 then h (s+2) else s
```

$$\begin{tikzcd}
 & & & \texttt{h (y-1)} \arrow[r] & \texttt{h (s+2)} \\
\texttt{f x y} \arrow[r] & \texttt{g (x-1)} \arrow[r] & \texttt{f (u-2) (u-3)} \arrow[ur] \arrow[dr] & & \\
 & & & \texttt{g (h (y-1))} &
\end{tikzcd}$$

where the call from `f` to `h` is a *nested call* because the value returned from `h` is passed as argument to another call (to `g`). The corresponding call path is

[`f x y` [`g (x-1)` [`f (u-2) (u-3)` [`h (y-1)` [`h (s+2)`]] [`g (h (y-1))`]]]].

Nested calls are distinguished from *tail calls* in call paths by the fact that their immediately enclosing brackets are followed by an opening bracket—in this case [`h (y-1)`...] is followed by [`g (h (y-1))`].

Intuitively, '[' represents a call and ']' a return from a call. Note that call paths represent *partially completed* computations: a call path can be extended, even infinitely if the program contains recursive calls. Furthermore, when extending, a tail call can turn into a nested call. For instance, the above call path can be obtained by extending

```
[f x y [g (x-1) [f (u-2) (u-3) [h (y-1) [h (s+2)]]]]]
```

changing `h (y-1)` from tail to nested call.

4.1 Call path grammar

Formally, call paths are the words of a language $L(G)$ over a grammar $G = \{S ::= [f1\ x_1 \ldots x_n\ C_{e_{f1}}]\} \cup G'$, where G' is generated by traversing the program and producing rules according to the scheme in Figure 4.1. Note that the ε in (g_5') enables the production of call paths representing *partial* computations, and rule (g_3) enables call paths for *potential* computations: the actual value of e_1 is not considered when choosing between e_2 and e_3.

Expression e	Grammar rules produced by e	
c	$C_e ::= \varepsilon$	(g_1)
x	$C_e ::= \varepsilon$	(g_2)
if e_1 **then** e_2 **else** e_3	$C_e ::= C_{e_1}C_{e_2} \mid C_{e_1}C_{e_3}$	(g_3)
$b\ e_1 \ldots e_n$	$C_e ::= C_{e_1} \ldots C_{e_n}$	(g_4)
$f\ e_1 \ldots e_n$	$C_e ::= C_{e_1} \ldots C_{e_n} C_e'$	(g_5)
	$C_e' ::= [f\ e_1 \ldots e_n\ C_{e_f}] \mid \varepsilon$	(g_5')

Figure 4.1: Scheme for producing grammar rules from the subject program

4.2 Subpaths

We define a *prefix* of a call path $\pi = [f\ \vec{e}\ \pi_1 \cdots \pi_n]$ to be either π or $[f\ \vec{e}\ \pi_1 \cdots \pi_{i-1}\ \pi'_i]$, with $0 \leq i \leq n$, where π'_i is a prefix of π_i. A *subpath* of π is either a prefix of π or a subpath of some π_i, and the *final call* of π is $f\ \vec{e}$ if $n = 0$, otherwise it is the final call of π_n. Thus, for

$$\pi = [\texttt{f x y}[\texttt{g (x-1)}[\texttt{f (u-2) (u-3)}[\texttt{h (y-1)}[\texttt{h (s+2)}]]\,[\texttt{g (h (y-1))}]]]]$$

some subpaths are

$$[\texttt{f x y } [\texttt{g (x-1) } [\texttt{f (u-2) (u-3) } [\texttt{h (y-1) } [\texttt{h (s+2)}]]]]]$$

(with final call `h (s+2)`) and

$$[\texttt{g (x-1) } [\texttt{f (u-2) (u-3) } [\texttt{h (y-1) } [\texttt{h (s+2)}]]\,[\texttt{g (h (y-1))}]]],$$

(with final call `g (h (y-1))`). In general, the notation $[\ldots[f\ \vec{e}]\!]\!]$ indicates that $f\ \vec{e}$ is the final call.

We define a flattening function that operates on call paths $\cdot^{\flat} : \mathit{CallPath} \to (\mathit{Funname} \times \mathit{Expression}^n)\,\mathit{List}$. Informally speaking, it erases all but the outermost square brackets,

$$\begin{array}{rcl} [f\ \vec{e}]^{\flat} & = & [f\ \vec{e}] \\ [f\ \vec{e}\,\pi_1 \ldots \pi_n]^{\flat} & = & (f\ \vec{e}) : \pi_1^{\flat} + \!\!+ \cdots + \!\!+ \pi_n^{\flat}, \end{array}$$

corresponding to taking a prefix of an inorder traversal of the call tree for the potential computations.

4.3 State transformers

We now extend the notion of a *state transformer* $st^{\cdot} : \mathit{CallPath} \to \mathit{Expression}$ defined by Glenstrup and Jones (1996) to fully recursive programs by the definition

$$\begin{array}{lcl} st^{[f\ \vec{e}]} & = & \vec{e} \\ st^{[f\ \vec{e}\ \pi_1 \ldots \pi_n]} & = & [f_1 \mapsto e_1, \ldots, f_m \mapsto e_m](st^{\pi_n}), \end{array}$$

where $[\ldots \mapsto \ldots]$ denotes parallel substitution. Thus nested calls $\pi_1 \cdots \pi_{n-1}$ do not contribute to the state transformer—they appear directly as call expressions in π_n. Intuitively st^π expresses the expressions $\vec{e}^k$ of the final call as a function of the free variables (function parameters) of the expressions $\vec{e}^1$ of the first call.

Example 4.1
Consider the program p

```
gcd x y =  if x = y  then x  else
           if gt x y then gcd (sub x y) y  else gcd x (sub y x)
gt x y  =  x > y
sub x y =  if y = 0 then x else sub (x - 1) (y - 1)
```

After some trivial simplifications, the grammar G for p is

$$\begin{array}{lll}
S & ::= & [\texttt{gcd x y } C_{\mathrm{if}_1}] \\
C_{\mathrm{if}_1} & ::= & \varepsilon \mid C_{\mathrm{if}_2} \\
C_{\mathrm{if}_2} & ::= & C_{\texttt{gt}} C_{\texttt{subxy}} C_{\texttt{gcdsubxyy}} \mid C_{\texttt{gt}} C_{\texttt{subyx}} C_{\texttt{gcdxsubyx}} \\
C_{\texttt{gt}} & ::= & \varepsilon \mid [\texttt{gt x y}] \\
C_{\texttt{subxy}} & ::= & \varepsilon \mid [\texttt{sub x y } C_{\mathrm{if}_3}] \\
C_{\mathrm{if}_3} & ::= & \varepsilon \mid C_{\texttt{subx}-\texttt{1y}-\texttt{1}} \\
C_{\texttt{subx}-\texttt{1y}-\texttt{1}} & ::= & \varepsilon \mid [\texttt{sub (x - 1) (y - 1) } C_{\mathrm{if}_3}] \\
C_{\texttt{gcdsubxyy}} & ::= & \varepsilon \mid [\texttt{gcd (sub x y) y } C_{\mathrm{if}_2}] \\
C_{\texttt{subyx}} & ::= & \varepsilon \mid [\texttt{sub y x } C_{\mathrm{if}_3}] \\
C_{\texttt{gcdxsubyx}} & ::= & \varepsilon \mid [\texttt{gcd x (sub y x) } C_{\mathrm{if}_2}]
\end{array}$$

and some examples of call paths $\pi \in L(G)$ are

$$\begin{array}{l}
[\texttt{gcd x y }[\texttt{gt x y}][\texttt{sub x y }[\texttt{sub (x-1) (y-1)}]] \\
\qquad\qquad [\texttt{gcd (sub x y) y }[\texttt{gt x y}][\texttt{sub x y}]]]
\end{array}$$

with $st^\pi = \texttt{sub (sub x y) y}$ and

$$\begin{array}{l}
[\texttt{gcd x y }[\texttt{gt x y}][\texttt{sub x y}] \\
\qquad\qquad [\texttt{gcd (sub x y) y }[\texttt{gt x y}][\texttt{sub y x}][\texttt{gcd (sub y x) y}]]]
\end{array}$$

with $st^\pi = \texttt{gcd (sub y (sub x y) y)}$.

Given the argument values of the ith function f^i, call paths can be used to calculate what argument values f^i passes in a call to the kth function f^k, as shown in the following

Lemma 4.2
Assume $\exists s' \forall s : \mathcal{E}\ [\![e^{f^i}]\!]\ \{f^i_1 \mapsto v^i_1, \ldots, f^i_m \mapsto v^i_m\}\ s\ \underbrace{\rightarrow \cdots \rightarrow}_{k-i-1\ \mathcal{C}\ \text{ops}}\ \mathcal{C}\ [\![f^k]\!]\ \vec{v}^k\ (s' +\!\!+\ s)$.

Given

$$\begin{array}{lcl} \pi & = & [f^1\ \vec{e}^1\ \pi_1 \ldots \pi_{n-1}\ \pi_n], \quad \text{where } \pi_n = [\ldots[f^i\ \vec{e}^i]\!]] \\ \pi' & = & [f^1\ \vec{e}^1\ \pi_1 \ldots \pi_{n-1}\ \pi'_n], \quad \text{where } \pi'_n = [\ldots[f^i\ \vec{e}^i\ \hat{\pi}_1 \ldots \hat{\pi}_l\ [f^k\ \vec{e}^k]\!]] \\ \multicolumn{3}{l}{\mathcal{E}\ [\![st^\pi]\!]\ \{f^0_1 \mapsto v^0_1, \ldots, f^0_p \mapsto v^0_p\} = \vec{v}^i} \end{array}$$

then $\mathcal{E}\ [\![st^{\pi'}]\!]\ \{f^0_1 \mapsto v^0_1, \ldots, f^0_p \mapsto v^0_p\} = \vec{v}^k$.

Proof: by total induction on the length of $\pi^\flat$.

Case *length* $\pi^\flat = 1$**:** then $\pi \equiv [f^1\ \vec{e}^1]$, $\pi' \equiv [f^1\ \vec{e}^1\ \hat{\pi}_1 \ldots \hat{\pi}_l\ [f^k\ \vec{e}^k]]$, and $st^\pi = \vec{e}^1$, $st^{\pi'} = [f^1_1 \mapsto e^1_1, \ldots, f^1_m \mapsto e^1_m]\vec{e}^k$. By assumption we have for some s'

$$\begin{array}{l} \mathcal{E}\ [\![e^{f^1}]\!]\ \{f^1_1 \mapsto v^1_1, \ldots, f^1_m \mapsto v^1_m\}\ s \rightarrow^* \\ \mathcal{E}\ [\![f^k\ \vec{e}^k]\!]\ \{f^1_1 \mapsto v^1_1, \ldots, f^1_m \mapsto v^1_m\}\ (s' +\!\!+\ s) \rightarrow^+ \\ \mathcal{E}\ [\![e^k_1]\!]\ \{f^1_1 \mapsto v^1_1, \ldots, f^1_m \mapsto v^1_m\}\ s_1 \rightarrow^* \mathcal{A}\ s_1\ v^k_1 \rightarrow \\ \quad \vdots\ (q \text{ times}) \\ \mathcal{E}\ [\![e^k_q]\!]\ \{f^1_1 \mapsto v^1_1, \ldots, f^1_m \mapsto v^1_m\}\ s_q \rightarrow^* \mathcal{A}\ s_q\ v^k_q \rightarrow \cdots \end{array}$$

where $s_1 = (\langle f\ \bullet\ e_2 \ldots e_q, \varrho\rangle : s' +\!\!+\ s), \ldots, s_q = (\langle f\ v_1 \ldots v_{q-1}\ \bullet, \varrho\rangle : s' +\!\!+\ s)$, and it is straightforward to show that the q evaluations of $e^k_1, \ldots, e^k_q$ using these stacks produce the same values for any stack s'', in other words, $\mathcal{E}\ [\![\vec{e}^k]\!]\ \{f^1_1 \mapsto v^1_1, \ldots, f^1_m \mapsto v^1_m\} = \vec{v}^k$, so by the Substitution Lemma (cf. page 46) we find that $\mathcal{E}\ [\![st^{\pi'}]\!]\ \{f^0_1 \mapsto v^0_1, \ldots, f^0_p \mapsto v^0_p\} = \mathcal{E}\ [\![[f^1_1 \mapsto e^1_1, \ldots, f^1_m \mapsto e^1_m]\vec{e}^k]\!]\ \{f^0_1 \mapsto v^0_1, \ldots, f^0_p \mapsto v^0_p\} = \vec{v}^k$.

Case *length* $\pi^\flat > 1$**:** For some $\vec{v}$, $\mathcal{E}\ [\![\vec{e}^1]\!]\ \{f^0_1 \mapsto v^0_1, \ldots, f^0_p \mapsto v^0_p\} = \vec{v}^1$, so by the

Substitution Lemma we know that

$$\begin{aligned} & \mathcal{E}\ [\![st^{\pi_n}]\!]\ \{f_1^1 \mapsto v_1^1,\ldots,f_m^1 \mapsto v_m^1\} \\ = \quad & \mathcal{E}\ [\![[f_1^1 \mapsto e_1^1,\ldots,f_m^1 \mapsto e_m^1]\, st^{\pi_n}]\!]\ \{f_1^0 \mapsto v_1^0,\ldots,f_p^0 \mapsto v_p^0\} \\ = \quad & \mathcal{E}\ [\![st^{\pi}]\!]\ \{f_1^0 \mapsto v_1^0,\ldots,f_p^0 \mapsto v_p^0\} = \vec{v}^i. \end{aligned}$$

By using the same lemma again, and the induction hypothesis, we finally obtain

$$\begin{aligned} & \mathcal{E}\ [\![st^{\pi'}]\!]\ \{f_1^0 \mapsto v_1^0,\ldots,f_p^0 \mapsto v_p^0\} \\ = \quad & \mathcal{E}\ [\![[f_1^1 \mapsto e_1^1,\ldots,f_m^1 \mapsto e_m^1]\, st^{\pi'_n}]\!]\ \{f_1^0 \mapsto v_1^0,\ldots,f_p^0 \mapsto v_p^0\} \\ = \quad & \mathcal{E}\ [\![st^{\pi'_n}]\!]\ \{f_1^1 \mapsto v_1^1,\ldots,f_m^1 \mapsto v_m^1\} = \vec{v}^k \end{aligned}$$

□

We now prove for any fully recursive program p the following theorem that shows the connection between call paths and state transition sequences (cf. the tail recursive case in Glenstrup & Jones, 1996, Lemma 3). It is useful for analysing program computations: if we can prove some property for all call paths, this will be true for all actual computations.

Theorem 4.3 (Connecting call paths and computations via st^{π})
For any state transition sequence $(f1,\vec{v}^1) \to \cdots \to (f^k,\vec{v}^k)$ there exists a call path $\pi \in L(G)$ with $\pi^\flat = [f1\,\vec{x}^1,\ldots,f^k\vec{e}^k]$ such that $\mathcal{E}\ [\![st^{\pi'}]\!]\ \{f_1^s \mapsto v_1^s,\ldots,f_n^s \mapsto v_n^s\} = \vec{v}^t$ for all subpaths $\pi' = [f^s\vec{e}^s\ [\ldots[f^t\vec{e}^t]\!]\!]$ of π. Call path π is termed the corresponding call path *of the transition sequence.*

Proof: Using induction on the length of the transition sequence. The base case is clear, as $[f1\ \vec{x}] \in L(G)$ and the required equations are easily checked. For the induction step, let the sequence be given by $\tau = (f1,\vec{v}^1) \to \cdots \to (f^{k-1},\vec{v}^{k-1}) \to (f^k,\vec{v}^k)$ and the call path supplied by the hypothesis by $\pi_{k-1} = [f1\ \vec{x}^1\ [\ldots[f^{k-1}\vec{e}^{k-1}]\!]\!]$.

Let $r \geq 0$ denote the number of function returns encountered after calling f^{k-1} before f^k is called. Now π_k is constructed from π_{k-1} by starting at the final call and moving past r right brackets before inserting $f^k\ \vec{e}^k$:

$$\pi_k = \overbrace{[\ldots[f^i\ \vec{e}^i\,\underbrace{[f^{i+1}\ \vec{e}^{i+1}\ldots[}_{r\text{ unbalanced brackets}}f^{k-1}\ \vec{e}^{k-1}\underbrace{]\ldots]}_{r\text{ brackets}}}^{\pi_{k-1}}[f^k\ \vec{e}^k]\!]\!]$$

We can see that $\pi_k \in L(G)$:

1. $f^i\ \vec{e}^i$ must be present in π_{k-1}, because to make r returns, there must have been r nested calls.

2. f^i must be able to call f^k (obvious), so e^{f^i} must contain a call to f^k.

3. Thus, to generate π_{k-1}, rule (g_5) must have been used, with (g'_5) generating ε. If we instead let (g'_5) generate $[f^k\ \vec{e}^k\ \varepsilon]$, we get π_k.

To show the required equations, let $\pi'_k = [f^s\ \vec{e}^s\ [\ldots[f^i\ \vec{e}^i\ \hat{\pi}_1 \ldots \hat{\pi}_l\ [f^t\ \vec{e}^t]]$ be a subpath of π_k. Wlog. we can assume $t = k$; otherwise the equations are satisfied by virtue of the properties of π_{k-1}.

As $\pi'_i = [f^s\ \vec{e}^s\ [\ldots[f^i\ \vec{e}^i]]$ is a subpath of π_{k-1}, we know that $\mathcal{E}\ [\![st^{\pi'_i}]\!]\ \{f^s_1 \mapsto v^s_1, \ldots, f^s_n \mapsto v^s_n\} = \vec{v}^i$, and then by Lemma 4.2 we obtain $\mathcal{E}\ [\![st^{\pi'_k}]\!]\ \{f^s_1 \mapsto v^s_1, \ldots, f^s_n \mapsto v^s_n\} = \vec{v}^t$. □

Chapter 5

Capturing Parameter Dependency

The termination detection presented in this paper works by considering how parameters behave in recursive calls, and to this end we need an approximation of how the actual parameter values are computed.

These *parameter dependencies* express whether the parameters decrease or increase at each function call, and for simple argument expressions like in

```
f x y = ... g (x + 1) (y - 1) ...
```

this is straightforward, but when we consider nested calls we must consider how their return values depend on their arguments. For instance, in

```
f x y = ... h (g (x + 1) (y - 1)) ...,
```

apart from recording that `g`'s parameters depend on `f`'s, we must also consider how `h`'s parameters depend on `f`'s via the nested call to `g`.

In the present chapter we develop machinery to perform an analysis of how the value size of an expression e depends on the values of the parameters of the function it occurs in.

One of our goals is not to handle constants too conservatively. In a function like

```
f x = if length x < 2 then
        [4, 2, 1, 7]
      else
        cdr (f (cddr x))
```

we do *not* want the analysis to tell us that the result of `f x` is greater than `x` (cf. the analysis in Andersen & Holst, 1996), even though this is true when list `x` is shorter than 3.

The reason is this: when detecting nontermination, we are looking for parameters that can risk increasing *beyond any bound*. Given a variable x in program p, if for some program input we find that x is bound to infinitely many different values during evaluation, we say that x is *unbounded*. Similarly, if an expression e for some input is encountered infinitely many times during evaluation and there is no bound on the values it evaluates to, e is said to be unbounded.

When function `f` defined above is used in a nested call like

```
g x = ... g (f x) ...
```

`x` should not be deemed unbounded just because of `f`'s increasing effect on small `x` values. Thus it would be unnecessarily conservative to state that `f x` can cause a "dangerous increase" of `x`.

On the other hand, the dependency analyses must be *safe*. We present two analyses with different modalities, a *must-always-decrease* and a *might-possibly-increase* analysis. Whenever the must-always-decrease analysis produces a decreasing dependency, all actual computations must exert a decreasing behaviour. Further, if there is a risk of unbounded increase, the might-possibly-increase analysis must state this.

5.1 Material dependency

Several extensionally equal functions (i.e. seen as "black boxes" they always yield equal results) can be intensionally different. An example is the add function for natural numbers:

```
add₁ x y = x + y
```

```
add2 x y = if x = 0 then y else add2 (x - 1) (y + 1)

add3 x y = if x = 0 then y else 1 + add3 (x - 1) y

add4 x y = if x = 0 then
             if y = 0 then 0 else 1 + add4 x (y - 1)
           else
             1 + add4 (x - 1) y
```

One of the (intensional) differences between these functions is how the return value is constructed: in `add`$_1$, both `x` and `y` are used in a basic operation: we say that `add`$_1$ `x y` is *materially dependent* on `x` and `y`. In `add`$_2$ and `add`$_3$ the value of `x` is tested, but is not used in a basic operation to construct the return value; `add`$_2$ `x y` and `add`$_3$ `x y` are only materially dependent on `y`. Similarly, `add`$_4$ `x y` is materially dependent on neither `x` nor `y`.

Formally, an expression e is materially dependent on one of its free variables x iff the value of e potentially (i.e. assuming either branch of an **if**-expression can be taken) can be constructed by basic operations applied to the value of x. These variables can easily be determined syntactically by descending the expression recursively, collecting all variables not occurring in the conditional part of an **if**-expression. If the size of an expression e is related to some of its free variables $\vec{x}$ when these are given infinitely many different values (i.e. $\exists\varrho\forall K\exists v : \mathcal{E}\;[\![e]\!]\;(\varrho + \{x \mapsto v\}) > K$) without being materially dependent on $\vec{x}$, we say the expression is *immaterially dependent* on $\vec{x}$.

Let $\tau = \mathcal{E}\;[\![e]\!]\;\varrho\; s \rightarrow \cdots \rightarrow \mathcal{A}\; s\; v$ be the transition sequence for evaluating an expression e in the environment ϱ (note that this is a subsequence of the evaluation of a program). If there exists some program input such that for any bound K, there exists an evaluation of e encountered during program evaluation (using the program input) in which the number of $\mathcal{C}$-operators in τ is greater than K, we say e requires an unbounded number of function calls to be computed.

We can now make the following

Observation 5.1 (Sources of unboundedness)
An expression can only be unbounded if it is materially dependent on an unbounded variable or the expression requires an unbounded number of function calls to be computed

5.2 Capturing size dependency

As we are trying to capture the behaviour of parameter values as computed by function calls we will need some mechanism to describe how the value of an expression relates to those of its free variables.

Our starting point is the underlying domain, *Value*, with some well-founded partial order $\leq$. We are interested in two conceptually separate properties of an expression:

Decreasing property: its ability *whenever* evaluation terminates to yield a value (equal to or) less than the value of some of its free variables, and

Non-bounding property: its risk of *possibly* being able to yield a value greater than (or just related to) the value of some of its free variables.

For example, the conditional expression **if** `x` **then** `y` **else** `cdr y` will always yield a value less than or equal to `y`'s value, while the conditional expression **if** `x` **then** `y` **else** `cons 1 y` is potentially "dangerous": it can possibly yield a value greater than the value of `y`.

To this end, we introduce a set, D, of *dependency operators* ↓,↧,↥,↑, and we call them *decreasing*, *non-increasing*, *non-bounding* and *diverging* operators, respectively. We also impose a partial order $\sqsubseteq$ on operators such that ↧ $\sqsubseteq$ ↓, ↥ $\sqsubseteq$ ↑ and $\delta \sqsubseteq \delta$ for any $\delta \in D$.

An operator is combined with a variable name and is intended to be a description relative to this variable. Thus e.g. the dependency "$\downarrow(f_x)$" should be read as "always strictly less than f_x." As the expression value can depend on several of the free variables we gather the dependencies for an expression in two dependency sets: $\Delta^{\downarrow}$ for the decreasing and $\Delta^{\uparrow}$ for the increasing operators. Some examples of possible dependency sets are shown in Figure 5.1 on the following page.

Expression e	Δ for e	Intuitive reading
if `x` **then** `car y`	{↓(`y`)}	always less than `y`
else `cdr y`	{↥(`y`)}	possibly unbounded for unbounded `y`
if `x` **then** `car y`	{↧(`y`)}	always less than or equal to `y`
else `y`	{↥(`y`)}	possibly unbounded for unbounded `y`

`cdr (min x y)`	{↓(`x`),↓(`y`)}
	{↥(`x`),↥(`y`)}
		always less than `x` and less than `y`
		possibly unbounded for unbounded `x` or `y`

`cons x y`	{}
	{↑(`x`),↑(`y`)}
		no decreasing guarantees
		possibly greater than `x` or greater than `y`

if `x` **then** `y`	{}	no decreasing guarantees
else `cons 1 z`	{↥(`y`),↑(`z`)}	possibly unbounded for unbounded `y` or greater than `z`

Figure 5.1: Some examples of dependency sets

Formally, we introduce the following domains:

$\mu \in \{\downarrow,\uparrow\}$	decreasing and non-bounding modality
$\delta \in D^{\downarrow} = \{\downarrow,$↧$\}, \quad \delta \in D^{\uparrow} = \{\uparrow,$↥$\}$	dependency operator
$\delta(f_x) \in Dep^{\mu} = D^{\mu} \times Varname$	dependency
↥$(K) \in BoundDep = \{$↥$\} \times Value$	bound dependency
$\Delta \in DepSet^{\downarrow} = \mathcal{P}(Dep^{\downarrow})$	decreasing dependency set
$\Delta \in DepSet^{\uparrow} = \mathcal{P}(Dep^{\uparrow} \cup BoundDep)$	increasing dependency set
$\rho \in DepEnv^{\mu} = Varname \rightarrow DepSet^{\mu}$	dependency environment,
$\phi \in DepFunEnv^{\mu} = Funname \rightarrow (DepSet^{\mu})^{n} \rightarrow DepSet^{\mu}$	function return value dependency environment

Note that the bound dependency ↥(K) is only used to state the safety of $\mathcal{E}^{\uparrow}$; it is never computed in an actual implementation. We extend the partial order

$\sqsubseteq$ to dependencies, dependency sets and dependency environments by

$$\begin{array}{rcl} \delta_1(f_x) \sqsubseteq \delta_2(g_y) & \Leftrightarrow & f_x \equiv g_y \wedge \delta_1 \sqsubseteq \delta_2 \\ \Delta_1 \sqsubseteq \Delta_2 & \Leftrightarrow & \forall(\delta_1(f_x)) \in \Delta_1 \exists(\delta_2(g_y)) \in \Delta_2 : \delta_1(f_x) \sqsubseteq \delta_2(g_y) \\ \rho_1 \sqsubseteq \rho_2 & \Leftrightarrow & \forall f_x : \rho_1\ f_x \sqsubseteq \rho_2\ f_x \quad (\text{when } \mathit{dom}\ \rho_1 = \mathit{dom}\ \rho_2) \end{array}$$

The dependency set domains have the following top and bottom elements:

$$\begin{array}{rclrcl} \bot_{\mathit{DepSet}^{\downarrow}} & = & \{\} & \top_{\mathit{DepSet}^{\downarrow}} & = & \{\downarrow(x) \mid x \in \mathit{Varname}\} \\ \bot_{\mathit{DepSet}^{\uparrow}} & = & \{\} & \top_{\mathit{DepSet}^{\uparrow}} & = & \{\uparrow(x) \mid x \in \mathit{Varname}\} \end{array}$$

and the partial orders on the dependency set transformer domains are given by

$$\begin{array}{lcl} \forall f_1^{\downarrow}, f_2^{\downarrow} \in (\mathit{DepSet}^{\downarrow})^n \to \mathit{DepSet}^{\downarrow} : (f_1^{\downarrow} \sqsubseteq f_2^{\downarrow} & \Leftrightarrow & \forall \vec{\Delta} : f_1^{\downarrow}\ \vec{\Delta} \sqsupseteq f_2^{\downarrow}\ \vec{\Delta}) \\ \forall f_1^{\uparrow}, f_2^{\uparrow} \in (\mathit{DepSet}^{\uparrow})^n \to \mathit{DepSet}^{\uparrow} : (f_1^{\uparrow} \sqsubseteq f_2^{\uparrow} & \Leftrightarrow & \forall \vec{\Delta} : f_1^{\uparrow}\ \vec{\Delta} \sqsubseteq f_2^{\uparrow}\ \vec{\Delta}) \end{array}$$

Note the reversed relation in the first of these two definitions.

5.3 Safety of size approximations

If we are to use dependency sets for expressions as approximations of their actual values during computation we must formally define what we mean by requirements like "always less than…" and "possibly unbounded…", in other words how the approximations must relate to the actual values.

Given a value environment ϱ, we can describe how a concrete value v relates to the values of the variables in $\mathit{dom}\ \varrho$ by the function $\alpha_{\varrho}^{\downarrow} : \mathit{Value} \to \mathit{DepSet}^{\downarrow}$ given by

$$\alpha_{\varrho}^{\downarrow}\ v = \{\downarrow(f_x) \mid v < \varrho\ f_x\} \cup \{\overline{\downarrow}(f_x) \mid v \leq \varrho\ f_x\},$$

and we can extend it to $\alpha_{\varrho}^{\downarrow} : \mathit{Environment} \to \mathit{DepEnv}^{\downarrow}$ by letting $\alpha_{\varrho}^{\downarrow}\ \varrho'\ f_x = \alpha_{\varrho}^{\downarrow}\ (\varrho'\ f_x)$.

Given e.g. the environment $\varrho = \{\mathtt{x} \mapsto \mathtt{[1]}, \mathtt{y} \mapsto \mathtt{[]}\}$ we get $\alpha_{\varrho}^{\downarrow}\ \varrho = \{\mathtt{x} \mapsto \{\overline{\downarrow}(\mathtt{x})\}, \mathtt{y} \mapsto \{\overline{\downarrow}(\mathtt{y}), \downarrow(\mathtt{x}), \overline{\downarrow}(\mathtt{x})\}\}$.

In a somewhat similar way we define $\alpha^{\uparrow}_{\varrho,K} : \mathit{Value} \to \mathit{DepSet}^{\uparrow}$ by

$$\alpha^{\uparrow}_{\varrho,K}\, v = \{\uparrow(f_x) \mid \varrho\, f_x < v\} \cup \{\twoheaduparrow(K) \mid K \leq v\},$$

where $K \in \mathit{Value}$ is a bound that v is compared to, and extend it exactly like $\alpha^{\downarrow}_{\varrho}$ to environments. Later, we will consider $\alpha^{\uparrow}_{\varrho,K}\, v$ for varying bounds K, the idea being that if some expression value can exceed all bounds by varying some parameter f_x, it must be classified at least as $\twoheaduparrow(f_x)$.

Size approximation functions $\mathcal{E}^{\downarrow}$ and $\mathcal{E}^{\uparrow}$

We now assume given two size approximation functions: a decreasing size function $\mathcal{E}^{\downarrow} : \mathit{Expression} \to \mathit{DepEnv}^{\downarrow} \to \mathit{DepSet}^{\downarrow}$ and an increasing size function $\mathcal{E}^{\uparrow} : \mathit{Expression} \to \mathit{DepEnv}^{\uparrow} \to \mathit{DepSet}^{\uparrow}$. When these functions are given an expression and an estimate of the values of the free variables (represented by the dependency environment), they must return a safe estimate of the expression value:

$\mathcal{E}^{\downarrow}$ must be safe in the sense that if $\mathcal{E}^{\downarrow}$ claims that the value of an expression e is less than some parameter f_x, then no matter under which environment ϱ the expression is evaluated, the *actual* value of e—if it exists—must be less than the *actual* value of f_x. Thus $\mathcal{E}^{\downarrow}$ must satisfy

Definition 5.2 (Safety of $\mathcal{E}^{\downarrow}$)
$\mathcal{E}^{\downarrow}$ is called safe *iff*

$$\forall e \forall \varrho : \mathit{fv}\ e \subseteq \mathit{dom}\ \varrho \Rightarrow \mathcal{E}^{\downarrow}\ [\![e]\!]\ (\alpha^{\downarrow}_{\varrho}\ \varrho) \sqsubseteq \alpha^{\downarrow}_{\varrho}\ (\mathcal{E}\ [\![e]\!]\ \varrho)$$

This implies that removing $\downarrow$,$\twoheaddownarrow$-dependencies from a safe estimate is safe (corresponding to conservatively ignoring some decreasing transitions in the program).

Towards formulating safety of $\mathcal{E}^{\uparrow}$ One might think that for $\mathcal{E}^{\uparrow}$ to be safe one must under similar quantification require the reverse relationship $\alpha^{\uparrow}_{\varrho,K}\ (\mathcal{E}\ [\![e]\!]\ \varrho) \sqsubseteq \mathcal{E}^{\uparrow}\ [\![e]\!]\ (\alpha^{\uparrow}_{\varrho,K}\ \varrho)$, but this turns out to be too strong: for $e \equiv$ **if** ... **then** `x` **else** `cons 1 y` this would require that $\uparrow(\mathtt{x}) \in \mathcal{E}^{\uparrow}\ [\![e]\!]\ (\alpha^{\uparrow}_{\varrho,K}\ \varrho)$, as $\alpha^{\uparrow}_{\varrho,K}$ $(\mathcal{E}\ [\![e]\!]\ \varrho) = \{\uparrow(\mathtt{x}),\uparrow(\mathtt{y})\}$ for $\varrho = \{\mathtt{x} \mapsto \mathtt{[]}, \mathtt{y} \mapsto \mathtt{[]}\}, K = \mathtt{[1,1]}$, and yet it cannot yield unboundedly many values greater than `x` for fixed `y`.

The problem is that we want to attribute increasing dependencies to expressions according to their *asymptotic* behavior, i.e. what the relation between function input and output is when the size of the input goes towards infinity. Intuitively, if we vary the value of a parameter f_x while holding the remaining parameters fixed, we want to attribute $\uparrow(f_x)$ to e if the value of e asymptotically is greater than the value of f_x, and $⇞(f_x)$ if e asymptotically can yield unbounded values (though not necessarily greater than the value of f_x). Figure 5.2 shows some examples of the behaviour of expressions for varying `x` and fixed $\mathtt{y} = 4$, together with the kind of increasing dependencies, relative to `x`, we want to attribute to them. The dependency $⇞(K)$ expresses that the expression value is greater than a bound K; in the example K is greater than 4 so e_3 need not be attributed $⇞(K)$.

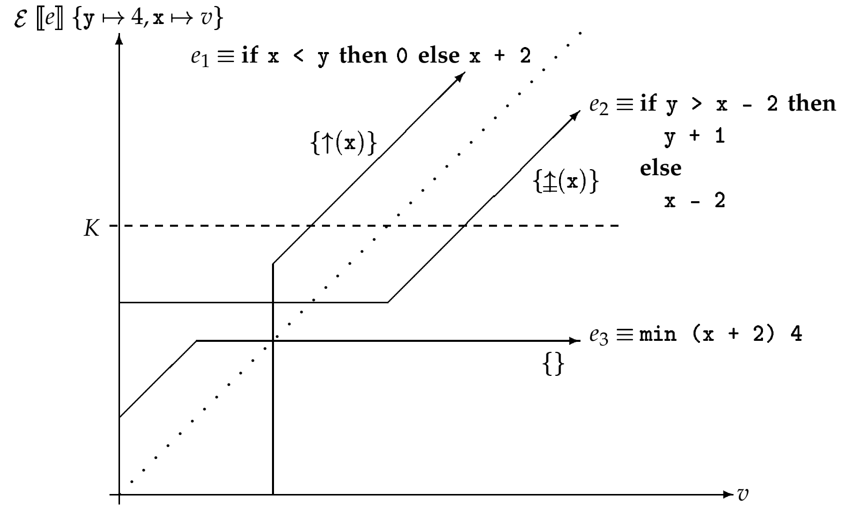


Figure 5.2: Expression value behavior when varying one parameter value

Expressions with several variables In general, however, looking at the expression value while varying just one variable is not sufficient: consider the program

```
f x y = ... if ... f (g x y) (g x y) ... else ...
g u v = 1 + min u v
```

For any one value of `u` it is correct to conclude that `g` is bounded by (the value of) `1 + u` (and similarly for `v`), yet `g` is *not* bounded when *both* `u` and `v` vary, causing nontermination in the present example.

The solution is for an expression e to consider all subsets of the free variables when varying: for every subset X that leads to unbounded variation in e there must be at least one $x \in X$ for which $\delta(x) \in \mathcal{E}^{\uparrow}\,[\![e]\!]\ \rho^{\uparrow}_{id}$ with $\rho^{\uparrow}_{id} = \{\ddagger(x) \mid x \in fv\ e\}$, where $\delta = \uparrow$ or $\ddagger$, depending on whether or not the value of e is strictly greater than that of x infinitely often.

Formally, for a set of variables X, we define a partial order $\sqsubseteq_X\ \subseteq DepSet^{\uparrow} \times DepSet^{\uparrow}$ that only compares dependencies using *some* of the variables, namely those found in X:

$$\begin{aligned}\Delta_1 \sqsubseteq_X \Delta_2 \Leftrightarrow ((&\forall f_x \in X\, \forall \delta_1(f_x) \in \Delta_1 \exists g_y \in X\ \exists \delta_2(g_y) \in \Delta_2 : \delta_1 \sqsubseteq \delta_2)\\ \wedge\, (&\forall \delta_1(K) \in \Delta_1\, \exists g_y \in X\ \exists \delta_2(g_y) \in \Delta_2 : \delta_1 \sqsubseteq \delta_2))\end{aligned}$$

Note that X and $\sqsubseteq_X$ are only used to state the safety of $\mathcal{E}^{\uparrow}$; it is never computed in an actual implementation. Here are some examples of its use:

$$\begin{array}{rlll}
\{\uparrow(\mathtt{x}),\ddagger(\mathtt{y}),\uparrow(\mathtt{z})\} & \sqsubseteq_{\{\mathtt{x},\mathtt{y}\}} & \{\uparrow(\mathtt{x}),\uparrow(\mathtt{y})\} & \text{because } \mathtt{z} \notin \{\mathtt{x},\mathtt{y}\}, \uparrow \sqsubseteq \uparrow, \ddagger \sqsubseteq \uparrow\\
\{\uparrow(\mathtt{x}),\ddagger(\mathtt{y}),\uparrow(\mathtt{z})\} & \sqsubseteq_{\{\mathtt{x},\mathtt{z}\}} & \{\uparrow(\mathtt{x}),\uparrow(\mathtt{y})\} & \text{because } \uparrow(\mathtt{x}) \in \Delta_2\\
\{\uparrow(\mathtt{x}),\ddagger(\mathtt{y}),\uparrow(\mathtt{z})\} & \not\sqsubseteq_{\{\mathtt{z}\}} & \{\uparrow(\mathtt{x}),\uparrow(\mathtt{y})\} & \text{because } \uparrow(\mathtt{z}) \notin \Delta_2\\
\{\ddagger(\mathtt{x}),\ddagger(K)\} & \sqsubseteq_{\{\mathtt{x},\mathtt{y}\}} & \{\uparrow(\mathtt{x})\} & \text{because } \uparrow(\mathtt{x}) \in \Delta_2\\
\{\ddagger(\mathtt{x}),\ddagger(K)\} & \not\sqsubseteq_{\{\mathtt{y}\}} & \{\uparrow(\mathtt{x})\} & \text{because } \ddagger(K) \in \Delta_1\text{, but } \delta(\mathtt{y}) \notin \Delta_2
\end{array}$$

Note specifically that $\{\uparrow(\mathtt{u})\} \sqsubseteq_{\{\mathtt{u},\mathtt{v}\}} \{\uparrow(\mathtt{v})\}$, which is what enables us to let $\mathcal{E}^{\uparrow}\,[\![\mathtt{1\ +\ min\ u\ v}]\!]\ \rho^{\uparrow}_{id} = \{\uparrow(\mathtt{v})\}$, that is, we need not include *both* $\uparrow(\mathtt{u})$ and $\uparrow(\mathtt{v})$. Informally, the dependency set states that "if `v` is fixed, `1 + min u v` cannot be made to return unboundedly large values by varying other free variables (i.e. `u`)."

For $\mathcal{E}^{\uparrow}$ to be safe we then require the slightly weaker condition that

Definition 5.3 (Safety of $\mathcal{E}^{\uparrow}$)
$\mathcal{E}^{\uparrow}$ *is called* safe *iff*

$$\begin{array}{l}
\forall e \forall \varrho : \mathit{fv}\ e \subseteq \mathit{dom}\ \ \varrho \Rightarrow \\
\quad \forall X \subseteq \mathit{dom}\ \ \varrho : X \neq \{\} \Rightarrow \\
\qquad (\alpha^{\uparrow}_{\varrho',K}\ (\mathcal{E}\ [\![e]\!]\ \varrho') \sqsubseteq_X \mathcal{E}^{\uparrow}\ [\![e]\!]\ (\alpha^{\uparrow}_{\varrho',K}\ \varrho')) \quad \textit{where } \varrho' = \varrho + \{x_i \mapsto v_i\}_{x_i \in X} \\
\qquad\ \ \textit{for almost all } (\vec{v}, K) \in (\mathit{Value}^n \times \mathit{Value})
\end{array}$$

where in general if $P(v)$ is a predicate parameterised on v then the phrase "$P(v)$ for almost all v" means "$\{v \mid \neg P(v)\}$ is finite." Similarly, "$a \underset{\text{a.a.}v}{R} b$" means "$\{v \mid \neg(a\ R\ b)\}$ is finite." The definition implies that adding $\uparrow$,↥-dependencies to a safe estimate is safe (corresponding to pessimistically pretending there are some extra increasing transitions in the program).

Thus, in a diagram, we require:

$$\begin{tikzcd}
\mathit{DepEnv}^{\downarrow} \arrow[d, "\mathcal{E}^{\downarrow}\ [\![e]\!]"'] \arrow[dr, phantom, "\sqsubseteq"] & \mathit{Environment} \arrow[l, "\alpha^{\downarrow}_{\varrho}"'] \arrow[r, "\alpha^{\uparrow}_{\varrho,K}"] \arrow[d, "\mathcal{E}\ [\![e]\!]"] \arrow[dr, phantom, "\underset{\text{a.a.}(\vec{v},K)}{\sqsubseteq_X}"] & \mathit{DepEnv}^{\uparrow} \arrow[d, "\mathcal{E}^{\uparrow}\ [\![e]\!]"] \\
\mathit{DepSet}^{\downarrow} & \mathit{Value} \arrow[l, "\alpha^{\downarrow}_{\varrho}"] \arrow[r, "\alpha^{\uparrow}_{\varrho,K}"'] & \mathit{DepSet}^{\uparrow}
\end{tikzcd}$$

so $\mathcal{E}^{\downarrow}$ and $\mathcal{E}^{\uparrow}$ can be seen as lower and upper approximations to $\mathcal{E}$.

State transformers (cf. page 52) in conjunction with size dependency functions are very useful as they can reveal what parts of the first expression in a call path the last expression depends upon. Formally this is defined by

Definition 5.4 (Dependency along a call path)
Given a call path $\pi = [f^1\ \vec{e}^1\ [\ldots [f^k \vec{e}^k]\!]$ *we say* f^k_i depends on $x \in \mathit{fv}\ \vec{e}^1$ along π with effect δ iff

$$\delta(f^1_j) \in \mathcal{E}^{\mu}\ [\![e_i]\!]\ \rho^{\mu}_{id}\ \textit{for some } \delta \in D^{\mu}, \textit{where }\ \vec{e} = st^{\pi}\ \textit{ and } \begin{cases} \rho^{\uparrow}_{id}\ x = \{\text{↥}(x)\} \\ \rho^{\downarrow}_{id}\ x = \{\text{↧}(x)\} \end{cases}.$$

For $\delta \in D^{\downarrow}, D^{\uparrow}$ we say f_i^k depends decreasingly *respectively* increasingly *on x along π.*

Based on Observation 5.1 on page 60, we are now able to introduce some concrete size approximation functions and argue for their safety.

5.4 Concrete size approximations

An example of safe size approximation functions is given in Figure 5.3. The operator $\mathcal{RI} : \mathit{Expression} \to \mathit{Boolean}$, defined in Figure 5.4 on page 70, detects whether there is a risk of a "recursive increase," i.e. that the value produced by the expression is produced using an increasing constructor (`cons`) in (possibly mutually) recursive calls. Note that a function `f x = 42` returning a constant value need not be classified by $\uparrow(\mathtt{x})$, even though $\mathcal{E}\ [\![\mathtt{f\ x}]\!]\ \varrho > \mathcal{E}\ [\![\mathtt{x}]\!]\ \varrho$ for $\varrho\ \mathtt{x} \in \{0, \ldots, 41\}$. This matches our intuition that such a function does not in itself cause divergence.

Proposition 5.5 (Size function well-definedness)
The size functions $\mathcal{E}^{\downarrow}$ and $\mathcal{E}^{\uparrow}$ in Figure 5.3 are well-defined.

Proof: By checking that $\mathcal{E}^{\downarrow}_{\mathbf{e}}$ and $\mathcal{E}^{\uparrow}_{\mathbf{e}}$ are monotonic in ϕ, using structural induction on e, we verify that the fixpoint exists. For $\mathcal{E}^{\uparrow}_{\mathbf{e}}$ this is straightforward; for $\mathcal{E}^{\downarrow}_{\mathbf{e}}$, note that in the definition of the partial order on $(\mathit{DepSet}^{\downarrow})^n \to \mathit{DepSet}^{\downarrow}$ the ordering is reversed, cf. Section 5.2 on page 61. □

We would like to prove that the $\mathcal{E}^{\mu}$ operators are safe approximations, and to this end we first define a notion of correct environment description:

Definition 5.6 (Correct environment description)
Given an initial environment ϱ_0, we say that $\rho \in \mathit{DepEnv}^{\downarrow}$ ϱ_0-describes ϱ if and only if $\mathit{dom}\ \rho = \mathit{dom}\ \varrho$ and

$$\begin{aligned}\forall x_1 \in \mathit{dom}\ \varrho\ \forall x_2 \in \mathit{dom}\ \varrho_0 : {} & (\downarrow(x_2) \in \rho\ x_1 \Rightarrow \varrho\ x_1 < \varrho_0\ x_2) \wedge \\ & (\overline{\downarrow}(x_2) \in \rho\ x_1 \Rightarrow \varrho\ x_1 \leq \varrho_0\ x_2)\end{aligned}$$

From the definition of $\alpha^{\downarrow}_{\varrho}$ (cf. page 63) we can see that this is equivalent to $\forall x \in \mathit{dom}\ \varrho : \rho\ x \sqsubseteq \alpha^{\downarrow}_{\varrho_0}\ (\varrho\ x)$, or equivalently $\rho \sqsubseteq \alpha^{\downarrow}_{\varrho_0}\ \varrho$. The initial environment

$$\begin{aligned}
&\mathcal{E}^{\mu} = \mathcal{E}^{\mu}_{\mathbf{e}}\ (\mathit{fix}\ (\lambda\phi.\\
&\qquad \{\mathit{f1} \mapsto \lambda\Delta_1 \ldots \Delta_m.\mathcal{E}^{\mu}_{\mathbf{e}}\ \phi\ \llbracket e_1 \rrbracket\ \{x_1 \mapsto \Delta_1, \ldots, x_m \mapsto \Delta_m\}, \ldots,\\
&\qquad\ \ \mathit{fn} \mapsto \lambda\Delta_1 \ldots \Delta_k.\mathcal{E}^{\mu}_{\mathbf{e}}\ \phi\ \llbracket e_n \rrbracket\ \{x_1 \mapsto \Delta_1, \ldots, x_k \mapsto \Delta_k\}\}))
\end{aligned}$$

$$\begin{aligned}
&\mathcal{E}^{\mu}_{\mathbf{e}} : (\mathit{Funname} \to (\mathit{DepSet}^{\mu})^n \to \mathit{DepSet}^{\mu})\\
&\quad \to \mathit{Expression} \to \mathit{DepEnv}^{\mu} \to \mathit{DepSet}^{\mu}
\end{aligned}$$

$$\begin{aligned}
\mathcal{E}^{\downarrow}_{\mathbf{e}}\ \phi\ \llbracket c \rrbracket\ \rho &= \{\}\\
\mathcal{E}^{\downarrow}_{\mathbf{e}}\ \phi\ \llbracket x \rrbracket\ \rho &= \rho\ x\\
\mathcal{E}^{\downarrow}_{\mathbf{e}}\ \phi\ \llbracket \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 \rrbracket\ \rho &= \mathcal{E}^{\downarrow}_{\mathbf{e}}\ \phi\ \llbracket e_2 \rrbracket\ \rho \sqcap^{\downarrow} \mathcal{E}^{\downarrow}_{\mathbf{e}}\ \phi\ \llbracket e_3 \rrbracket\ \rho\\
\mathcal{E}^{\downarrow}_{\mathbf{e}}\ \phi\ \llbracket b\ e_1 \ldots e_n \rrbracket\ \rho &= \mathcal{E}^{\downarrow}_{\mathbf{b}}\ b\ (\mathcal{E}^{\downarrow}_{\mathbf{e}}\ \phi\ \llbracket e_1 \rrbracket\ \rho) \ldots (\mathcal{E}^{\downarrow}_{\mathbf{e}}\ \phi\ \llbracket e_n \rrbracket\ \rho)\\
\mathcal{E}^{\downarrow}_{\mathbf{e}}\ \phi\ \llbracket f\ e_1 \ldots e_n \rrbracket\ \rho &= \phi\ f\ (\mathcal{E}^{\downarrow}_{\mathbf{e}}\ \phi\ \llbracket e_1 \rrbracket\ \rho) \ldots (\mathcal{E}^{\downarrow}_{\mathbf{e}}\ \phi\ \llbracket e_n \rrbracket\ \rho)
\end{aligned}$$

$$\begin{aligned}
\text{where}\quad & \Delta_1 \sqcap^{\downarrow} \Delta_2 = \{\overline{\downarrow}(f_x) \mid \overline{\downarrow}(f_x) \in \Delta_1 \wedge \downarrow(f_x) \in \Delta_2\} \cup\\
& \qquad\qquad\quad \{\overline{\downarrow}(f_x) \mid \overline{\downarrow}(f_x) \in \Delta_2 \wedge \downarrow(f_x) \in \Delta_1\} \cup (\Delta_1 \cap \Delta_2)\\
\text{and}\quad & \mathcal{E}^{\downarrow}_{\mathbf{b}}\ \texttt{cons}\ \Delta_1\ \Delta_2 = \{\}\\
& \mathcal{E}^{\downarrow}_{\mathbf{b}}\ \texttt{car}\ \Delta_1 = \mathcal{E}^{\downarrow}_{\mathbf{b}}\ \texttt{cdr}\ \Delta_1 = \{\downarrow(f_x) \mid \delta(f_x) \in \Delta_1\}
\end{aligned}$$

$$\begin{aligned}
\mathcal{E}^{\uparrow}_{\mathbf{e}}\ \phi\ \llbracket c \rrbracket\ \rho &= \{\}\\
\mathcal{E}^{\uparrow}_{\mathbf{e}}\ \phi\ \llbracket x \rrbracket\ \rho &= \rho\ x\\
\mathcal{E}^{\uparrow}_{\mathbf{e}}\ \phi\ \llbracket \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 \rrbracket\ \rho &= \mathcal{E}^{\uparrow}_{\mathbf{e}}\ \phi\ \llbracket e_2 \rrbracket\ \rho \sqcup^{\uparrow} \mathcal{E}^{\uparrow}_{\mathbf{e}}\ \phi\ \llbracket e_3 \rrbracket\ \rho \sqcup^{\uparrow}\\
&\quad\ \{\uparrow(x) \mid x \in \mathit{fv}\ e_1 \wedge (\mathcal{RI}\ \llbracket e_2 \rrbracket \vee \mathcal{RI}\ \llbracket e_3 \rrbracket)\}\\
\mathcal{E}^{\uparrow}_{\mathbf{e}}\ \phi\ \llbracket b\ e_1 \ldots e_n \rrbracket\ \rho &= \mathcal{E}^{\uparrow}_{\mathbf{b}}\ b\ (\mathcal{E}^{\uparrow}_{\mathbf{e}}\ \phi\ \llbracket e_1 \rrbracket\ \rho) \ldots (\mathcal{E}^{\uparrow}_{\mathbf{e}}\ \phi\ \llbracket e_n \rrbracket\ \rho)\\
\mathcal{E}^{\uparrow}_{\mathbf{e}}\ \phi\ \llbracket f\ e_1 \ldots e_n \rrbracket\ \rho &= \phi\ f\ (\mathcal{E}^{\uparrow}_{\mathbf{e}}\ \phi\ \llbracket e_1 \rrbracket\ \rho) \ldots (\mathcal{E}^{\uparrow}_{\mathbf{e}}\ \phi\ \llbracket e_n \rrbracket\ \rho)
\end{aligned}$$

$$\begin{aligned}
\text{where}\quad & \Delta_1 \sqcup^{\uparrow} \Delta_2 = \{\uparrow(f_x) \mid \uparrow(f_x) \in \Delta_1 \cup \Delta_2\} \cup\\
& \qquad\qquad\quad \{\underline{\uparrow}(f_x) \mid \underline{\uparrow}(f_x) \in \Delta_1 \cup \Delta_2 \wedge \uparrow(f_x) \notin \Delta_1 \cup \Delta_2\}\\
\text{and}\quad & \mathcal{E}^{\uparrow}_{\mathbf{b}}\ \texttt{cons}\ \Delta_1\ \Delta_2 = \{\uparrow(f_x) \mid \delta(f_x) \in \Delta_1 \sqcup^{\uparrow} \Delta_2\}\\
& \mathcal{E}^{\uparrow}_{\mathbf{b}}\ \texttt{car}\ \Delta_1 = \mathcal{E}^{\uparrow}_{\mathbf{b}}\ \texttt{cdr}\ \Delta_1 = \Delta_1
\end{aligned}$$

Figure 5.3: Examples of safe size approximation functions

$$\mathcal{RI}\,[\![e]\!] = \exists f : \uparrow(f) \in (\mathcal{RI}_{\mathbf{e}}\; C_e \ddagger [\![e]\!])$$

where C_e is the set of functions that are mutually recursive with the function in which e occurs

$$\begin{array}{lcl}
\mathcal{RI}_{\mathbf{e}}\; C\; \delta\; [\![c]\!] & = & \{\} \\
\mathcal{RI}_{\mathbf{e}}\; C\; \delta\; [\![x]\!] & = & \{\} \\
\mathcal{RI}_{\mathbf{e}}\; C\; \delta\; [\![\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3]\!] & = & \mathcal{RI}_{\mathbf{e}}\; C\; \delta\; [\![e_2]\!] \cup \mathcal{RI}_{\mathbf{e}}\; C\; \delta\; [\![e_3]\!] \\
\mathcal{RI}_{\mathbf{e}}\; C\; \delta\; [\![b\; e_1 \ldots e_n]\!] & = & \mathcal{RI}_{\mathbf{b}}\; C\; \delta\; [\![b\; e_1 \ldots e_n]\!] \\
\mathcal{RI}_{\mathbf{e}}\; C\; \delta\; [\![f\; e_1 \ldots e_n]\!] & = & \{\delta(f) \mid f \in C\} \cup \\
 & & \mathcal{E}^{\uparrow}\; [\![e^f]\!]\; \{x_1 \mapsto \Delta_1, \ldots, x_n \mapsto \Delta_n\}, \\
 & & \text{where } \Delta_i = \mathcal{RI}_{\mathbf{e}}\; C \ddagger [\![e_i]\!] \\
\mathcal{RI}_{\mathbf{b}}\; C\; \delta\; [\![\texttt{cons}\ \ e_1\; e_2]\!] & = & \mathcal{RI}_{\mathbf{e}}\; C \uparrow [\![e_1]\!] \cup \mathcal{RI}_{\mathbf{e}}\; C \uparrow [\![e_2]\!] \\
\mathcal{RI}_{\mathbf{b}}\; C\; \delta\; [\![\texttt{car}\ \ e_1]\!] & = & \mathcal{RI}_{\mathbf{b}}\; C\; \delta\; [\![\texttt{cdr}\ \ e_1]\!] \;=\; \mathcal{RI}_{\mathbf{e}}\; C\; \delta\; [\![e_1]\!]
\end{array}$$

Figure 5.4: Operator for detecting values constructed by recursive increase

ϱ_0 is used as a "reference point": all values are compared to the values of the variables bound in ϱ_0.

One might think that the safety proof requires fixpoint induction because $\mathcal{E}^{\mu}$ are defined by fixpoint iteration, but it turns out that the property we want to prove in general does *not* hold before the fixpoint is reached (recall from Section 5.2 on page 61 that $\bot_{DepSet\downarrow}$ claims the expression in question is less than all of its free variables). Thus, the property is not stable and cannot be shown by fixpoint induction.

Clearly, simply applying total induction on e will not work either because of the fixpoint iteration. The problem is related to the fact that for totally nonterminating programs like `f x = f (x + 1)` we find, perhaps rather surprisingly, that $\mathcal{E}^{\downarrow}\; [\![\texttt{f x}]\!]\; \rho^{\downarrow}_{id} = \{\downarrow(\texttt{x})\}$. The key point is that what we want to show is in fact that *if* evaluation of e terminates, *then* $\mathcal{E}^{\mu}$ produces a correct description. What we need is some clear connection between the evaluation and approximation, and this is given by the inference system presented in Figure 5.5 on the facing page. Informally, judgement $\varrho, \rho \vdash e \hookrightarrow v : \Delta$ states

$$\frac{}{\varrho,\rho \vdash c \hookrightarrow c : \{\}} \qquad\qquad \frac{}{\varrho,\rho \vdash x \hookrightarrow \varrho\, x : \rho\, x}$$

$$\frac{\varrho,\rho \vdash e_1 \hookrightarrow \mathsf{true} : \Delta \quad \varrho,\rho \vdash e_2 \hookrightarrow v_2 : \Delta_2}{\varrho,\rho \vdash \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 \hookrightarrow v_2 : \Delta_2}$$

$$\frac{\varrho,\rho \vdash e_1 \hookrightarrow \mathsf{false} : \Delta \quad \varrho,\rho \vdash e_3 \hookrightarrow v_3 : \Delta_3}{\varrho,\rho \vdash \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 \hookrightarrow v_3 : \Delta_3}$$

$$\frac{\varrho,\rho \vdash e_1 \hookrightarrow v_1 : \Delta_1 \quad \varrho,\rho \vdash e_2 \hookrightarrow v_2 : \Delta_2}{\varrho,\rho \vdash \mathtt{cons}\ e_1\, e_2 \hookrightarrow (v_1.v_2) : \{\}}$$

$$\frac{\varrho,\rho \vdash e_1 \hookrightarrow (v_1.v_2) : \Delta \quad \downarrow\!\Delta = \{\downarrow\!(x) \mid \delta(x) \in \Delta\}}{\varrho,\rho \vdash \mathtt{car}\ e_1 \hookrightarrow v_1 : \downarrow\!\Delta}$$

$$\frac{\varrho,\rho \vdash e_1 \hookrightarrow (v_1.v_2) : \Delta \quad \downarrow\!\Delta = \{\downarrow\!(x) \mid \delta(x) \in \Delta\}}{\varrho,\rho \vdash \mathtt{cdr}\ e_1 \hookrightarrow v_2 : \downarrow\!\Delta}$$

$$\frac{\begin{array}{c}\varrho,\rho \vdash e_i \hookrightarrow v_i : \Delta_i, i = 1,\ldots,n \\ \{f_i \mapsto v_i \mid i = 1,\ldots,n\},\{f_i \mapsto \Delta_i \mid i = 1,\ldots,n\} \vdash e^f \hookrightarrow v^f : \Delta^f\end{array}}{\varrho,\rho \vdash f\, e_1 \ldots e_n \hookrightarrow v^f : \Delta^f}$$

Figure 5.5: Inference system connecting evaluation and decreasing size approximation

that if evaluating e under environment ϱ terminates, it yields v, and also that under dependency environment ρ, Δ decribes what e is decreasingly dependent on.

The following lemmas state this formally, and to prove them we use the big-step definition of evaluation from Section 3.3 on page 48. As the inference system and the big-step semantics have the same structure, the following lemma is easily checked:

Lemma 5.7
For all e, ϱ, ρ, if $\mathcal{E}$ $[\![e]\!]$ ϱ terminates without errors, then $\varrho, \rho \vdash e \hookrightarrow v : \Delta$ is well-defined and $\mathcal{E}$ $[\![e]\!]$ $\varrho = v$

Now that we have made the connection to normal evaluation, we find the required safety relation in the following

Lemma 5.8
Let ϱ_0 be given. Now for all e, ϱ, ρ, if $\mathcal{E}$ $[\![e]\!]$ ϱ terminates without errors and ρ ϱ_0-describes ϱ, then

$$\varrho, \rho \vdash e \hookrightarrow v : \Delta \text{ is well-defined and } \forall \rho' \sqsubseteq \rho : \mathcal{E}^{\downarrow} \ [\![e]\!] \ \rho' \sqsubseteq \Delta \sqsubseteq \alpha^{\downarrow}_{\varrho_0} \ v$$

Proof: As the evaluation terminates, the inference tree is finite and we prove the lemma by induction on the height of the inference tree.

Case c**:** OK.

Case x**:** We find $\mathcal{E}^{\downarrow}$ $[\![x]\!]$ $\rho' = \rho' \ x \sqsubseteq \rho \ x = \Delta$, and because ρ ϱ_0-describes ϱ we also have $\Delta = \rho \ x \sqsubseteq \alpha^{\downarrow}_{\varrho_0} \ (\varrho \ x)$.

Case if e_1 **then** e_2 **else** e_3**:** As $\Delta_1 \sqcap^{\downarrow} \Delta_2 \sqsubseteq \Delta_i$ for $i \in \{1,2\}$ we find by using the induction hypothesis that $\mathcal{E}^{\downarrow}$ $[\![\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3]\!]$ $\rho' = \mathcal{E}^{\downarrow}$ $[\![e_2]\!]$ $\rho' \sqcap^{\downarrow}$ $\mathcal{E}^{\downarrow}$ $[\![e_3]\!]$ $\rho' \sqsubseteq \mathcal{E}^{\downarrow}$ $[\![e_i]\!]$ $\rho' \sqsubseteq \Delta_i \sqsubseteq \alpha^{\downarrow}_{\varrho_0} \ v_i$ for $i = 2,3$

Case `cons`**:** OK.

Case `car` **&** `cdr`**:** As $v_1 \geq v_2 \Rightarrow \alpha^{\downarrow}_{\varrho_0} \ v_1 \sqsubseteq \alpha^{\downarrow}_{\varrho_0} \ v_2$ we see by the induction hypothesis that $\downarrow(x) \in \downarrow\Delta \Rightarrow \delta(x) \in \Delta_i \Rightarrow \delta(x) \in \alpha^{\downarrow}_{\varrho_0} \ (v_1.v_2) \Rightarrow \delta(x) \in \alpha^{\downarrow}_{\varrho_0} \ v_i$ for $i = 1,2$.

Case $f\,e_1 \ldots e_n$**:** Let $\rho^f = \{f_i \hookrightarrow \mathcal{E}^{\downarrow}\ [\![e_i]\!]\ \rho' \mid i = 1, \ldots, n\}$. By using the induction hypothesis $\mathcal{E}^{\downarrow}\ [\![e_i]\!]\ \rho' \sqsubseteq \Delta_i$ for $i = 1, \ldots, n$ we find that $\rho^f \sqsubseteq \{f_i \mapsto \Delta_i \mid i = 1, \ldots, n\}$, and using it once again we obtain $\mathcal{E}^{\downarrow}\ [\![f\,e_1 \ldots e_n]\!]\ \rho' = \mathcal{E}^{\downarrow}\ [\![e^f]\!]\ \rho^f \sqsubseteq \Delta^f \sqsubseteq \alpha^{\downarrow}_{\varrho_0}\ v^f$. Note that the use of the hypothesis hinges on the fact that ϱ_0 is fixed before the induction.

□

Finally, we are able to prove

Proposition 5.9 (Size function safety)
The size functions $\mathcal{E}^{\downarrow}$ and $\mathcal{E}^{\uparrow}$ in Figure 5.3 are safe approximations to $\mathcal{E}$.

Proof: The proof of safety for $\mathcal{E}^{\downarrow}$ is given by letting $\rho = \alpha^{\downarrow}_{\varrho}\ \varrho$ and using Lemmas 5.7 and 5.8 on the facing page.

The proof of safety for $\mathcal{E}^{\uparrow}$ is somewhat more complicated, so will content ourselves with arguing informally. Basically, we must make sure that

1. If $\mathcal{E}\ [\![e]\!]\ \varrho$ is greater than $\varrho\ x$ for infinitely many ϱ then $\uparrow(x) \in \mathcal{E}^{\uparrow}\ [\![e]\!]\ \rho_{id}$.
2. If $\mathcal{E}\ [\![e]\!]\ \varrho$ can be forced to be arbitrarily large by binding some variable x to arbitrarily large values in ϱ, then $\delta(x) \in \mathcal{E}^{\uparrow}\ [\![e]\!]\ \rho_{id}$ for some δ.

Consider first the material dependencies for the various syntactic constructions:

Case c**:** As c cannot be greater than arbitrarily large values, no dependency is needed.

Case x**:** Obvious.

Case if e_1 **then** e_2 **else** e_3**:** Material dependency concerns only e_2 and e_3, and they are handled recursively.

Case $b\,e_1 \ldots e_n$**:** Obvious.

Case $f\,e_1 \ldots e_n$**:** Obvious by the fixpoint construction.

For the immaterial dependencies, recall from Observation 5.1 on page 60 that the construction of unbounded values requires an unbounded number of function calls. Assume given some function call $f\,e_1 \ldots e_n$. For every input, either the call loops infinitely, in which case there is nothing to show, or the recursive calls are finally terminated by some stop condition.

This stop condition must be present in e_1 of some conditional. Furthermore, e_2 or e_3 must contain a recursive call, and if arbitrarily large values are to be returned, they must be constructed using increasing basic operations (i.e. `cons`) in these recursive calls. But this is exactly the case that is catered for by the $\mathcal{RI}$-construction: if there is a risk of recursive increase in any conditional branch, all variables in the stop condition are conservatively added to the dependency set. □

5.5 Composing dependencies

It is intuitively clear that we will need to calculate approximations of the effect of composing size dependencies from the individual function calls.

We therefore also need an associative dependency combinator $\Box^{\mu}$: $D^{\mu} \times D^{\mu} \to D^{\mu}$ with unit element, ι^{μ}. This combinator is used to calculate the dependency effect of composing two expressions, so intuitively $\delta_1 \Box^{\mu} \delta_2$ should yield a dependency operator that gives a safe description of how h_z relates to f_x if g_y is $\delta_1(f_x)$ and h_z is $\delta_2(g_y)$. Given $\Box^{\mu}$ and the dependencies along simple call paths in the program, the algorithms can compute the total effect of dependencies along all loops in the program. $\mathcal{E}^{\downarrow}$ and $\Box^{\downarrow}$ must be compatible in the sense that for all $e_1, e_2, \varrho_1, \varrho_2$ they must satisfy the condition:

$$\begin{aligned}
&\text{if } \delta_1(f_x) \in \mathcal{E}^{\downarrow}\ [\![e_1]\!]\ (\alpha^{\downarrow}_{\varrho_1}\ \varrho_1) \text{ and } \delta_2(g_y) \in \mathcal{E}^{\downarrow}\ [\![e_2]\!]\ (\alpha^{\downarrow}_{\varrho_2}\ \varrho_2)\\
&\text{then } \exists \delta(f_x) \in \mathcal{E}^{\downarrow}\ [\![e_2]\!]\ (\alpha^{\downarrow}_{\varrho_2'}\ \varrho_2') : \delta_1 \Box^{\downarrow} \delta_2 \sqsubseteq \delta,\\
&\text{where } \varrho_2' = \varrho_2 + \{g_y \mapsto \mathcal{E}\ [\![e_1]\!]\ \varrho_1\}
\end{aligned}$$

Similarly, for all $e_1, e_2, \varrho_1, \varrho_2$, $\mathcal{E}^{\uparrow}$ and $\Box^{\uparrow}$ must satisfy

$$\begin{aligned}
&\text{if } \delta_1(f_x) \in \mathcal{E}^{\uparrow}\ [\![e_1]\!]\ (\alpha^{\uparrow}_{\varrho_1}\ \varrho_1) \text{ and } \delta_2(g_y) \in \mathcal{E}^{\uparrow}\ [\![e_2]\!]\ (\alpha^{\uparrow}_{\varrho_2}\ \varrho_2)\\
&\text{then } \exists \delta(f_x) \in \mathcal{E}^{\uparrow}\ [\![e_2]\!]\ (\alpha^{\uparrow}_{\varrho_2'}\ \varrho_2') : \delta \sqsubseteq \delta_1 \Box^{\uparrow} \delta_2,\\
&\text{where } \varrho_2' = \varrho_2 + \{g_y \mapsto \mathcal{E}\ [\![e_1]\!]\ \varrho_1\}
\end{aligned}$$

Composing dependencies should also satisfy the following

Requirement 5.10 (Substitution for $\mathcal{E}^{\uparrow}$ and $\mathcal{E}^{\downarrow}$)
Given expression e and $\{y_1, \ldots, y_m\} \supseteq fv\ e$, then

$$\begin{array}{l} \delta(y) \in \mathcal{E}^{\uparrow} \llbracket [y_1 \mapsto e_1, \ldots, y_n \mapsto e_n]e \rrbracket\ \rho^{\uparrow}_{id} \ \Rightarrow \\ \exists i : \delta'(y) \in \mathcal{E}^{\uparrow} \llbracket e_i \rrbracket\ \rho^{\uparrow}_{id} \wedge \delta''(y_i) \in \mathcal{E}^{\uparrow} \llbracket e \rrbracket\ \rho^{\uparrow}_{id}, \end{array}$$

where $\delta \sqsubseteq \delta' \Box^{\uparrow} \delta''$, and

$$\begin{array}{l} \delta(y) \in \mathcal{E}^{\downarrow} \llbracket [y_1 \mapsto e_1, \ldots, y_n \mapsto e_n]e \rrbracket\ \rho^{\downarrow}_{id} \ \Leftarrow \\ \exists i : \delta'(y) \in \mathcal{E}^{\downarrow} \llbracket e_i \rrbracket\ \rho^{\downarrow}_{id} \wedge \delta''(y_i) \in \mathcal{E}^{\downarrow} \llbracket e \rrbracket\ \rho^{\downarrow}_{id} \end{array}$$

where $\delta \sqsupseteq \delta' \Box^{\downarrow} \delta''$.

For our size dependencies we can let composition be given by the following

Definition 5.11 (Dependency composition)
Let $↧ \Box^{\downarrow} \delta = \delta \Box^{\downarrow} ↧ = \delta$, $\downarrow \Box^{\downarrow} \downarrow = \downarrow$, $⇞ \Box^{\uparrow} \delta = \delta \Box^{\uparrow} ⇞ = \delta$, $\uparrow \Box^{\uparrow} \uparrow = \uparrow$, $\iota^{\downarrow} = ↧$ and $\iota^{\uparrow} = ⇞$

It is straightforward to see that $\Box^{\mu}$ is compatible with $\mathcal{E}^{\mu}$ of Figure 5.3, and the following lemma can be proved by fixpoint induction:

Lemma 5.12
The size functions $\mathcal{E}^{\uparrow}$ and $\mathcal{E}^{\downarrow}$ in Figure 5.3 satisfy Requirement 5.10

It is quite possible to imagine other, more elaborate size dependency functions which can give better approximations for the return values of user-defined functions. Even type-system based size dependency functions similar to those suggested by Hughes et al. (1996) could be employed, as long as they satisfy the safety conditions. More specific dependency operators could also be used; the choice of how detailed they are and the choice of $\Box^{\mu}$ affects the complexity and computability properties of the semi-ring algorithm used in Section 6.5 on page 85.

Chapter 6

Determining Boundedness of Parameters

6.1 Bounded variation

Recall the Definition 3.8 on page 45 of *bounded variation:* a function parameter is said to be of BV when the set of possible values it can take during any program evaluation is finite:

$$\boxed{f_i \text{ is BV iff } \forall \vec{u} \in \mathit{Value}^a : \{v_i \mid (\mathit{f1}, \vec{u}) \to^* (f, \vec{v})\} \text{ is finite,}}$$

where $a = \mathit{arity}\ \mathit{f1}$.

The BV property is in general undecidable, so the object of this section is to show how two conditions called *domination* and *anchoring* conditions can detect a "large subset" of p's BV parameters. The domination condition detects parameters whose values will always be dominated by one of p's input parameters, one of p's program constants, or a bounded combination of BV parameters. The anchoring condition is more general, detecting also parameters whose value *growth* is limited by the value of some other BV parameter.

The conditions operate on graph representations of the parameter dependencies in the program. We start by constructing two *size dependency graphs* from the program: SDG$^{\uparrow}$ and SDG$^{\downarrow}$. The nodes represent the function parameters in p, while the edges represent the dependencies arising from the arguments used at the various call sites. The SDG$^{\uparrow}$ records increasing and non-bounding properties and is used to detect domination, while the SDG$^{\downarrow}$ records decreasing and non-increasing properties. When a potentially diverging loop is detected in the SDG$^{\uparrow}$, the SDG$^{\downarrow}$ is searched for a corresponding, decreasing bounded loop that can act as an "anchor." This is in fact done by extracting information about loops from the SDG$^{\uparrow}$ and SDG$^{\downarrow}$ graphs and then building a loop dependency graph, LDG. It is by examining this LDG that the relationship between diverging loops and their anchors is detected.

Example 6.1
For the program shown in Figure 6.1, to ease readability we assume we are working with the natural numbers as our well-founded *Value* domain. The program does not compute anything sensible, but it illustrates some non-trivial program constructs. We have labeled each function call site with a letter *a*–*g* for later reference.

The size dependencies in this program are:

e	$\mathcal{E}^{\uparrow}\,[\![e]\!]\,\rho^{\uparrow}_{id}$	$\mathcal{E}^{\downarrow}\,[\![e]\!]\,\rho^{\downarrow}_{id}$
`f x y z d`	{↑(x)}	{}
`g u v w`	{↑(u)}	{}
`h r s`	{⇞(r)}	{}
`dec n`	{⇞(n)}	{↓(n)}
`(`**if** `h`^{e}`v 0 w = 0` **then** `v` **else** `dec v)`	{⇞(v)}	{⇟(v)}

where $\rho^{\mu}_{id}\ x = \{\iota^{\mu}(x)\}$

The conditions to be presented work by assigning *property marks* to the nodes of these graphs. These marks range over a simple domain with two values: $\perp \sqsubseteq B$, and the algorithms are constructed in such a way that the property mark of a node never decreases. Initially all nodes are marked $\perp$ (the node has not been reached yet). Subsequently, they can change to B (of Bounded variation).

It is important to note the difference between the *property* BV and the *mark* '*B*': the marks are approximations to bounded variation: a BV variable might

```
contrived ≡
  f1 a b    = f^a a (a + 2) a b;

  f x y z d = if z > 0 ∧ d > 0 then
                f^b (x + 1) z (z - 1) (d - 1)
              else
                if y > 0 then g^c x (y - 1) d else x;

  g u v w   = if w > 0 then
                f^d (u + 1) (if h^e v 0 = 0 then v else dec v)
                   (v - 1) (w - 1)
              else
                u

  h r s     = if r > 0 then
                h^f (r - 1) 42
              else
                if s > 0 then h^g r (s - 1) else r

  dec n     = n - 1
```

Figure 6.1: An example program with call site labels *a*–*g*

not be marked B. However, the marks will be seen to be safe in the sense that any variable marked B *is* in fact BV.

In the following we will assume that all functions of p have been extended with a call depth parameter `dp` as described in the introduction on page 15.

6.2 Detecting BV by domination

The SDG$^{\uparrow}$ is a graph $(V, E_{\mathcal{S}})$, where $V \subseteq$ *Varname* is the set of variable names in p, $E_{\mathcal{S}}$ is a set of labeled edges defined by

$$\begin{aligned} E_{\mathcal{S}} &= \mathcal{SD}_{\mathbf{e}} [\![e^{f1}]\!] \, \rho_{id}^{\uparrow} \cup \cdots \cup \mathcal{SD}_{\mathbf{e}} [\![e^{fn}]\!] \, \rho_{id}^{\uparrow} \\ \rho_{id}^{\mu} \, x &= \{\iota^{\mu}(x)\}, \end{aligned}$$

and operator $\mathcal{SD}_{\mathbf{e}}$ is given in Figure 6.2. The superscript s in rule (∇'_5) is

$$\begin{array}{ll}
(\nabla'_1)\ \mathcal{SD}_{\mathbf{e}}\ [\![c]\!]\ \rho & = \{\} \\
(\nabla'_2)\ \mathcal{SD}_{\mathbf{e}}\ [\![x]\!]\ \rho & = \{\} \\
(\nabla'_3)\ \mathcal{SD}_{\mathbf{e}}\ [\![\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3]\!]\ \rho & = \mathcal{SD}_{\mathbf{e}}\ [\![e_1]\!]\ \rho \cup \mathcal{SD}_{\mathbf{e}}\ [\![e_2]\!]\ \rho \cup \mathcal{SD}_{\mathbf{e}}\ [\![e_3]\!]\ \rho \\
(\nabla'_4)\ \mathcal{SD}_{\mathbf{e}}\ [\![b\ e_1 \ldots e_n]\!]\ \rho & = \mathcal{SD}_{\mathbf{e}}\ [\![e_1]\!]\ \rho \cup \cdots \cup \mathcal{SD}_{\mathbf{e}}\ [\![e_n]\!]\ \rho \\
(\nabla'_5)\ \mathcal{SD}_{\mathbf{e}}\ [\![f^s\ e_1 \ldots e_n]\!]\ \rho & = \mathcal{SD}_{\mathbf{e}}\ [\![e_1]\!]\ \rho \cup \cdots \cup \mathcal{SD}_{\mathbf{e}}\ [\![e_n]\!]\ \rho \cup \\
 & \quad \{g_y \xrightarrow{\delta} f_1 \mid \delta(g_y) \in \mathcal{E}^{\uparrow}\ [\![e_1]\!]\ \rho\} \cup \cdots \cup \\
 & \quad \{g_y \xrightarrow{\delta} f_n \mid \delta(g_y) \in \mathcal{E}^{\uparrow}\ [\![e_n]\!]\ \rho\}
\end{array}$$

Figure 6.2: Size dependency graph (SDG$^{\uparrow}$) edge generating operator $\mathcal{SD}_{\mathbf{e}}$

the call site number. The SDG$^{\uparrow}$ of Example 6.1 is shown in Figure 6.3; for readability we have omitted labels on edges labeled $\ddagger$. The dashed boxes enclose strongly connected components.

We then try to detect some parameters that in any computation will always be less than (i.e. dominated by) one of p's input parameters, a program constant given in p, or a bounded combination of BV parameters.

First, we mark program input as 'B' (recall that we assume wlog. that the goal function *f1* is not called from within the program, so the nodes representing program input will be trivial singleton SCCs with no loops). To detect further BV parameters, we first break up the SDG$^{\uparrow}$ into a DAG of strongly connected components (SCCs) and sort them topologically. As nodes in the same SCC are all mutually reachable, they must always carry the same mark, so we will use the term "the mark of C" synonymously with "marks of all the nodes in SCC C." We now apply the following condition in topological order to each SCC which is not already marked as B:

Condition 6.2 (Bounded domination)
If *all predecessors of SCC $\mathcal{C}$ are marked B,* and *$\mathcal{C}$ contains no $\uparrow$-labeled edges,* then *mark $\mathcal{C}$ as B.*

In the example of Figure 6.3, SCC's containing more than one node are indicated by dashed boxes; nodes `a`, `b`, `d`, `w`, $\texttt{f1}_{\texttt{dp}}$, $\texttt{dec}_{\texttt{dp}}$, `y`, `z`, `v`, `r`, and `s` can be

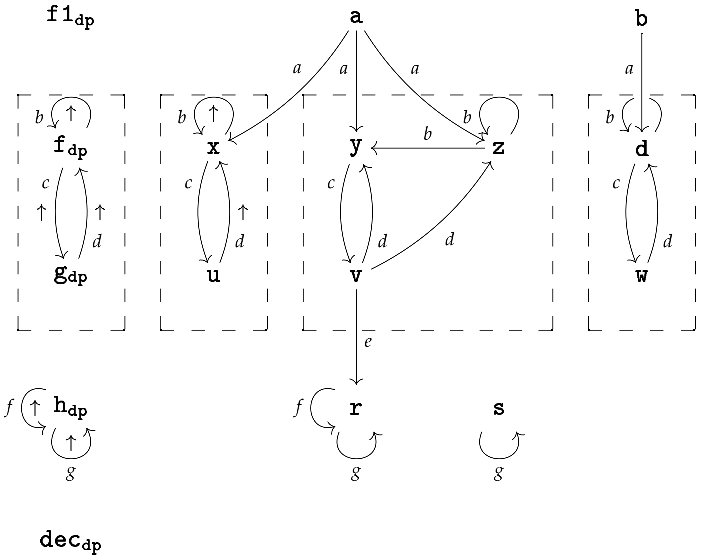


Figure 6.3: Increasing size dependency graph (SDG$^{\uparrow}$) of Example 6.1

marked as B by the domination condition.

The nodes in the SDG$^{\uparrow}$ thus marked B do not participate in any "dangerous" loops where values can increase, and they obtain their values along edges from other B-marked parameters, so they are of BV.

We can now show

Theorem 6.3 (Soundness of bounded domination)
If all variables of p whose corresponding nodes are marked as B are BV, then all variables of p whose corresponding nodes get marked as B by the domination condition are in fact of BV.

Proof: The soundness of this condition's markings is shown in the following section. Informally its correctness relies on the fact that unbounded variables must either receive values from other unbounded variables or participate in an increasing loop. □

6.3 Detecting BV by anchoring

The domination condition can detect many BV parameters, but fails to detect parameters that are BV by virtue of being restricted to grow only by the *number of times* some other BV parameter can be decreased. This is what the *bounded anchoring* condition is intended to detect.

Before we start detecting these restricted BV parameters, conceptually we first construct the graph representing the restricting properties in p, the decreasing size dependency graph, SDG$^{\downarrow}$. Its construction is similar to that of the SDG$^{\uparrow}$, except that all occurrences of $\mathcal{E}^{\uparrow}$ are replaced by $\mathcal{E}^{\downarrow}$:

$$E_{\mathcal{S}^{\downarrow}} \quad = \quad \mathcal{SD}_{\mathbf{e}}\ [\![e^{f1}]\!]\ \rho^{\downarrow}_{id} \cup \cdots \cup \mathcal{SD}_{\mathbf{e}}\ [\![e^{fn}]\!]\ \rho^{\downarrow}_{id}$$

and $\mathcal{SD}_{\mathbf{e}}$ is given in Figure 6.4. The SDG$^{\downarrow}$ for Example 6.1 can be seen in Figure 6.5; for readability, all ↧-marks and `dp` nodes have been omitted.

$$\begin{array}{lcl}
\mathcal{SD}_{\mathbf{e}}\ [\![c]\!]\ \rho & = & \{\} \\
\mathcal{SD}_{\mathbf{e}}\ [\![x]\!]\ \rho & = & \{\} \\
\mathcal{SD}_{\mathbf{e}}\ [\![\textbf{if}\ e_1\ \textbf{then}\ e_2\,\textbf{else}\ e_3]\!]\ \rho & = & \mathcal{SD}_{\mathbf{e}}\ [\![e_1]\!]\ \rho \cup \mathcal{SD}_{\mathbf{e}}\ [\![e_2]\!]\ \rho \cup \mathcal{SD}_{\mathbf{e}}\ [\![e_3]\!]\ \rho \\
\mathcal{SD}_{\mathbf{e}}\ [\![b\ e_1 \ldots e_n]\!]\ \rho & = & \mathcal{SD}_{\mathbf{e}}\ [\![e_1]\!]\ \rho \cup \cdots \cup \mathcal{SD}_{\mathbf{e}}\ [\![e_n]\!]\ \rho \\
\mathcal{SD}_{\mathbf{e}}\ [\![f\ e_1 \ldots e_n]\!]\ \rho & = & \mathcal{SD}_{\mathbf{e}}\ [\![e_1]\!]\ \rho \cup \cdots \cup \mathcal{SD}_{\mathbf{e}}\ [\![e_n]\!]\ \rho \cup \\
 & & \{g_y \xrightarrow{\delta} f_1 \mid \delta(g_y) \in \mathcal{E}^{\downarrow}\ [\![e_1]\!]\ \rho\} \cup \cdots \cup \\
 & & \{g_y \xrightarrow{\delta} f_n \mid \delta(g_y) \in \mathcal{E}^{\downarrow}\ [\![e_n]\!]\ \rho\}
\end{array}$$

Figure 6.4: Size dependency graph (SDG$^{\downarrow}$) edge generating operator

Each SCC of the SDG$^{\uparrow}$ will be internally connected by a number of loops, where a loop $\ell = f_x \longrightarrow \cdots \longrightarrow f_x$ is a cyclic path, *possibly containing the same edge several times*. If we write $g_y \xrightarrow{\delta,s} f_x$ to indicate that the edge represents a parameter dependency at call site s, we define a *sibling loop* of a loop $\ell =$

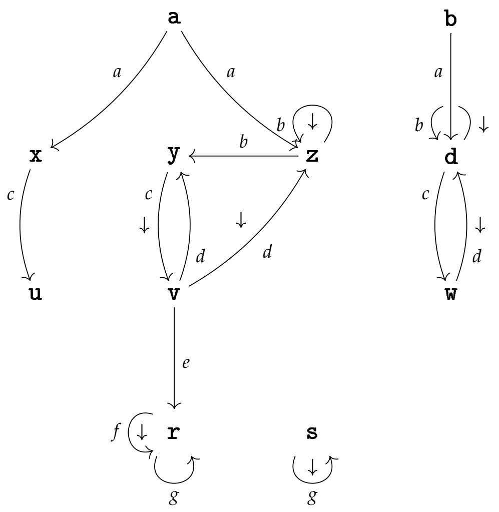


Figure 6.5: Decreasing size dependency graph (SDG$^{\downarrow}$) of Example 6.1

$f_x \xrightarrow{\delta_1,s_1} \cdots \xrightarrow{\delta_n,s_n} f_x$ to be another loop in the SDG$^{\downarrow}$, $\ell' = f_y \xrightarrow{\delta'_1,s_1} \cdots \xrightarrow{\delta'_n,s_n} f_y$ with an identical list of call sites $s_1, \ldots, s_n$. As a couple of examples,

$\mathtt{y} \xrightarrow{\downarrow,c} \mathtt{v} \xrightarrow{\overset{=}{\downarrow},d} \mathtt{y}$ from Figure 6.5 is a sibling loop of

$\mathtt{f}_{\mathtt{dp}} \xrightarrow{\uparrow,c} \mathtt{g}_{\mathtt{dp}} \xrightarrow{\uparrow,d} \mathtt{f}_{\mathtt{dp}}$ in Figure 6.3 and

$\mathtt{y} \xrightarrow{\downarrow,c} \mathtt{v} \xrightarrow{\downarrow,d} \mathtt{z} \xrightarrow{\overset{=}{\downarrow},b} \mathtt{y}$ is a sibling loop of

$\mathtt{x} \xrightarrow{\underset{=}{\uparrow},c} \mathtt{u} \xrightarrow{\uparrow,d} \mathtt{x} \xrightarrow{\uparrow,b} \mathtt{x}.$

Note that all parameters of an SDG$^{\uparrow}$ loop have the same mark as they are all in the same SCC.

We now extract some information about loops in SDG$^{\uparrow}$ and their sibling loops in the SDG$^{\downarrow}$; this is done by extending the SDG$^{\uparrow}$ edge generating function $\mathcal{SD}_{\mathbf{e}}$ of Figure 6.2 to generate edges representing a combination of SDG$^{\uparrow}$ and SDG$^{\downarrow}$ edges. This new graph is simply called SDG, and consists of the same nodes as the SDG$^\uparrow$ and SDG$^\downarrow$ graphs. Instead of consisting of only simple edges $g_y \xrightarrow{\delta} f_x$ like the SDG$^\uparrow$, the SDG contains sibling annotated edges (SA-edges) $g_y \xrightarrow[S]{\delta} f_x \in E_{\mathcal{S}}$, where $S = \{g_v \xrightarrow{\delta'} f_u, \ldots\}$ is a set of sibling edges in SDG$^\downarrow$. In other words, each edge is labelled with its increasing effect from the SDG$^\uparrow$ and the set of sibling edges in the SDG$^\downarrow$ along with their decreasing effects; we will sometimes write an SA-edge as a pair $(g_y \xrightarrow{\delta} f_x, S)$.

The SA-edges are generated by similar $\mathcal{SD}_{\mathbf{e}}$ rules (∇_1)–(∇_4) using two dependency environments ($\rho^\uparrow$ and $\rho^\downarrow$) and replacing rule (∇_5') with

$$
\begin{aligned}
(\nabla_5)\quad & \mathcal{SD}_{\mathbf{e}}\ \llbracket f^s e_1 \ldots e_n \rrbracket\ \rho^\uparrow\ \rho^\downarrow \\
&= \mathcal{SD}_{\mathbf{e}}\ \llbracket e_1 \rrbracket\ \rho^\uparrow\ \rho^\downarrow \cup \cdots \cup \mathcal{SD}_{\mathbf{e}}\ \llbracket e_n \rrbracket\ \rho^\uparrow\ \rho^\downarrow \cup \\
&\quad \left\{ g_y \xrightarrow[S]{\delta} f_1 \;\middle|\; \delta(g_y) \in \mathcal{E}^\uparrow\ \llbracket e_1 \rrbracket\ \rho^\uparrow \right\} \cup \cdots \cup \\
&\quad \left\{ g_y \xrightarrow[S]{\delta} f_n \;\middle|\; \delta(g_y) \in \mathcal{E}^\uparrow\ \llbracket e_n \rrbracket\ \rho^\uparrow \right\} \\
& \text{where } S = \{ g_z \xrightarrow{\delta'} f_j \mid \delta'(g_z) \in \mathcal{E}^\downarrow\ \llbracket e_j \rrbracket\ \rho^\downarrow \}
\end{aligned}
$$

The SDG is a faithful representation of the call paths of p—this is captured by the following

Proposition 6.4 (Call path and SDG path correspondence)
Assume there exists a call path $\pi = \lfloor f^1\ \vec{e}^1\ \lfloor \ldots \lfloor f^m\ \vec{e}^m \rrbracket$ in $L(G)$.

1. *If $f^m_{i_k}$ depends on $x \in fv\ \vec{e}^1$ along π with effect δ then there exists a path $x \xrightarrow[S_1]{\delta_1} f^1_{i_1} \xrightarrow[S_2]{\delta_2} \cdots \xrightarrow[S_k]{\delta_k} f^k_{i_k}$ in the SDG where $\delta \sqsubseteq \delta_1 \square^\uparrow \cdots \square^\uparrow \delta_k$ and $f^k = f^m$.*

2. *If further there exist sibling edges $y \xrightarrow{\delta_1'} f^1_{j_1} \in S_1$, $f^1_{j_1} \xrightarrow{\delta_2'} f^2_{j_2} \in S_2$, $\ldots$, $f^{k-1}_{j_{k-1}} \xrightarrow{\delta_k'} f^k_{j_k} \in S_k$, then $f^k_{j_k}$ depends on y along π with effect δ', where $\delta_1' \square^\downarrow \cdots \square^\downarrow \delta_{k-1}' \sqsubseteq \delta'$.*

Note that the length of $\pi^\flat$ might *not* equal the length of the SDG path: π includes nested calls, whereas the effect of these is represented as edge labels in the SDG path.
Proof: by structural induction on π.

Case $\pi = [f\, e_1 \ldots e_n]$**:** $k = 1$, so assume $f^1_{i_1}$ depends on x along π with effect δ, i.e. $\delta(x) \in \mathcal{E}^\uparrow \llbracket (st^\pi)_{i_1} \rrbracket\ \rho^\uparrow_{id} = \mathcal{E}^\uparrow \llbracket e_{i_1} \rrbracket\ \rho^\uparrow_{id}$.

1. This means that rule (∇_5) generates an edge $x \xrightarrow[S_1]{\delta} f^1_{i_1}$.

2. Furter, by construction of S_1, if $y \xrightarrow{\delta'} f^1_{j_1} \in S_1$ it must be because $\delta'(y) \in \mathcal{E}^\downarrow \llbracket e_{j_1} \rrbracket\ \rho^\downarrow_{id} = \mathcal{E}^\downarrow \llbracket (st^\pi)_{j_1} \rrbracket\ \rho^\downarrow_{id}$, i.e. $f^1_{j_1}$ depends on y along π with effect δ'.

Case $\pi = [f\, e_1 \ldots e_n\ \pi_1 \ldots \pi_l]$**:** Assume $f^k_{i_k}$ depends on x along π with effect δ, i.e.

$$\delta(x) \in \mathcal{E}^\uparrow \llbracket (st^\pi)_{i_k} \rrbracket\ \rho^\uparrow_{id} = \mathcal{E}^\uparrow \llbracket [f^1_1 \mapsto e_1, \ldots, f^1_n \mapsto e_n](st^{\pi_l})_{i_k} \rrbracket\ \rho^\uparrow_{id}.$$

1. By the Substitution Requirement 5.10 there exists an i_1 such that $\delta_1(x) \in \mathcal{E}^\uparrow \llbracket e_{i_1} \rrbracket\ \rho^\uparrow_{id}$ and $\delta^\bullet(f^1_{i_1}) \in \mathcal{E}^\uparrow \llbracket (st^{\pi_l})_{i_k} \rrbracket\ \rho^\uparrow_{id}$ with $\delta \sqsubseteq \delta_1 \mathbin{\Box^\uparrow} \delta^\bullet$. First, this implies that applying (∇_5) to the expression $f\, e_1 \ldots e_n$ produces the edge $x \xrightarrow[S_1]{\delta_1} f^1_{i_1}$. Second, it implies that $f^k_{i_k}$ depends on $f^1_{i_1}$ along π_l with effect $\delta^\bullet$, so by the induction hypothesis there exists a path $f^1_{i_1} \xrightarrow[S_2]{\delta_2} \cdots \xrightarrow[S_k]{\delta_k} f^k_{i_k}$ with $\delta^\bullet \sqsubseteq \delta_2 \mathbin{\Box^\uparrow} \cdots \mathbin{\Box^\uparrow} \delta_k$. This proves the existence of the required path.

2. Now assume $y \xrightarrow{\delta'_1} f^1_{j_1} \in S_1, f^1_{j_1} \xrightarrow{\delta'_2} f^2_{j_2} \in S_2, \ldots, f^{k-1}_{j_{k-1}} \xrightarrow{\delta'_k} f^k_{j_k} \in S_k$. By the induction hypothesis $f^k_{j_k}$ depends on $f^1_{j_1}$ along π_l with effect δ° where $\delta'_2 \mathbin{\Box^\downarrow} \cdots \mathbin{\Box^\downarrow} \delta'_k \sqsubseteq \delta^\circ$, i.e. $\delta^\circ(f^1_{j_1}) \in \mathcal{E}^\downarrow \llbracket (st^{\pi_l})_{j_k} \rrbracket\ \rho^\downarrow_{id}$. We know that $y \xrightarrow{\delta'_1} f^1_{j_1} \in S_1$, so by construction of S_1 we have $\delta'_1(y) \in \mathcal{E}^\downarrow \llbracket e_{j_1} \rrbracket\ \rho^\downarrow_{id}$. The Substitution Requirement now yields

$$\delta'(y) \in \mathcal{E}^\downarrow \llbracket [f^1_1 \mapsto e_1, \ldots, f^1_n \mapsto e_n](st^{\pi_l})_{j_k} \rrbracket\ \rho^\downarrow_{id} = \mathcal{E}^\downarrow \llbracket (st^\pi)_{j_k} \rrbracket\ \rho^\downarrow_{id},$$

where $\delta_1' \sqcap^{\downarrow} \cdots \sqcap^{\downarrow} \delta_k' = \delta_1' \sqcap^{\downarrow} \delta^{\circ} \sqsubseteq \delta'$. In other words, $f_{j_k}^{k}$ depends on y along π with effect δ'.

□

Our aim is now to obtain summary information describing all loops. This can be done by using an algorithm for computing the "sum of products" for "costs" along all paths (Aho, Hopcroft, & Ullman, 1975) of each SCC. To be able to use the algorithm, we must supply a "cost" for each edge, and product and sum operators that together make up a semi-ring structure. We let $Cost = D^{\uparrow} \times \mathcal{P}(V \times D^{\downarrow} \times V)$ be the set of SA-edge labels and introduce the following

Definition 6.5 (Semi-ring loop cost structure)
Define the cost labelling function $l : E_{\mathcal{S}} \rightarrow Cost$ *by* $l\left(g_y \xrightarrow[S]{\delta} f_x\right) = (\delta, S)$ *and multiplication* $\otimes : (Cost \times Cost) \rightarrow Cost$ *by*

$$\gamma_1 \otimes \gamma_2 = \bigcup_{(\delta_1, S_1) \in \gamma_1} \bigcup_{(\delta_2, S_2) \in \gamma_2} \{(\delta_1 \sqcap^{\uparrow} \delta_2, S_1 \circ S_2)\},$$

$$\textit{where } S_1 \circ S_2 = \left\{ h_w \xrightarrow{\delta_1' \sqcap^{\downarrow} \delta_2'} f_u \;\middle|\; \exists g_v : h_w \xrightarrow{\delta_1'} g_v \in S_1 \wedge g_v \xrightarrow{\delta_2'} f_u \in S_2 \right\},$$

and finally $\mathbf{1} = \left\{ \left(\pm, \{ f_x \xrightarrow{\mp} f_x \mid f_x \in p \} \right) \right\}$.

Lemma 6.6
The components in Definition 6.5 define a closed semi-ring structure $\mathbb{S} = (Cost, \cup, \otimes, \emptyset, \mathbf{1})$.

Proof:

- $(\mathbb{S}, \cup, \emptyset)$ is a monoid: It is closed under $\cup$ (there are no restrictions on the elements of *Cost*), $\cup$ is associative and has identity element $\emptyset$.
- $(\mathbb{S}, \otimes, \mathbf{1})$ is a monoid: It is closed under $\otimes$, and due to the associativity of $\sqcap^{\downarrow}$ we have

$$
\begin{aligned}
&(S_1 \circ S_2) \circ S_3 \\[1em]
&= \left\{ k_r \xrightarrow{\delta_{12} \Box^{\downarrow} \delta_3} f_u \;\middle|\; \exists g_v : k_r \xrightarrow{\delta_{12}} g_v \in (S_1 \circ S_2) \wedge g_v \xrightarrow{\delta_3} f_u \in S_3 \right\} \\[1em]
&= \left\{ k_r \xrightarrow{\delta_{12} \Box^{\downarrow} \delta_3} f_u \;\middle|\; \begin{array}{c} \exists g_v : k_r \xrightarrow{\delta_{12}} g_v \in \left\{ k_r \xrightarrow{\delta_1 \Box^{\downarrow} \delta_2} g_v \;\middle|\; \begin{array}{r} \exists h_w : k_r \xrightarrow{\delta_1} h_w \in S_1 \wedge \\ h_w \xrightarrow{\delta_2} g_v \in S_2 \end{array} \right\} \\ \wedge\, g_v \xrightarrow{\delta_3} f_u \in S_3 \end{array} \right\} \\[1em]
&= \left\{ k_r \xrightarrow{\delta_1 \Box^{\downarrow} \delta_2 \Box^{\downarrow} \delta_3} f_u \;\middle|\; \begin{array}{r} \exists g_v \exists h_w : k_r \xrightarrow{\delta_1} h_w \in S_1 \wedge \qquad\qquad\qquad \\ h_w \xrightarrow{\delta_2} g_v \in S_2 \wedge g_v \xrightarrow{\delta_3} f_u \in S_3 \end{array} \right\} \\[1em]
&= \left\{ k_r \xrightarrow{\delta_1 \Box^{\downarrow} \delta_{23}} f_u \;\middle|\; \begin{array}{l} \exists h_w : k_r \xrightarrow{\delta_1} h_w \in S_1 \wedge \\ \quad h_w \xrightarrow{\delta_{23}} f_u \in \left\{ h_w \xrightarrow{\delta_2 \Box^{\downarrow} \delta_3} f_u \;\middle|\; \begin{array}{r} \exists g_v : h_w \xrightarrow{\delta_2} g_v \in S_2 \wedge \\ g_v \xrightarrow{\delta_3} f_u \in S_3 \end{array} \right\} \end{array} \right\} \\[1em]
&= \left\{ k_r \xrightarrow{\delta_1 \Box^{\downarrow} \delta_{23}} f_u \;\middle|\; \exists h_w : k_r \xrightarrow{\delta_1} h_w \in S_1 \wedge h_w \xrightarrow{\delta_{23}} f_u \in (S_2 \circ S_3) \right\} \\[1em]
&= S_1 \circ (S_2 \circ S_3).
\end{aligned}
$$

This, together with the associativity of $\Box^{\uparrow}$ shows that $\otimes$ is associative, and as ↥ and ↧ are identity elements for $\Box^{\uparrow}$ and $\Box^{\downarrow}$, $\mathbf{1}$ is an identity element for $\otimes$.

- $\emptyset$ is an annihilator for $\otimes$, i.e. $\gamma \otimes \emptyset = \emptyset \otimes \gamma = \emptyset$.

- $\cup$ obviously commutes and is idempotent.

- $\otimes$ distributes over $\cup$:

$$\begin{aligned}
\gamma_1 \otimes (\gamma_2 \cup \gamma_3) &= \bigcup_{(\delta_1,S_1)\in\gamma_1} \bigcup_{(\delta_{23},S_{23})\in(\gamma_2\cup\gamma_3)} \{(\delta_1 \square^{\uparrow} \delta_{23}, S_1 \circ S_{23})\} \\
&= \left(\bigcup_{(\delta_1,S_1)\in\gamma_1} \bigcup_{(\delta_2,S_2)\in\gamma_2} \{(\delta_1 \square^{\uparrow} \delta_2, S_1 \circ S_2)\}\right) \cup \\
&\quad \left(\bigcup_{(\delta_1,S_1)\in\gamma_1} \bigcup_{(\delta_3,S_3)\in\gamma_3} \{(\delta_1 \square^{\uparrow} \delta_3, S_1 \circ S_3)\}\right) \\
&= (\gamma_1 \otimes \gamma_2) \cup (\gamma_1 \otimes \gamma_3)
\end{aligned}$$

  and similarly $(\gamma_1 \cup \gamma_2) \otimes \gamma_3 = (\gamma_1 \otimes \gamma_2) \cup (\gamma_1 \otimes \gamma_3)$.

- We are only using properties of set unions and products, so this property also holds for infinite sequences: $(\cup_i \gamma_i) \otimes (\cup_j \gamma'_j) = \cup_{i,j}(\gamma_i \otimes \gamma'_j)$

- $\cup$ is associative over infinite sequences.

□

Using (an adapted version of) the semi-ring algorithm we compute for each node f_x in each SCC a "total cost" $C(f_x)$ describing size changing effects along all loops from f_x to f_x and their sibling loop effects. Some of the total costs for Example 6.1 can be seen in Figure 6.6.

From this information we build a loop dependency graph, LDG, by computing all the total costs $\bigcup_{f_x \in p} C(f_x)$. For each cost $(\uparrow, S) \in C(f_x)$ we create a *balloon node* $v_{balloon}$ representing the increasing loop $f_x \xrightarrow{\uparrow}{}^{*} f_x$ and for each decreasing loop $f_y \xrightarrow{\downarrow}{}^{*} f_y \in S$ an *anchor node* v_{anchor} representing $f_y \xrightarrow{\downarrow}{}^{*} f_y$, and a *chain edge* between $v_{balloon}$ and v_{anchor}. Each balloon and anchor node always carries the same mark ($\perp$ or B) as the parameter it represents.

Now we can detect some BV parameters by applying the following condition in topological order to each SCC C of the SDG$^{\uparrow}$ which is not already marked as B:

Condition 6.7 (Bounded anchoring)
If *all predecessors of C are marked B* and *all balloon nodes in the LDG containing a node from C are anchored in (have a chain edge from) an anchor node marked B,* then *mark C as B.*

$$
\begin{aligned}
C(\mathtt{x}) &= \{(\uparrow,\{\mathtt{z}\overset{\downarrow}{\rightarrow}\mathtt{z},\mathtt{z}\rightarrow\mathtt{y},\mathtt{d}\overset{\downarrow}{\rightarrow}\mathtt{d}\}),(\uparrow,\{\mathtt{y}\overset{\downarrow}{\rightarrow}\mathtt{y},\mathtt{y}\overset{\downarrow}{\rightarrow}\mathtt{z},\mathtt{d}\overset{\downarrow}{\rightarrow}\mathtt{d}\}),\\
&\qquad(\uparrow,\{\mathtt{z}\overset{\downarrow}{\rightarrow}\mathtt{y},\mathtt{z}\overset{\downarrow}{\rightarrow}\mathtt{z},\mathtt{d}\overset{\downarrow}{\rightarrow}\mathtt{d}\}),(\underline{\ddagger},\{\mathtt{x}\rightarrow\mathtt{x},\mathtt{y}\rightarrow\mathtt{y},\mathtt{z}\rightarrow\mathtt{z},\mathtt{d}\rightarrow\mathtt{d}\})\}\\
C(\mathtt{u}) &= \{(\uparrow,\{\mathtt{v}\overset{\downarrow}{\rightarrow}\mathtt{v},\mathtt{w}\overset{\downarrow}{\rightarrow}\mathtt{w}\}),(\underline{\ddagger},\{\mathtt{u}\rightarrow\mathtt{u},\mathtt{v}\rightarrow\mathtt{v},\mathtt{w}\rightarrow\mathtt{w}\})\}\\
C(\mathtt{f}_{\mathtt{dp}}) &= \{(\uparrow,\{\mathtt{z}\overset{\downarrow}{\rightarrow}\mathtt{z},\mathtt{z}\rightarrow\mathtt{y},\mathtt{d}\overset{\downarrow}{\rightarrow}\mathtt{d}\}),(\uparrow,\{\mathtt{y}\overset{\downarrow}{\rightarrow}\mathtt{y},\mathtt{y}\overset{\downarrow}{\rightarrow}\mathtt{z},\mathtt{d}\overset{\downarrow}{\rightarrow}\mathtt{d}\}),\\
&\qquad(\uparrow,\{\mathtt{z}\overset{\downarrow}{\rightarrow}\mathtt{y},\mathtt{z}\overset{\downarrow}{\rightarrow}\mathtt{z},\mathtt{d}\overset{\downarrow}{\rightarrow}\mathtt{d}\}),(\underline{\ddagger},\{\mathtt{x}\rightarrow\mathtt{x},\mathtt{y}\rightarrow\mathtt{y},\mathtt{z}\rightarrow\mathtt{z},\mathtt{d}\rightarrow\mathtt{d}\})\}\\
C(\mathtt{g}_{\mathtt{dp}}) &= \{(\uparrow,\{\mathtt{v}\overset{\downarrow}{\rightarrow}\mathtt{v},\mathtt{w}\overset{\downarrow}{\rightarrow}\mathtt{w}\}),(\underline{\ddagger},\{\mathtt{u}\rightarrow\mathtt{u},\mathtt{v}\rightarrow\mathtt{v},\mathtt{w}\rightarrow\mathtt{w}\})\}\\
C(\mathtt{h}_{\mathtt{dp}}) &= \{(\uparrow,\{\mathtt{r}\overset{\downarrow}{\rightarrow}\mathtt{r}\}),(\uparrow,\{\mathtt{r}\rightarrow\mathtt{r},\mathtt{s}\overset{\downarrow}{\rightarrow}\mathtt{s}\}),(\underline{\ddagger},\{\mathtt{r}\rightarrow\mathtt{r},\mathtt{s}\rightarrow\mathtt{s}\})\}
\end{aligned}
$$

Figure 6.6: Semi-ring total costs for some nodes of the SDG for Example 6.1.

The LDG for Example 6.1 is shown in Figure 6.7 on the facing page; small regular expressions have been added by hand, indicating the call site sequences that constitute the loops represented by the balloon.

We can immediately see that every balloon has an anchor, and as the anchors have already been marked as B, the balloons are well-anchored. Using bounded anchoring, we can mark nodes $\mathtt{x}$, $\mathtt{u}$, $\mathtt{f}_{\mathtt{dp}}$, $\mathtt{g}_{\mathtt{dp}}$ and $\mathtt{h}_{\mathtt{dp}}$ as B.

Recall that a balloon node represents an increasing loop for some variable f_x, and an anchor node represents a decreasing loop for some sibling variable f_y. Condition 6.7 on the page before thus expresses that *"for each* increasing loop on f_x *there exists* a BV variable f_y such that f_y decreases along the corresponding sibling loop."

Note the order of the quantifications: If we swap them, we get a weaker form of bounded anchoring: "*there exists* a BV variable f_y such that *for each* increasing loop on f_x, f_y decreases along the corresponding sibling loop." In terms of balloon and anchor nodes, it can be stated as

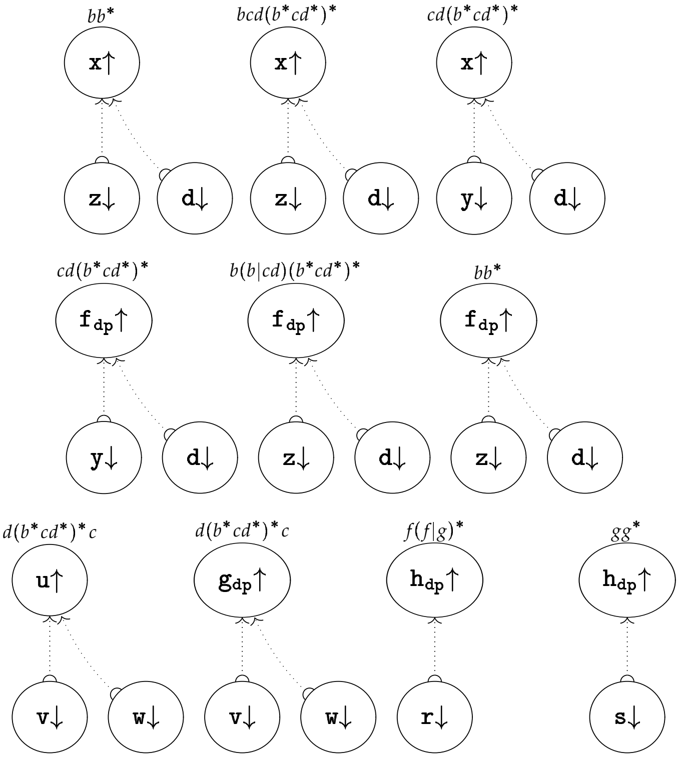


Figure 6.7: Loop dependency graph for Example 6.1

Condition 6.8 (Weak bounded anchoring)
If *all predecessors of C are marked B* and *there exists a B-marked node f_y such that all balloon nodes in the LDG containing a node from C are anchored in (have a chain edge from) an f_y anchor node,* then *mark C as B.*

Usually, we will only be using Condition 6.7; if we want to emphasize this, we call it *strong* bounded ancoring.

In fact, the domination condition is a special case of the anchoring condition: if C contains no $\uparrow$-labeled edges, there will be no balloon nodes in the LDG containing a node from C, so the second part of the bounded anchoring condition will be vacuously true. Consequently, the proof of soundness for bounded domination will follow from that of bounded anchoring.

In the following we will initialy assume that $\mathcal{E}^{\uparrow}$ is safe (cf. Definition 5.3 on page 67) for *all* $(\vec{v}, K)$.

To ease the presentation of size relationships we introduce the following shorthand

Notation 6.9
We write $e \prec x$ only when $\downarrow\!(x) \in \mathcal{E}^{\downarrow}\,[\![e]\!]\ \rho^{\downarrow}_{id}$, and we write $e_1 \trianglerighteq e_2$ when $\exists \rho : \mathcal{E}^{\uparrow}\,[\![e_1]\!]\ \rho \sqsupset \mathcal{E}^{\uparrow}\,[\![e_2]\!]\ \rho$.

Recall the two different modalities, cf. Definitions 5.2 and 5.3:

- *if* we have $e \prec x$ then the value of e must *always* be smaller than that of x, and conversely,
- *only if* we have $e \trianglerighteq x$ can e *possibly* be dependent on x.

The call paths can be seen as paths in a graph, justifying calling a set P_{sc} of parameters that satisfies the conditions

1. if $f_i \in P_{sc}$ and $\pi_1 = [f\ \vec{e}^{1}\ [\ldots [g\ \vec{e}^{k}]\!]], \pi_2 = [g\ \vec{e}'^{1}\ [\ldots [f\ \vec{e}'^{m}]\!]]$ are valid call paths and g_j depends increasingly on f_i along π_1 and f_i depends increasingly on g_j along π_2, then $g_j \in P_{sc}$
2. P_{sc} is not a proper subset of any other set of strongly connected parameters.

a *strongly connected component*.

Definition 6.10
Given parameters f_i and f_j and a call path $\pi = [f^1\ \vec{e}^1\ [\ldots[f^k\ \vec{e}^k]\!]$ with $f^1 = f^k$, we say that f_i is anchored in *f_j* along *π iff f_i^k depends increasingly only on f_i^1 (itself) or f^1's BV parameters along π, and $(st^\pi)_j \prec f_j^1$ and $BV(f_j)$.*

Theorem 6.11 (Soundness of Bounded Anchoring)
Given a strongly connected component P_{sc}, if *both the following conditions are satisfied:*

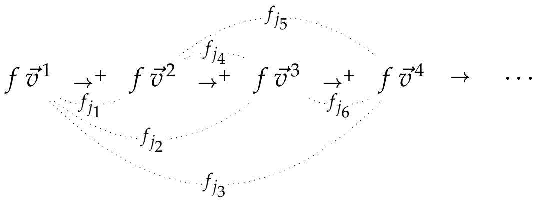
1. *For every call path, $\pi = [g\ \vec{e}^1\ [f\ \vec{e}^2]]$, of length 2, where $f_i \in P_{sc}$ for some $0 < i \leq arity(f)$, if f_i depends increasingly on g_j for some $0 < j \leq arity(g)$, then g_j is BV or $g_j \in P_{sc}$*

2. *For every $f_i \in P_{sc}$ and every call path $\pi = [f^1\ \vec{e}^1\ [\ldots[f^k\ \vec{e}^k]\!]$ with $f = f^1 = f^k$ where $(st^\pi)_i \trianglerighteq e_i^1$ and $k > 1$ there is a $j(\pi) \neq i$ such that f_i is anchored in $f_{j(\pi)}$ along π*

then *f_i is BV and we say that f_i is anchored in $\{f_{j(\pi)} \mid j(\pi)$ is given by condition 2$\}$ for all $f_i \in P_{sc}$.*

To prove this key theorem we first introduce some lemmas. In the following, we assume fixed P_{sc}, $f_i \in P_{sc}$ and program input $\vec{\sigma}$ given.

Given an f_i in a P_{sc} satisfying the conditions of Theorem 6.11, consider a transition sequence loop $(f, \vec{v}) \to^* (f, \vec{v}')$ where $v_i' > v_i$. By Theorem 4.3 on page 55, the corresponding call path π satisfies $\mathcal{E}\ [\![st^\pi]\!]\ \{f_1 \mapsto v_1, \ldots, f_n \mapsto v_n\} = \vec{v}'$, and as $\mathcal{E}^\uparrow$ is safe we have $\uparrow(e_i^1) \in \mathcal{E}^\uparrow\ [\![(st^\pi)_i]\!]\ \rho_{id}^\uparrow$, i.e. $(st^\pi)_i \trianglerighteq e_i^1$. So whenever such an increasing transition sequence occurs, the second condition in conjunction with the safety of $\mathcal{E}^\downarrow$ ensures the existence of a j such that f_j is BV and $v_j' < v_j$—this is what guarantees termination. However, j is not necessarily the same for all loops:


f_{j_5}
f_{j_4}
$f\ \vec{v}^1 \ \to^+\ f\ \vec{v}^2\ \to^+\ f\ \vec{v}^3\ \to^+\ f\ \vec{v}^4\ \to\ \cdots$
f_{j_1}
f_{j_6}
f_{j_2}
f_{j_3}

The aim is to show that the number of recurring $\rightarrow^+$ transitions where f_i increases in any such sequence is bounded because the value of *some* BV sibling $j_?$ always decreases.

This first requires a proof that such a single $j_?$ exists. Intuitively, this is done by subsequently thinning out sequences where the value of one f_j is known to decrease from the first state to all subsequent ones (but not necessarily *between* subsequent states).

The arity A of f is finite, so whenever some property $P(j)$ holds for M cases (possibly with different j), there must be a j' such that $P(j')$ holds for at least $\frac{M}{A}$ cases. This is used in

Lemma 6.12
Given an $N \in \mathbb{N}$, assume that for arbitrary $M \in \mathbb{N}$ a transition sequence $(f1,\vec{\sigma}) \rightarrow^+ (f^1,\vec{v}^1) \rightarrow^+ (f^2,\vec{v}^2) \rightarrow^+ (f^M,\vec{v}^M)$ occurs with a corresponding call path $\pi_M = [\![f^1\ \vec{e}^1\ [\![\dots\ [\![f^2\ \vec{e}^2\ [\![\dots\ [\![f^M\ \vec{e}^M]\!]]\!]$ where $f = f^1 = f^2 = \dots = f^M$. If for every subpath $\pi_M(s,t) = [\![f^s\ \vec{e}^s\ [\![\dots[\![f^t\ \vec{e}^t]\!]]\!]$ there is a $j(\pi_M(s,t))$ such that f_i is anchored in $f_{j(\pi_M(s,t))}$ along $\pi_M(s,t)$, then a transition sequence $(f1,\vec{\sigma}) \rightarrow^+ (f^1,\vec{v}^1) \rightarrow^+ (f^{M_0},\vec{v}^{M_0})$ occurs for some $M_0 \geq N$, and there exists a j and a sequence $m_1 < m_2 < \dots < m_N$ such that $(st^{\pi_{M_0}(m_p,m_{p+1})})_j \prec f_j^{m_p}$ for all $p \in \{1,\dots,N-1\}$

Proof: Let $A = \mathit{arity}(f)$. Given a call path π_M there must exist a sequence $s_1^1 < s_2^1 < \dots < s_{a_1}^1$ where $a_1 \geq \frac{M}{A}$ and a j^1 such that

$$(st^{\pi(1,s_1^1)})_{j^1} \prec f_{j^1}^1, \quad (st^{\pi(1,s_2^1)})_{j^1} \prec f_{j^1}^1, \quad \dots, \quad (st^{\pi(1,s_{a_1}^1)})_{j^1} \prec f_{j^1}^1$$

Further, there must exist a subsequence of $s_1^1,s_2^1,\dots,s_{a_1}^1$, call it $s_1^2,s_2^2,\dots,s_{a_2}^2$, where $a_2 \geq \frac{a_1}{A}$ and a j^2 such that

$$(st^{\pi(2,s_1^2)})_{j^2} \prec f_{j^2}^2, \quad (st^{\pi(2,s_2^2)})_{j^2} \prec f_{j^2}^2, \quad \dots \quad (st^{\pi(2,s_{a_2}^2)})_{j^2} \prec f_{j^2}^2$$

Continuing this way so that $s_1^{q+1},s_2^{q+1},\dots$ is a subsequence of $s_1^q,s_2^q,\dots$ for all $q \in \{1,\dots,A \cdot N\}$, we get a sequence $j^1, j^2,\dots, j^{A \cdot N}$, and there must be a j and a sequence $m_1 < m_2 < \dots < m_N$ such that $j = j^{m_1} = j^{m_2} = \dots = j^{m_N}$. If we demand that $M_0 \geq N \cdot A^{A \cdot N}$ then $a_{A \cdot N} \geq \frac{M_0}{A^{A \cdot N}} \geq N$, and thus $(st^{\pi(m_p,m_{p+1})})_j \prec f_j^{m_p}$

for all $p \in \{1,\dots,N-1\}$. Now the transition sequence $(f1,\vec{\sigma}) \to^+ (f^1,\vec{v}^1) \to^+ (f^2,\vec{v}^2) \to^+ (f^{M_0},\vec{v}^{M_0})$ has the desired properties. □

We now show that when a value "enters via" a transition from a BV parameter g_j to a parameter $f_i \in P_{sc}$, it can only increase a bounded number of times.

Again the argument of dividing the total number of cases where $P(f_j)$ holds (for possibly different f_j) by the finite number of possible function parameters is used.

Lemma 6.13
Assume P_{sc} satisfies the conditions of Theorem 6.11. Then there exists an $M \in \mathbb{N}$ such that for any transition sequence $(f1,\vec{\sigma}) \to^+ (f,\vec{v}^m) \to^+ (f,\vec{v}^k)$ with corresponding call path $[f1\ \vec{e}^1\ [\dots[f^m\ \vec{e}^m\ [\dots[f^k\ \vec{e}^k]\!]\!]$, where $f^k_{j_k}$ depends increasingly on $f^m_{j_m}$ along $\pi(m,k)$ via expressions $e^{m+1}_{j_{m+1}},\dots,e^k_{j_k}$ for some $f^k_{j_k}, f^m_{j_m} \in P_{sc}$, the number of $s \geq m$ for which $e^{s+1}_{j_{s+1}} \trianglerighteq f^s_{j_s}$ is less than M.

Proof: Let F be the number of functions in p and A the maximum function arity in p and assume that no such m exists. For arbitrary M' we can then find a transition sequence where the number of s for which $e^{s+1}_{j_{s+1}} \trianglerighteq e^s_{j_s}$ is at least $A \cdot F \cdot M'$. Then there must exist a function f where the number of s for which $f^s = f$ and $e^{s+1}_{j_{s+1}} \trianglerighteq e^s_{j_s}$ is at least M'. From Lemma 6.12 it now follows that for any $N \in \mathbb{N}$ a transition sequence $(f1,\vec{\sigma}) \to^+ (f,\vec{v}^{m_1}) \to^+ (f,\vec{v}^{m_2}) \to^+ (f,\vec{v}^{m_N})$ occurs with corresponding call path π such that $(st^{\pi(m_p,m_{p+1})})_j \prec f_j^{m_p} = f_j$, where f_j is BV.

This latter property—BV(f_j)—enables us to find an $N_0 \in \mathbb{N}$ such that the length of any descending chain

$$v_1 > v_2 > \cdots > v_m, \quad \text{where } v_1,\dots,v_m \in \{v_j \mid (f1,\vec{\sigma}) \to^* (f,\vec{v})\} \qquad (*)$$

is less than N_0.

It is clear that $\vec{v}^{m_q} = [\![st^{\pi(m_p,m_q)}]\!]\vec{v}^{m_p}$, and by the former property and the definition of $\prec$, we find that $[\![(st^{\pi(m_p,m_{p+1})})_j]\!]\vec{v}^{m_p} < [\![f_j^{m_p}]\!]\vec{v}^{m_p}$, so

$$v_j^{m_p} = [\![f_j^{m_p}]\!]\vec{v}^{m_p} > [\![(st^{\pi(m_p,m_{p+1})})_j]\!]\vec{v}^{m_p} = ([\![st^{\pi(m_p,m_{p+1})}]\!]\vec{v}^{m_p})_j = v_j^{m_{p+1}}$$

for all $p \in \{1,\ldots,N-1\}$. Choosing $N > N_0$ we get a chain

$$v_j^{m_1} > v_j^{m_2} > \cdots > v_j^{m_N},$$

which contradicts $(*)$ and thus forces us to conclude that M does in fact exist.

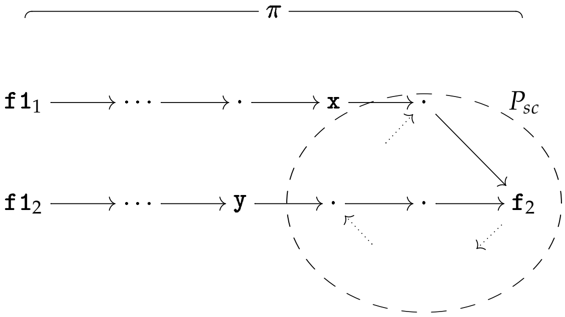
□

For some call path π that "ends" in the P_{sc}, i.e. where one of the parameters of the final call is in P_{sc}, we now want to separate the BV properties of the parameters at the various "entering points" of the P_{sc} from the length of the transition sequence *inside* the P_{sc}. To do this we define a

Definition 6.14 (BV based State Transformer, bst^π)
Given a P_{sc} satisfying condition 1 of Theorem 6.11, we define a BV-based state transformer *for a call path $\pi = [(f^1,\vec{e}^1),\ldots,(f^k,\vec{e}^k)]$ to be $bst^\pi = bst_k$, where*

$$\begin{array}{lcl} bst_1 & = & \vec{e}^1 \\ bst_{m+1} & = & (\gamma_1 \circ \cdots \circ \gamma_n)\vec{e}^{m+1}, \text{where } \gamma_1,\ldots,\gamma_n \text{ are substitutions given by} \\ & & \gamma_j = \begin{cases} [e'_j / f_j^m], & \text{if } f_j^m \in P_{sc} \\ [\langle m, f_j^m \rangle / f_j^m], & \text{if } f_j^m \notin P_{sc} \end{cases} \\ & & n = arity(bst_m),\ \vec{e}' = bst_m. \end{array}$$

Consider the following SDG$^\uparrow$ snippet:



In this case, $(st^\pi)_2$ expresses $\texttt{f}_2$ in terms of $\texttt{f1}_1$ and $\texttt{f1}_2$, whereas $(bst^\pi)_2$ expresses $\texttt{f}_2$ in terms of $\texttt{x}$ and $\texttt{y}$.

The key to prove Theorem 6.11 is König's Lemma for graphs (König, 1936; Diestel, 1997): we will define a graph containing exactly those nodes that correspond to the set of values we want to prove finite. Each edge between two (parameters × values)-pairs represents an actual computation with that "value dependency."

In the following definition, π is intended to be the corresponding call path to the existentially quantified transition sequence.

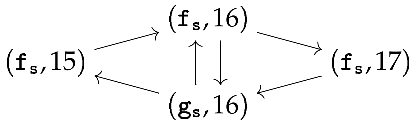
Definition 6.15 (Transition Graph of Values, TGV)
Given a set of strongly connected parameters, P_{sc}, the transition graph of values *for P_{sc} is $TGV = (\mathcal{V},\mathcal{E})$, where*

$$\begin{aligned}
\mathcal{V} &= \{(f_i,v_i) \mid f_i \in P_{sc}, (f1,\vec{\sigma}) \to^{+} (f,\vec{v})\}\\
\mathcal{E} &= \{(g_j,w_j) \to (f_i,v_i) \mid (f1,\vec{\sigma}) \to^{+} (g,\vec{w}) \to (f,\vec{v}) \wedge g_j, f_i \in P_{sc} \wedge\\
&\qquad \exists \pi = [g\ \vec{x}\ [f\ \vec{e}]] : \vec{v} = [\![st^{\pi}]\!]\vec{w} \wedge f_i \text{ depends increasingly on } g_j \text{ along } \pi\}
\end{aligned}$$

Example 6.16
Consider the program

```
f1(b1,b2) = f(b1,b2)
f(b,s) = if b < 17 then g(b, min(b,s)) else f(b-1, s+1)
g(b,s) = if ... then f(17, min(b,s)) else 42
```

Since calls to `f1` for $\vec{\sigma}$ = (`b1`,`b2`) = (`17`,`15`) is `f1(17,15)` → `f(17,15)` → `f(16,16)` → `g(16,16)` → `f(17,16)` → `f(16,17)` → `g(16,16)` → ⋯, (part of) the TGV for $P_{sc} = \{$`f`$_s$,`g`$_s\}$ of this small program is



For call path π = [`f1 (b1,b2)` [`f (b,s)` [`f (b-1,s+1)` [`g (b-1,min(b,s))` [`f (17,min(b,s)`)]]]]] the BV based state transformer is given by bst^{π} = `(17, min(`⟨4,`g`$_b$⟩`, min(`⟨3,`f`$_b$⟩`,`⟨1,`f1`$_{b2}$⟩`+1)), `⟨4,`g`$_d$⟩`)`

König's Lemma says that $A \wedge B \Rightarrow C$, where $A \equiv$ "TGV is infinite," $B \equiv$ "TGV is locally finite," $C \equiv$ "TGV has a non-trivial infinite path." We want to show $\neg A$, by using $B \wedge \neg C \Rightarrow \neg A$, which is equivalent to König's Lemma.

First we need to prove B by looking at how the dependent successor nodes of a node (g_j, w_j) is computed.

Lemma 6.17
Given a transition graph of values, TGV, for a set of strongly connected parameters, P_{sc}, that satisfies the conditions of Theorem 6.11, TGV will be locally finite, i.e. each node in TGV will have finitely many outedges.

Proof: Consider a node $(g_j, w_j) \in \mathcal{V}$. Then, for any given transition sequence $(f1, \vec{\sigma}) \rightarrow^{+} (f^{k-1}, \vec{v}^{k-1}) \rightarrow (f^k, \vec{v}^k)$ with corresponding call path $\pi = [f1\ \vec{e}^1\ [\ldots[f^{k-1}\ \vec{e}^{k-1}\ [\ldots[f^k\ \vec{e}^k]\!]$ where $f^{k-1} = g$ and $v_j^{k-1} = w_j$, let $I = \{i \mid f_i^k$ depends increasingly on g_j along $\pi(k-1,k)\}$. For each $i \in I$ we can compute v_i^k by $v_i^k = ([\![bst^\pi]\!]\varrho)_i$, where $\varrho\ \langle m, f_j^m\rangle = v_j^m$. By definition of bst^π, pairs $\langle m, f_j^m\rangle$ occurring in bst^π lie in the domain of ϱ.

Let $V_{max}(0) = \max_\pi\{v_j^m \mid \langle m, f_j^m\rangle$ occurs in $bst^\pi\}$; this number is finite because all f_j^m's lie outside the P_{sc} and are therefore by assumption BV. According to Lemma 6.13 there exists an M such that for every pair $\langle m, f_j^m\rangle$ occurring in bst^π, if f_i^k depends increasingly on f_j^m along $\pi(m,k)$ via expressions $e_{j_m}^m, e_{j_{m+1}}^{m+1}, \ldots, e_i^k$, the number of s, $m < s < k$, for which there exists an increasing operation from $f_{j_s}^s$ to $e_{j_{s+1}}^{s+1}$ is less than M.

Consider $s = 1$; then the value of f_i^k is bounded by the result of 1 increasing operation, the arguments of which are all bounded by $V_{max}(0)$. Call this new bound $V_{max}(1)$. Continuing in this fashion, we get bounds $V_{max}(0), \ldots, V_{max}(M)$ for $s = 0, \ldots, M$. This in turn implies that for the considered node (g_j, w_j),

$$\begin{aligned}\{(f^k, v_i^k) \mid (f1, \vec{\sigma}) \rightarrow^{*} (g, \vec{w}') \rightarrow (f^k, \vec{v}^k) \text{ occurs } \wedge w'_j = w_j \\ \wedge \exists \pi = [(g, \vec{e}^1), (f, \vec{e}^2)] : \vec{v} = [\![st^\pi]\!]\vec{w}' \wedge \\ f_i \text{ depends increasingly on } g_j \text{ along } \pi\}\end{aligned}$$

is finite because $v_i^k \leq V_{max}(M)$, i.e. that the TGV is locally finite. □

Proof (of the soundness of bounded anchoring, Theorem 6.11): For given P_{sc} we can construct the transition graph of values, TGV, for P_{sc}. By Lemma 6.17 we establish that TGV is locally finite. Next we observe that because there are only finitely many functions in the program, Lemma 6.13 implies that there are no non-trivial (i.e. not with a bounded set of different path nodes) infinite paths in TGV, because an infinite path would have to pass through nodes (f,v) for arbitrary large v, thus entailing infinitely many increasing operations.

Using these two properties and the contraposition of König's Lemma we conclude that TGV is finite, and it follows that $\{v_i \mid (f1,\vec{\sigma}) \to^* (f,\vec{v})\}$ is a finite set for all $f_i \in P_{sc}$. □

Weakening the safety of $\mathcal{E}^\uparrow$. The above lemmas and theorems hold also when $\mathcal{E}^\uparrow$ only satisfies the safety in Defintion 5.3 on page 67 for *almost* all $(\vec{v},K)$. The intuition is this: Choose K_0 such that the safety is satisfied for all $(\vec{v},K)$ where $K > K_0$. Admittedly, there might now be transitions $(f,\vec{v}) \xrightarrow{*} (f,\vec{v}')$ where $v'_i > v$ without $\mathcal{E}^\uparrow$ warning about this, i.e. $\uparrow(e^1_i) \notin \mathcal{E}^\uparrow [\![(st^\pi)_i]\!]\ \rho^\uparrow_{id}$, but this is only possible if $\vec{v}' < K_0$. But this means that there can still only be finitely many transitions with *different* v'_i before we find $\uparrow(e^1_i) \in \mathcal{E}^\uparrow [\![(st^\pi)_i]\!]\ \rho^\uparrow_{id}$.

When computing $V_{max}(0),\ldots,V_{max}(M)$ we make sure they are at least K_0:

$$\begin{aligned} V_{max}(0) &= \max\left(\{K_0\} \cup \{v^m_j \mid \langle m, f^m_j\rangle \text{ occurs in } bst^\pi\}\right) \\ V_{max}(i) &= \max\left(\{K_0\} \cup \{op\ v_1 \ldots v_n \mid op \text{ is an increasing operation} \right. \\ &\qquad\qquad\qquad\qquad\qquad \left. \wedge\, v_j \le V_{max}(i-1), 1 \le j \le n\}\right) \end{aligned}$$

However, $V_{max}(M)$ is still a finite value, so the soundness theorem still holds.

6.4 Program termination

The bounded anchoring condition is applied repeatedly to the SCCs in topological order until no new SCCs change marking from $\perp$ to B. This will naturally stop in the end, as there are a finite number of SCCs and we only ever change marks from $\perp$ to B. Note that the SDG and LDG graphs need *not* be recomputed in these iterations.

After applying these two conditions exhaustively, any remaining ⊥-marked nodes must be considered unsafe, i,e. possibly unbounded. The B-marked nodes are described by the following

Theorem 6.18 (Soundness of program termination algorithm)
After applying the bounded anchoring and domination conditions some number of times to an SDG in which only program input nodes initially are marked B, if all `dp` *nodes are marked B then evaluation of p terminates.*

Proof: As the goal function is only ever called once, it is obvious that its parameters are BV. As the SDG is a faithful representation of $L(G)$ (cf. Proposition 6.4 on page 83), Theorem 6.11 on page 91 shows that each time the bounded anchoring condition is applied, only BV parameters are marked B. Thus, variables of p whose nodes end up being marked B *are in fact of bounded variation*, i.e. they can take on only finitely many different values during evaluation.

As the `dp` nodes record the call depth, we observe that if they all are BV, the call tree for any computation will have no infinite paths. Further, as we have no iteration constructs, each function can only make a bounded number of direct function calls before returning, i.e. each node of the call tree has a finite number of children. König's Lemma now gives us that the call tree is finite, and as all non-call operations terminate (we have no iteration constructs), the program will terminate. It is straightforward to express this argument in terms of call paths and transition sequences, involving the connection between call paths and actual computations (Theorem 4.3) along with Linear Compression (Corollary 3.5). □

If there is a node $f_{\texttt{dp}}$ that is not BV, there is a risk that p will loop infinitely in function f: if furthermore all ordinary variables are BV, p is quasiterminating, otherwise it may not terminate.

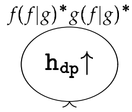
For Example 6.1 we see that all `dp` nodes are detected to be BV, so the program terminates. Now suppose we modified the definition of `h` slightly by changing an `r` to a `17`:

```
h r s = if r > 0 then h^f (r - 1) 42
        else if s > 0 then h^g 17 (s - 1) else r
```

Now we find that the edge $\mathtt{r} \xrightarrow{g} \mathtt{r}$ disappears from the SDG$^{\downarrow}$, and in the total costs for $\mathtt{h}_{\mathtt{dp}}$ are now

$$\mathtt{h}_{\mathtt{dp}} : \{(\uparrow, \{\mathtt{r} \xrightarrow{\downarrow} \mathtt{r}\}), (\uparrow, \{\mathtt{s} \xrightarrow{\downarrow} \mathtt{s}\}), (\uparrow, \{\}), (⇞, \{\mathtt{r} \longrightarrow \mathtt{r}, \mathtt{s} \longrightarrow \mathtt{s}\})\},$$

giving rise to an unanchored balloon



so now the analyses cannot guarantee termination, which is correct: for $\mathtt{a} = 0$, $\mathtt{b} = 0$, the modified `contrived` program loops endlessly exactly in function `h` as predicted.

Chapter 7

Extension to Partial Evaluation

In *off-line partial evaluation* we make use of a *two-level language*: every construct is annotated either as static or dynamic, according to when the data for computing it is present. The goal of this paper is to compute such a *binding-time division* which ensures that specialisation according to these binding times will terminate. The syntax and semantics of the two-level language is spelled out in Appendix A. It includes *specialisation points* which are marked call sites of the form $f^\star\, e_1 \ldots e_n$, indicating that partial evaluation should residualising f, i.e. generate a residual version specialised to its static arguments.

The extension of these algorithms to partial evaluation is fairly straightforward; the notion of bounded variation is now even more useful, as we can utilise memoisation by inserting specialisation points. Not only can we detect when specialisation could diverge, we can *make* it terminate: First we can make all ordinary variables not detected to be of bounded variation dynamic (as dynamic variables cannot cause the specialiser to diverge), and then for every unanchored **dp**-balloon we can insert specialisation points. At the specialisation points, the values of the static variables are memoised, and as they are all of bounded variation, specialisation will terminate.

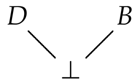
In the following, we will be referring again to the modified `contrived` program of Example 6.1 on page 77, where the expression $\texttt{h}^g$ `r (s - 1)` has been changed to $\texttt{h}^g$ `17 (s - 1)` as described on page 98.

7.1 Bounded static variation

For partial evaluation we replace the notion of bounded variation with *bounded static variation,* BSV:

$$\boxed{f_i \text{ is BSV iff } \forall \vec{u} \in \mathit{Value}^s : \{v_i \mid \exists \vec{w} \in \mathit{Value}^d : (f1, \vec{u}\,\vec{w}) \to^* (f, \vec{v})\} \text{ is finite,}}$$

where s is the number of static and d the number of dynamic parameters of *f1*.



As we are now also dealing with dynamic variables, we extend our set of property marks from Section 6.1 on page 77 to a domain of three values $\perp, D, B$, signifying that a variable is unreached ($\perp$), dynamic (D), or of bounded static variation (B). The binding time values are ordered by $\perp \sqsubseteq D$ and $\perp \sqsubseteq B$, and again, all nodes will initially be marked $\perp$.

7.2 Capturing dynamic dependency

As we now also need to make sure the binding time assignments of the variables is congruent (i.e. static expressions must not require the value of dynamic expressions to be computed), we need to know when an expression will be dynamic, or equivalently, which parameters and function return values must be available (not dynamic) for the value of an expression to be available. This is somewhat simpler than size dependency, as we can represent the "dynamicness" of the return value of each function explicitly, instead of computing some approximation.

For each function f in p the special symbol $f_{\mathbf{R}}$ represents the return value of f. We now define a dynamic dependency function $\mathcal{D}_{\mathbf{e}}$ in Figure 7.1 which given an expression returns a set of symbols (parameter names or return

$$Symbol = Varname \cup \{f_{\mathbf{R}} \mid f \in Funname\}$$

$$\mathcal{D}_{\mathbf{e}} : Expression \to \mathcal{P}(Symbol)$$

$$\begin{array}{lcl}
\mathcal{D}_{\mathbf{e}}\,[\![c]\!] & = & \{\} \\
\mathcal{D}_{\mathbf{e}}\,[\![x]\!] & = & \{x\} \\
\mathcal{D}_{\mathbf{e}}\,[\![\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3]\!] & = & \mathcal{D}_{\mathbf{e}}\,[\![e_1]\!] \cup \mathcal{D}_{\mathbf{e}}\,[\![e_2]\!] \cup \mathcal{D}_{\mathbf{e}}\,[\![e_3]\!] \\
\mathcal{D}_{\mathbf{e}}\,[\![b\ e_1 \ldots e_n]\!] & = & \mathcal{D}_{\mathbf{e}}\,[\![e_1]\!] \cup \cdots \cup \mathcal{D}_{\mathbf{e}}\,[\![e_n]\!] \\
\mathcal{D}_{\mathbf{e}}\,[\![f\ e_1 \ldots e_n]\!] & = & \{f_{\mathbf{R}}\}
\end{array}$$

Figure 7.1: Dynamic dependency function

value symbols) such that if one of these symbols is dynamic, the expression must be classified as dynamic.

We now capture the dependencies in the entire program in a *dynamic dependency graph*, DDG. One might think that the DDG and the SDG$^{\uparrow}$ convey the same information, but it is in fact slightly different, which can be seen from the example

```
f x y  =  g (if y then x - 1 else x - 2)
g x    =  ···
```

where the DDG should contain an edge from `y` to `x`, (`x` must be dynamic if `y` is), but the SDG$^{\uparrow}$ should not (the size of the value of the **if**-expression is only related to `x`). Furthermore, two variables connected via return nodes in the DDG may be represented directly by an edge in the SDG$^{\uparrow}$ (or may be absent if the edge effect computed by $\mathcal{E}^{\uparrow}$ is empty).

```
f x = g (h x)
g u = ···
h r = r + 1
```

DDG :

$$\begin{tikzcd} \mathtt{x} \arrow[r] & \mathtt{r} \arrow[r] & \mathtt{h}_{\mathbf{R}} \arrow[r] & \mathtt{u} \end{tikzcd}$$

SDG$^{\uparrow}$:

$$\begin{tikzcd} \mathtt{x} \arrow[r] \arrow[rr, bend left, "\uparrow"'] & \mathtt{r} & \mathtt{u} \end{tikzcd}$$

It must, however, always be the case that if there exists a path from f_x to g_y in the SDG$^{\uparrow}$, there exists a (possibly different) path from f_x to g_y in the DDG.

The reason why it was not desirable to treat size dependencies in a similar way, connecting `r` to $\mathtt{h}_\mathbf{R}$, is that if there are two or more calls to a function this can create graph loops that do not correspond to any program evaluation path—and it would be pointless to try to anchor them. It would also coerce the return value sizes of *all* the applications of f in the entire program, thus losing too much information about the sizes of the arguments at each call site. This coercion is acceptable in the case of dynamic dependency because we only need to distinguish between available/not available and we assume a monovariant binding time for each function: if its return value is classified as dynamic, it is dynamic at all call sites.

Our basic set of nodes, V, consists of all the function parameters $f_x, g_y, \ldots$ of the program, along with all the function return value symbols $f_\mathbf{R}, g_\mathbf{R}, \ldots$ The DDG for p is then defined to be the pair $(V, E_\mathcal{D})$, where $E_\mathcal{D}$ is a set of unlabeled edges

$$\begin{aligned} E_\mathcal{D} \;=\;& \mathcal{DD}_\mathbf{e}\,[\![e^{f1}]\!] \cup \cdots \cup \mathcal{DD}_\mathbf{e}\,[\![e^{fn}]\!] \cup \\ & \{f_x \to f1_\mathbf{R} \mid f_x \in \mathcal{D}_\mathbf{e}\,[\![e^{f1}]\!]\} \cup \cdots \cup \{f_x \to fn_\mathbf{R} \mid f_x \in \mathcal{D}_\mathbf{e}\,[\![e^{fn}]\!]\}, \end{aligned}$$

and $\mathcal{DD}_\mathbf{e}$ is defined in Figure 7.2. Each node carries the same mark from the binding time domain $\{\bot, D, B\}$ as the corresponding node in the SDG graphs, and is initially marked as $\bot$. The DDG for Example 6.1 is shown in Figure 7.3 (Compare with Figure 6.5).

$$\begin{aligned} \mathcal{DD}_\mathbf{e}\,[\![c]\!] \;&=\; \{\} \\ \mathcal{DD}_\mathbf{e}\,[\![x]\!] \;&=\; \{\} \\ \mathcal{DD}_\mathbf{e}\,[\![\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3]\!]\;\rho \;&=\; \mathcal{DD}_\mathbf{e}\,[\![e_1]\!] \cup \mathcal{DD}_\mathbf{e}\,[\![e_2]\!] \cup \mathcal{DD}_\mathbf{e}\,[\![e_3]\!] \\ \mathcal{DD}_\mathbf{e}\,[\![b\; e_1 \ldots e_n]\!] \;&=\; \mathcal{DD}_\mathbf{e}\,[\![e_1]\!] \cup \cdots \cup \mathcal{DD}_\mathbf{e}\,[\![e_n]\!] \\ \mathcal{DD}_\mathbf{e}\,[\![f\; e_1 \ldots e_n]\!] \;&=\; \mathcal{DD}_\mathbf{e}\,[\![e_1]\!] \cup \cdots \cup \mathcal{DD}_\mathbf{e}\,[\![e_n]\!] \cup \\ & \{g_y \longrightarrow f_1 \mid g_y \in \mathcal{D}_\mathbf{e}\,[\![e_1]\!]\;\rho\} \cup \cdots \cup \\ & \{g_y \longrightarrow f_n \mid g_y \in \mathcal{D}_\mathbf{e}\,[\![e_n]\!]\;\rho\} \end{aligned}$$

Figure 7.2: Dynamic dependency graph (DDG) edge generating operator

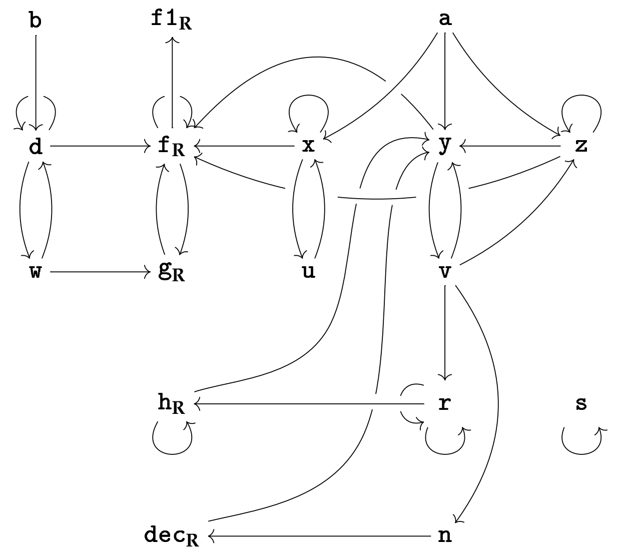


Figure 7.3: Dynamic dependency graph (DDG) for Example 6.1

We determine which parameters must be classified as dynamic by marking *f1*'s dynamic parameter nodes as *D* and then simply propagating this mark depth-first along the edges of the DDG. Using the DDG of Figure 7.3 we see that if `b` is dynamic then also `d` and `w` must be classified as dynamic; if furthermore `a` is dynamic then all parameters except `s` must be dynamic.

7.3 Detecting bounded static variation

BSV is detected in exactly the same way as BV, so as before we find for Example 6.1 that all nodes except $\mathtt{h}_{\mathtt{dp}}$ are BSV.

When no more node marks are changed from ⊥ to *B*, we re-mark the remaining $\perp$ nodes D. In the example, as node $\mathtt{h_{dp}}$ and $\mathtt{h_R}$ are not BSV, we change their marks to D. This is sound due to the following

Lemma 7.1
If after applying the bounded domination or anchoring conditions a number of times a node mark is changed from $\perp$ to D, the nodes marked B will still be BSV.

Proof: The only way a node can be marked B, save for the initial static input nodes, is by the bounded domination and anchoring conditions, but they rely only on B marks of other nodes, not $\perp$ marks. □

7.4 Inserting specialisation points

Now that all nodes are either marked D or B, we are ready to apply *call loop anchoring*. For partial evaluation, not only do we look for non-anchored `dp` loops, we in fact "break" each such loop by adding a specialisation point to one of its edges (i.e. a call site).

The reasoning is this: if all static parameters are BSV, and all `dp` loops are broken by inserting a specialisation point, then the memoisation at this point will at some point have memoised all the different values for the static parameters, and no further specialisation will occur in this loop. The `dp` nodes in the non-broken loops are BSV, i.e. the call depth is bounded, and cannot cause nontermination.

To do the call loop anchoring we include call site information when building the LDG; all we need to do to the $\mathcal{SD}_{\mathbf{e}}$ function shown in Figure 6.2 on page 79, extended on page 83, is change $g_y \xrightarrow[S_i]{\delta} f_i$ to $g_y \xrightarrow[S_i]{\delta,\{s\}} f_i$.

We now let $Cost = D^{\uparrow} \times \mathcal{P}(V \times D^{\downarrow} \times V) \times \mathcal{P}(CallSite)$ and use the following modified

Definition 7.2 (Cost structure with call site information)
Define the cost labelling function $l : E_{\mathcal{C}} \to Cost$ by $l\left(g_y \xrightarrow[S]{\delta,\sigma} f_x\right) = (\delta, S, \sigma)$ and

multiplication operator $\otimes : (Cost \times Cost) \to Cost$ *by*

$$\gamma_1 \otimes \gamma_2 = \bigcup_{(\delta_1, S_1, \sigma_1) \in \gamma_1} \bigcup_{(\delta_2, S_2, \sigma_2) \in \gamma_2} \{(\delta_1 \square^{\uparrow} \delta_2, S_2 \circ S_1, \sigma_1 \cup \sigma_2)\},$$

$$\text{where } S_2 \circ S_1 = \left\{ h_w \xrightarrow{\delta_2 \square^{\downarrow} \delta_1} f_u \;\middle|\; \exists g_v : h_w \xrightarrow{\delta_1} g_v \in S_1 \wedge g_v \xrightarrow{\delta_2} f_u \in S_2 \right\},$$

and finally $\mathbf{1} = \left\{ \left(\iota^{\uparrow}, \{ f_x \xrightarrow{\iota^{\downarrow}} f_x \mid f_x \in p \}, \emptyset \right) \right\}$.

As the call site sets are added and multiplied by simple set union, this structure is a simple extension of that of Lemma 6.6 on page 85, and we conclude the following

Corollary 7.3
The components defined in Definition 7.2 form a semi-ring structure $\mathbb{S} = (Cost, \cup, \otimes, \emptyset, \mathbf{1})$.

The "total cost" computed by the semi-ring algorithm will now be a set of triples, (δ, σ, S), where each triple describes loops with call site set σ and sibling loop effects S. Some of the total costs for `dp` loops of the modified Example 6.1 can be seen in Figure 7.4.

When we build the LDG, we have now automatically computed the sets of call sites σ that we previously added by hand as regular expressions, cf. Figure 6.7 on page 89.

When the bounded anchoring has finished, any $f_{\texttt{dp}}$ nodes that could not be detected to be bounded are marked D, and we can then insert the necessary specialisation points by applying the following condition to each $f_{\texttt{dp}}$ balloon node:

Condition 7.4 (Call loop anchoring)
For balloon node $(f_{\texttt{dp}}, \sigma)$*, where* $f_{\texttt{dp}}$ *is marked* D*,* if *there is no call site* $s \in \sigma$ *which is a specialisation point,* then *select a specialisation point* $s' \in \sigma$.

The decision as to which s' one should choose as the specialisation point is not trivial and there seems to be no simple criteria. One possibility is to give the call sites priority according to what kind of balloon nodes they are in, and always choosing the one which is in the fewest anchored and the

$$
\begin{aligned}
C(\mathtt{f}_{\mathtt{dp}}) = \{ & (\uparrow, \{\mathtt{z} \overset{\downarrow}{\to} \mathtt{z}, \mathtt{z} \to \mathtt{y}, \mathtt{d} \overset{\downarrow}{\to} \mathtt{d}\}, \{b\}), \\
& (\uparrow, \{\mathtt{y} \overset{\downarrow}{\to} \mathtt{y}, \mathtt{y} \overset{\downarrow}{\to} \mathtt{z}, \mathtt{d} \overset{\downarrow}{\to} \mathtt{d}\}, \{c,d\}), (\uparrow, \{\mathtt{y} \overset{\downarrow}{\to} \mathtt{y}, \mathtt{y} \overset{\downarrow}{\to} \mathtt{z}, \mathtt{d} \overset{\downarrow}{\to} \mathtt{d}\}, \{b,c,d\}), \\
& (\uparrow, \{\mathtt{z} \overset{\downarrow}{\to} \mathtt{y}, \mathtt{z} \overset{\downarrow}{\to} \mathtt{z}, \mathtt{d} \overset{\downarrow}{\to} \mathtt{d}\}, \{b\}), (\uparrow, \{\mathtt{z} \overset{\downarrow}{\to} \mathtt{y}, \mathtt{z} \overset{\downarrow}{\to} \mathtt{z}, \mathtt{d} \overset{\downarrow}{\to} \mathtt{d}\}, \{b,c,d\}), \\
& (\underline{\uparrow}, \{\mathtt{x} \to \mathtt{x}, \mathtt{y} \to \mathtt{y}, \mathtt{z} \to \mathtt{z}, \mathtt{d} \to \mathtt{d}\}, \{\})\}
\end{aligned}
$$

$$
\begin{aligned}
C(\mathtt{g}_{\mathtt{dp}}) = \{ & (\uparrow, \{\mathtt{v} \overset{\downarrow}{\to} \mathtt{v}, \mathtt{w} \overset{\downarrow}{\to} \mathtt{w}\}, \{c,d\}), (\uparrow, \{\mathtt{v} \overset{\downarrow}{\to} \mathtt{v}, \mathtt{w} \overset{\downarrow}{\to} \mathtt{w}\}, \{b,c,d\}), \\
& (\underline{\uparrow}, \{\mathtt{u} \to \mathtt{u}, \mathtt{v} \to \mathtt{v}, \mathtt{w} \to \mathtt{w}\}, \{\})\}
\end{aligned}
$$

$$
\begin{aligned}
C(\mathtt{h}_{\mathtt{dp}}) = \{ & (\uparrow, \{\mathtt{r} \overset{\downarrow}{\to} \mathtt{r}\}, \{f\}), (\uparrow, \{\mathtt{s} \overset{\downarrow}{\to} \mathtt{s}\}, \{g\}), (\uparrow, \{\}, \{f,g\}), \\
& (\underline{\uparrow}, \{\mathtt{r} \to \mathtt{r}, \mathtt{s} \to \mathtt{s}\}, \{\})\}
\end{aligned}
$$

Figure 7.4: Semi-ring total costs for some call loops in the call graph of Example 6.1

most unanchored balloons. The choice also has an impact on the degree of *residual code sharing*—it may be desirable to insert more specialisation points than necessary for termination, simply to reduce the size of the residual (specialised) program.

The LDG for the modified Example 6.1 is shown in Figure 7.5 on the following page. It can readily be seen that a balloon for $\mathtt{h}_{\mathtt{dp}}$ is not anchored, and a specialisation point is needed for call site f or g, which is exactly what we expected.

7.5 Cascading specialisation point consequences

Inserting these specialisation points is essential for making partial evaluation terminate, but it has a nasty consequence: it interferes with the conditions used to guarantee the BSV properties. In the program of the modified Exam-

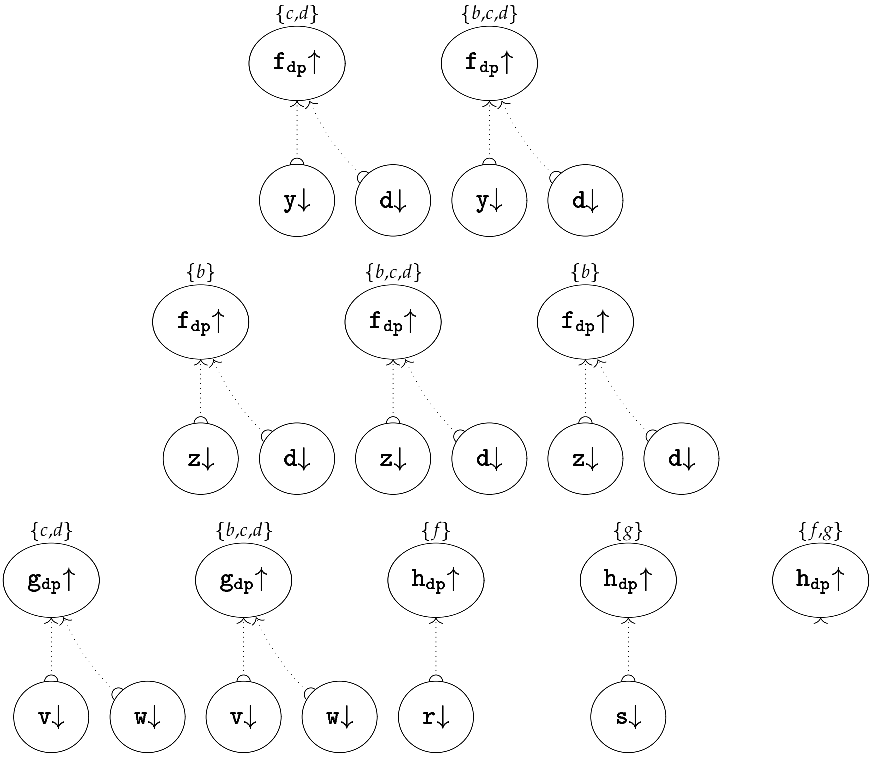


Figure 7.5: Part of the loop dependency graph for the modified Example 6.1

ple 6.1, the specialisation point at the recursive call to `h` is needed, but this causes `y`, which has been used as an anchor for guaranteeing `x` to be of BSV, to become a dynamic parameter and thus useless as an anchor!

Fortunately, this problem can be remedied, based on the simple observation that generalising, i.e. changing marks to D, does not change any δ-annotations on the edges of the size dependency graphs—only the annotations of nodes. Thus the same dependencies, and the same anchors persist. And this is where all our hard work setting up elaborate data structures is finally rewarded: First we note that when inserting a specialisation point $\ldots(f^{\star}\ e_1 \ldots e_n)\ldots$ we can use the DDG to propagate the newly introduced dynamic property (of the return value of f) by marking $f_{\mathtt{R}}$ as D and propagating this information in the DDG.

This solves the problem of parameters marked as B by domination: as no new $\uparrow$-edges are introduced, the only problem is if any predecessors of a B-marked SCC C has its mark changed to D, but this will immediately be propagated to the nodes of C, via the DDG.

To solve the problem of anchors rendered useless by specialisation point insertion, we use the DDG together with the LDG by *cascading* the generalising consequences of inserting specialisation points. This is done by applying the following condition to the LDG, for each newly inserted specialisation point:

Condition 7.5 (Dynamic cascading)
For a node v just marked D,

1. *Consider each successor v' in the DDG:* If *v' is not already marked D,* then *mark v' as D and apply dynamic cascading recursively.*

2. *Consider any anchor nodes containing v: follow the anchor chain to the balloon;* if *there are no other anchor chains left to anchors marked B,* and *the node, v', of the balloon is not already marked D,* then *mark v' as D and apply dynamic cascading recursively to v'.*

In Example 6.1, when we insert the specialisation point in the recursive call to `h`, we find that $\mathtt{h}_{\mathtt{R}}$ and consequently nodes `y`, `v`, `z`, `r`, `n`, $\mathtt{dec}_{\mathtt{R}}$, $\mathtt{f}_{\mathtt{R}}$, $\mathtt{g}_{\mathtt{R}}$, and $\mathtt{f1}_{\mathtt{R}}$ become dynamic, chopping all anchors except `d` and `w` anchors (cf. Figure 6.7). Now there are only four B-marked loops left in the LDG, shown

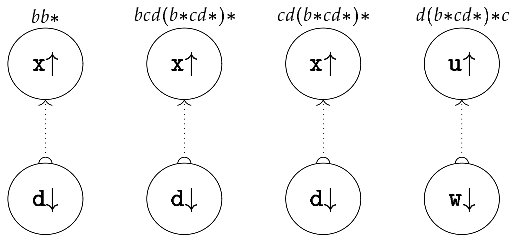


Figure 7.6: Remaining B-marked loops in the LDG after dynamic cascading

in Figure 7.6, and from the DDG it can be seen that if input `b` is dynamic, `d` and `w` become dynamic, causing `x` and `u` to be unanchored and thus turned dynamic.

In this way, each time a specialisation point causes some node to become dynamic *all* the (possibly transitively induced) effects this might have on the bounded domination and anchoring conditions used in the preceding analysis are accounted for. We thus have the following

Theorem 7.6 (Soundness of dynamic cascading)
After dynamic cascading has been applied to an LDG and DDG where nodes have been marked as B only by the domination and anchoring conditions, any remaining nodes marked as B are still of BSV.

Proof: Any remaining B-marked node f_x was originally marked B due to bounded domination or bounded anchoring, which rely only on other B marks, and only in two ways:

1. predecessors to the SCC containing f_x must be marked B for each increasing loop containing f_x, and

2. there must be at least one B-marked anchor.

If dynamic cascading invalidates condition 1 or 2, the corresponding cascading step 1 or 2 will re-mark f_x D. This relies on the fact that for every path from g_y to f_x in the SDG$^\uparrow$ there exists a path from g_y to f_x in the DDG. □

When dynamic cascading is finished, some BSV `dp`-loops might have turned dynamic. Thus call anchoring and dynamic cascading must be reapplied to insert any needed specialisation points until no further changes to node marks are made. This process cannot go on forever, as the number of nodes is finite and we only change nodes from B to D. Also note that the LDG need not be recomputed.

Finally, when call loop anchoring and dynamic cascading have stabilised, specialisation is guaranteed to terminate by

Theorem 7.7 (Termination of specialisation in partial evaluation)
If *all parameters in* p *classified as static are of BSV* and *all* $f_{\texttt{dp}}$ *balloon nodes in the LDG that are not anchored contain a specialisation point,* then *the evaluation* $\mathcal{PP}\ [\![p]\!]\ \vec{v}$ *will terminate for any* $\vec{v} \in V^s$.

Proof: As all parameters are BSV all loops with memoisation points will eventually terminate. Loops without specialisation points have anchored $f_{\texttt{dp}}$-nodes, so the call depth is bounded and specialisation of these loops will thus also terminate. □

Chapter 8

The Entire Algorithm

The entire algorithm for detecting program termination is summarised in

Algorithm 8.1 (Detection of evaluation termination)

Input: $p \in \mathit{Program}$.

Output: An answer "terminating" if p is guaranteed to terminate no matter which input is given, or an answer "quasiterminating" if p is guaranteed to only pass through finitely many different states during evaluation.

Steps:

1. Build the SDG.
2. Form the SCC DAG from the SDG, sort it topologically.
3. Build the LDG.
4. Apply the bounded anchoring condition in topological order to the SCCs, using the LDG for detecting anchors.
5. Reiterate from Step 4 until no property marks are changed.
6. If all $f_{\mathtt{dp}}$ nodes are marked B, termination of p is guaranteed. If all non-$f_{\mathtt{dp}}$ nodes are marked B, p is quasiterminating.

The extension to partial evaluation adds steps 1′, 5′, 7, 8 and extends step 6 in

Algorithm 8.2 (BTA guaranteeing termination)

Input: $p \in \mathit{Program}$ and binding times for goal function parameters.

Output: Binding-time division for all variables in p and a set of specialisation points, under which specialisation is guaranteed to terminate.

Steps:

1. Build the SDG$^{\uparrow}$.

1′. Build the DDG, mark p's dynamic input as D and propagate depth-first.

2. Form the SCC DAG from the SDG, sort it topologically.
3. Build the LDG.
4. Apply the bounded anchoring condition in topological order to the SCCs, using the LDG for detecting anchors.
5. Reiterate from Step 4 until no property marks are changed.

5′. Re-mark any remaining $\perp$-nodes as D.

6. For each balloon $f_{\mathbf{dp}}$ marked D: Apply the call loop anchoring condition; if a specialisation point is added for a call to function f, then mark node $f_{\mathbf{R}}$ as D and apply the dynamic cascading condition to it.
7. Reiterate from Step 6 until no binding time annotations are changed.
8. Now partial evaluation of p, using the specialisation points and the binding-time division indicated by the node marks, is guaranteed to terminate.

It is important to note that after each iteration in either of the two loops (steps 4–5 and 6–7), the binding time annotations are still correct with respect to BSV: in steps 5 and 7 any variables in p whose nodes are marked as B will be of BSV with the current assignments of Ds and Bs to the remaining variable nodes. It is therefore possible to "bail out" of the first loop before the annotations become stable, thus possibly pessimising, i.e. generalising some parameters that could have been detected to be of BSV. However, for partial evaluation to terminate it is essential that the second loop is iterated until no more changes occur. Otherwise there might be some program loop causing infinite specialisation, because there is no specialisation point in the loop.

8.1 Soundness

The overall structure of the formal soundness proof of the algorithm for detecting termination is shown in Figure 8.1 on the facing page.

Informally, the algorithms work because $\mathcal{E}^{\uparrow}$ captures all the risks of increasing transitions and $\mathcal{E}^{\downarrow}$ detects guaranteed decreasing transitions. During the anchoring (steps 4–5) the following invariant holds:

> *whenever a node is marked B, the corresponding parameter* is *BV (BSV).*

This is true because for a node to become marked B, only values from B-marked nodes can flow into it. Furthermore, if the node participates in increasing loops, there must be a strict decrease in a BV (BSV) sibling loop. As there are only a finite number of sibling parameters, each with a finite value on a well-founded domain, there can only be a finite number of increases of the parameter represented by the node, i.e. it is BV (BSV).

The same invariant holds after dynamic cascading (at step 7) because the connections described above are recorded in the LDG. If any sibling parameters that have been used as anchors are invalidated by being re-marked D, the effect of this on anchors is propagated in the dynamic cascading.

Thus, when the algorithms finish, any node marked B *is* BV (BSV).

For partial evaluation, the only risk of nontermination is now in loops of a function f whose $f_{\mathtt{dp}}$ node is marked D, but as every loop where $f_{\mathtt{dp}}$ is not anchored contains a specialisation point, specialisation will terminate.

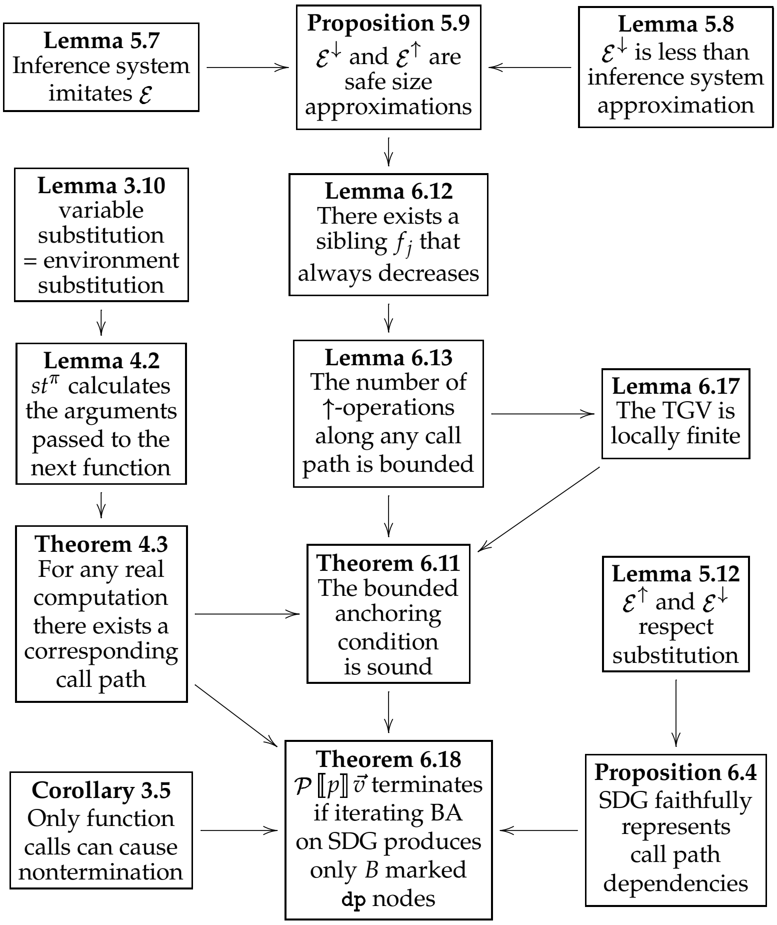

Lemma 5.7
Inference system imitates $\mathcal{E}$
Proposition 5.9
$\mathcal{E}^{\downarrow}$ and $\mathcal{E}^{\uparrow}$ are safe size approximations
Lemma 5.8
$\mathcal{E}^{\downarrow}$ is less than inference system approximation
Lemma 3.10
variable substitution = environment substitution
Lemma 6.12
There exists a sibling f_j that always decreases
Lemma 4.2
st^{π} calculates the arguments passed to the next function
Lemma 6.13
The number of $\uparrow$-operations along any call path is bounded
Lemma 6.17
The TGV is locally finite
Theorem 4.3
For any real computation there exists a corresponding call path
Theorem 6.11
The bounded anchoring condition is sound
Lemma 5.12
$\mathcal{E}^{\uparrow}$ and $\mathcal{E}^{\downarrow}$ respect substitution
Corollary 3.5
Only function calls can cause nontermination
Theorem 6.18
$\mathcal{P}\llbracket p \rrbracket \vec{v}$ terminates if iterating BA on SDG produces only B marked $\texttt{dp}$ nodes
Proposition 6.4
SDG faithfully represents call path dependencies

Figure 8.1: Overall structure of the soundness proof for the termination algorithm (note that bounded domination is not explicitly proved correct, as it is a special case of bounded anchoring)

Chapter 9

Implementation

When the binding-time and termination analysis algorithms described in this paper are to be implemented, we must be careful that it is done as time and space savingly as possible. In the following we will consider which data structures to use and what complexity they result in. Finally, we will report on results obtained using a prototype implementation of the analyses.

9.1 Data structures

Program syntax is represented by an annotated syntax tree; each function and parameter is represented uniquely, and in any expression we assume constant-time access to the contained functions and parameters. For instance, given expression `f (x + 1) (y - 1)`, finding (the internal representation of) `f`, `x` and `y` takes constant time. To achieve this, we must first traverse the source program and create the necessary links.

Size dependency sets are used for dependency insertion, traversal, $\sqcap^{\downarrow}$ and $\sqcup^{\uparrow}$, the latter involving $\cap$ and $\cup$ operations. Given function f there is an upper bound on the number of different elements that can be inserted, namely $arity(f)$. This is usually a small number, so a bit vector representation is reasonable.

We represent an increasing dependency set Δ by a tuple of vectors $(\vec{\Delta}^{\uparrow}, \vec{\Delta}^{\ddagger})$ such that

$$\begin{aligned} bit(i, \vec{\Delta}^{\uparrow}) &= 1 \text{ exactly when } \uparrow(f_i) \in \Delta \text{ and} \\ bit(i, \vec{\Delta}^{\ddagger}) &= 1 \text{ exactly when } \ddagger(f_i) \in \Delta \end{aligned}$$

Size dependency graphs are only explicitly built for the sibling-annotated SDG graph. As it contains labeled edges, we represent it by a list of nodes where each node has an adjacency list. The sibling annotations, e.g. $\{\mathtt{z} \xrightarrow{\downarrow} \mathtt{z}, \mathtt{z} \longrightarrow \mathtt{y}, \mathtt{d} \longrightarrow \mathtt{d}\}$ are represented straightforwardly as lists of tuples of type $(\mathit{Node} \times D^{\downarrow} \times \mathit{Node})$.

For the sake of computing SCCs we include for each node a reverse adjacency list containing all immediate predecessors.

Loop dependency graph LDG is used to get from balloon to anchor and from anchor to balloon, so here we again include a reverse adjacency list.

Dynamic dependency graph DDG is simply represented as an adjacency list in each node. When computing $\mathcal{D}_{\mathtt{e}}$ (cf. Figure 7.1 on page 102), we represent the sets in the *Symbol* domain as bit vectors.

Including call-site information for the `dp` nodes (to insert specialisation points, cf. Section 7.4) is done by adding a set σ to the cost triples, represented by a bit vector.

Call loop anchoring requires keeping track of which call sites have been selected as specialisation points. We store this information in a global bit vector.

9.2 Complexity

We will give some upper bounds on the amortised time and space complexity based on the following variables describing various aspects of the size of the program:

p	number of program expressions ("program size")
f	number of functions
n	total number of parameters in function definitions
a	maximum function arity
s	number of call sites
q	total number of arguments in function calls

9.2.1 Complexity of the analyses

Program syntax is created by traversing the program and making direct links for all function and parameter names. The complexity of this is

traversing program	$O(p)$
sorting function & parameter names	$O(f \cdot \log f + f \cdot a \cdot \log a)$
searching for parameters	$O(p \cdot \log a)$
searching for functions	$O(s \cdot \log f)$
Total:	$O((p+f \cdot a) \cdot \log a + (f+s) \cdot \log f)$

Size dependency operators $\mathcal{E}^{\mu}$ are computed in fixpoint iterations, and the result of a call to $\mathcal{E}^{\mu}$ is a dependency set, e.g. $\Delta = \{\uparrow(x_1), \ddagger(x_3)\}$. As dependency sets are represented by bit vectors, $\sqcap^{\downarrow}$ and $\sqcup^{\uparrow}$ are $O(a)$ time operations.

For each of f's parameters x we have either $\uparrow(x) \in \Delta$ or $\ddagger(x) \in \Delta$ or $\delta(x) \notin \Delta$, so the maximum chain length in the domain $(\mathit{DepSet}^{\mu})^{a} \to \mathit{DepSet}^{\mu}$ is $3 \cdot n$. As the iteration only changes these descriptions monotonically, and each change requires $O(p \cdot a)$ time to compute, the fixpoints are reached in time $O(p \cdot a \cdot 3 \cdot n)$. In the worst case, the fixpoint iteration must be performed at each argument, i.e. $O(q)$ times, yielding a total time of $O(p \cdot q \cdot a \cdot n)$ for computing $\mathcal{E}^{\mu}$.

Size dependency graph SDG is generated by applying the edge operator $\mathcal{SD}_{\mathbf{e}}$ recursively. Looking at the (∇_5) rule on page 83 we see that a new edge is inserted into an adjacency list $O(q \cdot a)$ times, because $|\Delta| \leq a$ for all dependency sets, specifically for $\Delta = \mathcal{E}^{\mu}\,[\![e_i]\!]\,\rho^{\mu}$.

Letting $\mathit{Adj} = D^{\uparrow} \times \mathit{Node} \times (\mathit{Node} \times D^{\downarrow} \times \mathit{Node})\,\mathit{List}$, we now spell out the operation $\mathit{insertEdge} : \mathit{Adj} \times \mathit{AdjList} \to \mathit{AdjList}$ that inserts SA-edges into the adjacency list of a node, indicating the time used during the entire analysis for each instruction:

```
$insertEdge((\delta_1, f_x, S), A) =$                      $O(q \cdot a)$
  for $(\delta_2, f'_x, S') \in A$                          $O(q \cdot a)$
    if $\delta_1 = \delta_2 \wedge f_x = f'_x$ then         $O(q \cdot a \cdot 2^{a^2})$
      $n \leftarrow |S'|$                                   $O(q \cdot a \cdot 2^{a^2})$
      for $(v \xrightarrow{\delta} y) \in S$                $O(q \cdot a \cdot 2^{a^2})$
        for $(v' \xrightarrow{\delta'} y') \in S'$          $O(q \cdot a \cdot 2^{a^2} \cdot a^2)$
          if $y = y' \wedge \delta = \delta' \wedge v = v'$ then   $O(q \cdot a \cdot 2^{a^2} \cdot a^2 \cdot a^2)$
            $n \leftarrow n - 1$                            $O(q \cdot a \cdot 2^{a^2} \cdot a^2 \cdot a^2)$
      if $n = 0$ then return $A$                            $O(q \cdot a \cdot 2^{a^2} \cdot a^2)$
  return $A \cup (\delta_1, f_x, S)$                        $O(q \cdot a)$
                                                    Total:  $O(q \cdot a^5 \cdot 2^{a^2})$
```

In total, the complexity of building the SDG is

traversing the program with $\mathcal{SD}_{\mathbf{e}}$	$O(p)$
creating sibling set S	$O(q \cdot a)$
creating SA-edges	$O(q \cdot a^5 \cdot 2^{a^2})$
Total:	$O(p + q \cdot a^5 \cdot 2^{a^2})$

The factor 2^{a^2} arises from the fact that given a call site, there are $O(a)$ possible sources and $O(a)$ possible destinations for a sibling edge, so there are $O(2^{a^2})$ different possible sibling edge sets S'.

The size of the SDG can be estimated to be

parameter nodes		n
`dp` nodes		f
Number of nodes,	$v =$	$n + f$
edges from parameter nodes		$O(q \cdot a)$
edges from `dp` nodes		$O(s)$
Number of edges,	$e =$	$O(q \cdot a + s)$

Computing SCCs can be done by an algorithm requiring (cf. Cormen, Leiserson, & Rivest, 1990)

computing reverse adjacency lists	$O(v+e)$
computing depth-first search twice	$O(v+e)$
creating SCCs	$O(v)$
Total:	$O(v+e)$

where v is the number of nodes and e the number of edges, so in program size terms the SCCs can be computed in time $O(n+f+q\cdot a+s)$.

Computing loop-costs involves computing the semi-ring product for an SCC C. We use an adapted version of Algorithm 5.5 from Aho et al. (1975), cf. Figure 9.1 on the facing page, for computing the total loop costs $c(x_i,x_i), x_i \in C$. In this algorithm, the parameter names in C have been numbered $x_1,\ldots,x_w$, and the time used during the entire analysis of all SCCs is measured in the number of nodes, v, and number of nodes in the largest SCC, u.

The $\cdot^*$ operation is simply computed as $c \leftarrow c \otimes c$ until a fixpoint is reached. As $|c_{\mathit{fix}}| \leq 2^{a^2}$, the fixpoint is reached using $O(2^{a^2})$ iterations involving the semi-ring product $\otimes$ of Definition 6.5 on page 85. That operator is computed naïvely as shown in Figure 9.2 on page 122.

In conclusion, we find that the main time spent in computing the semi-ring loop cost is in computing the $\cdot^*$ and $\otimes$ operations, which take time $O(v\cdot a^4\cdot 2^{4a^2}) + O(v\cdot u^2\cdot a^4\cdot 2^{3a^2}) = O(v\cdot a^4\cdot 2^{3a^2}\cdot(u^2+2^{a^2}))$. In terms of program size, this is $O(n\cdot a^4\cdot 2^{3a^2}\cdot(n^2+2^{a^2}))$ because this anchoring, i.e. loop costs without call-site information, is only relevant for ordinary parameter nodes.

Loop dependency graph LDG is created by travesing $O(v)$ nodes, and for each node traversing $O(2^{a^2})$ loop costs. For each loop cost (δ,S), $O(a^2)$ elements in S are traversed. Computing the LDG can thus be done in time $O(v\cdot 2^{a^2}\cdot a^2) = O((n+f)\cdot 2^{a^2}\cdot a^2)$.

```
for $i \in \{1,\dots,w\}$
  $c^0_{ii} \leftarrow \mathbf{1} \cup \bigcup_{x_i \xrightarrow[S]{\delta} x_i \in E} l\left(x_i \xrightarrow[S]{\delta} x_i\right)$          $O(v)$
for $i \in \{1,\dots,w\}$
  for $j \in \{1,\dots,w\}$                                          $O(v)$
    if $i \neq j$ then                                                $O(v \cdot u)$
      $c^0_{ij} \leftarrow \bigcup_{x_i \xrightarrow[S]{\delta} x_j \in E} l\left(x_i \xrightarrow[S]{\delta} x_j\right)$          $O(v \cdot u)$
for $k \in \{1,\dots,w\}$
  $c' \leftarrow \left(c^{k-1}_{kk}\right)^*$                                  $O(v)$
  for $i \in \{1,\dots,w\}$                                          $O(v)$
    $c'_{ik} \leftarrow c^{k-1}_{ik} \otimes c'$                               $O(v \cdot u)$
    for $j \in \{1,\dots,w\}$                                        $O(v \cdot u)$
      $c^k_{ij} \leftarrow c^{k-1}_{ij} \cup c'_{ik} \otimes c^{k-1}_{kj}$                $O(v \cdot u \cdot u)$
for $i \in \{1,\dots,w\}$
  $c(i,i) \leftarrow c^n_{ii}$                                       $O(v)$
```

Figure 9.1: Algorithm to compute total costs between vertices, adapted from Aho et al. (1975, algorithm 5.5).

```
$\gamma_1 \otimes \gamma_2 =$                                                        $O(1)$
  $A \leftarrow \{\}$                                                                $O(1)$
  for $(\delta_1, S_1) \in \gamma_1$                                                 $O(1)$
    for $(\delta_2, S_2) \in \gamma_2$                                               $O(2^{a^2})$
      $\delta^{\uparrow} \leftarrow \delta_1 \Box^{\uparrow} \delta_2$               $O(2^{a^2} \cdot 2^{a^2})$
      $S \leftarrow \{\}$                                                            $O(2^{a^2} \cdot 2^{a^2})$
      $(*$ compute $S_1 \circ S_2 *)$
      for $h_w \xrightarrow{\delta'_1} g_v \in S_1$                                  $O(2^{a^2} \cdot 2^{a^2})$
        for $g'_v \xrightarrow{\delta'_2} f_u \in S_2$                               $O(2^{a^2} \cdot 2^{a^2} \cdot a^2)$
          if $g_v = g'_v$ then                                                       $O(2^{a^2} \cdot 2^{a^2} \cdot a^2 \cdot a^2)$
            $\delta^{\downarrow} \leftarrow \delta'_1 \Box^{\downarrow} \delta'_2$   $O(2^{a^2} \cdot 2^{a^2} \cdot a^2 \cdot a^2)$
            $found \leftarrow$ false                                                 $O(2^{a^2} \cdot 2^{a^2} \cdot a^2 \cdot a^2)$
            for $h'_w \xrightarrow{\delta'} f'_u \in S$                              $O(2^{a^2} \cdot 2^{a^2} \cdot a^2 \cdot a^2)$
              if $\delta = \delta' \wedge h_w = h'_w \wedge f_u = f'_u$ then         $O(2^{a^2} \cdot 2^{a^2} \cdot a^2 \cdot a^2 \cdot a^2)$
                $found \leftarrow$ true                                              $O(2^{a^2} \cdot 2^{a^2} \cdot a^2 \cdot a^2 \cdot a^2)$
            if not $found$ then $S \leftarrow S \cup h_w \xrightarrow{\delta^{\downarrow}} f_u$   $O(2^{a^2} \cdot 2^{a^2} \cdot a^2 \cdot a^2)$
      $(*$ add result to $A$ if $(\delta^{\uparrow}, S) \notin A *)$
      $found \leftarrow$ false                                                       $O(2^{a^2} \cdot 2^{a^2})$
      for $(\delta', S') \in A$                                                      $O(2^{a^2} \cdot 2^{a^2})$
        if $\delta^{\uparrow} = \delta'$ then                                        $O(2^{a^2} \cdot 2^{a^2} \cdot 2^{a^2})$
          $n \leftarrow |S'|$                                                        $O(2^{a^2} \cdot 2^{a^2} \cdot 2^{a^2})$
          for $g_y \xrightarrow{\delta''} f_x \in S$                                 $O(2^{a^2} \cdot 2^{a^2} \cdot 2^{a^2})$
            for $g'_y \xrightarrow{\delta'''} f'_x \in S'$                           $O(2^{a^2} \cdot 2^{a^2} \cdot 2^{a^2} \cdot a^2)$
              if $\delta'' = \delta''' \wedge g_y = g'_y \wedge f_x = f'_x$ then     $O(2^{a^2} \cdot 2^{a^2} \cdot 2^{a^2} \cdot a^2 \cdot a^2)$
                $n \leftarrow n - 1$                                                 $O(2^{a^2} \cdot 2^{a^2} \cdot 2^{a^2} \cdot a^2 \cdot a^2)$
          if $n = 0$ then $found \leftarrow$ true                                    $O(2^{a^2} \cdot 2^{a^2} \cdot 2^{a^2})$
      if not $found$ then $A \leftarrow A \cup (\delta^{\uparrow}, S)$               $O(2^{a^2} \cdot 2^{a^2})$
  return $A$                                                                         $O(1)$
                                                                           Total :   $O(a^4 \cdot 2^{3a^2})$
```

Figure 9.2: Algorithm for computing the semi-ring product

Bounded anchoring, step number 4 of the algorithm, entails traversing $O(v)$ nodes and their $O(e)$ predecessors. For each node, also $O(2^{a^2})$ balloons and $O(2^{a^2} \cdot a^2)$ anchors may be checked. This step is iterated at most v times (each iteration must increase the number of nodes marked B by at least 1), so all the anchoring can be computed in time $O(v \cdot (e + v + v \cdot 2^{a^2} \cdot a^2)) = O(v \cdot (e + v \cdot 2^{a^2} \cdot a^2))$, i.e. $O((n+f) \cdot (q \cdot a + s + (n+f) \cdot 2^{a^2} \cdot a^2)) = O((n+f) \cdot q \cdot a + (n+f) \cdot s + (n+f)^2 \cdot 2^{a^2} \cdot a^2)$.

Dynamic dependency graph DDG is created using the dynamic dependency function $\mathcal{D}$ and the edge generating operator $\mathcal{DD}_{\mathbf{e}}$. During the computation of $\mathcal{D}_{\mathbf{e}}$, set unions are bit vector operations taking time $O(a+f)$, so the complexity for creating the DDG is

traversing the program with $\mathcal{D}_{\mathbf{e}}$	$O(p \cdot (a+f))$
creating edges to $f_{\mathbf{R}}$ nodes	$O(f \cdot (a+f))$
creating edges to parameters (cf. Figure 7.2)	$O(q \cdot (a+f) \cdot n)$
Total:	$O((f + q \cdot n + p) \cdot (a+f))$

Including call-site information is only done for the **dp** nodes, of which there are f (one per function).

This increases the complexity for computing the semi-ring product from $O(a^4 \cdot 2^{3a^2})$ to $O(a^4 \cdot 2^{3a^2} \cdot 2^{3s} \cdot s)$, as the domain of the **for**-loops is increased by a factor of 2^s and computing unions of call site sets is done in time $O(s)$. Similarly, a conservative approximation gives that the $\cdot^*$ operator now requires $O(2^{a^2} \cdot 2^s)$. On the other hand, in the complexity of the algorithm for total costs (Figure 9.1) v is replaced by f, so the complexity for computing the call-site-augmented total costs is $O(f \cdot a^4 \cdot 2^{4a^2} \cdot 2^{4s} \cdot s) + O(f \cdot u^2 \cdot a^4 \cdot 2^{3a^2} \cdot 2^{3s} \cdot s) = O(f \cdot a^4 \cdot 2^{3a^2} \cdot 2^{3s} \cdot s \cdot (u^2 + 2^{a^2} \cdot 2^s))$, i.e. in terms of program size it is $O(f \cdot a^4 \cdot 2^{3a^2} \cdot 2^{3s} \cdot s \cdot (f^2 + 2^{a^2} \cdot 2^s))$

Specialisation-point insertion and dynamic cascading, step number 6 of the algorithm, is performed by traversing $O(2^{a^2} \cdot 2^s)$ **dp** balloon nodes and for each such node marked D testing with bit vector operations in $O(s)$ time whether there is a specialisation point. If a specialisation point is inserted for function f, the function *cascade*, shown in Figure 9.3, is called on $f_{\mathbf{R}}$; it

```
 1  $cascade(x) =$                                   $O(v)$
 2    if $mark(x) \neq D$ then                       $O(v)$
 3      $mark(x) \leftarrow D$                       $O(v)$
 4      for $y \in succ(x)$                          $O(v)$
 5        if $mark(y) \neq D$ then                   $O(e)$
 6          $cascade(y)$                             $O(v)$
 7      for $z \in anchors(x)$                       $O(v)$
 8        $b \leftarrow balloon(z)$                  $O(v \cdot 2^{a^2})$
 9        if $mark(b) \neq D$ then                   $O(v \cdot 2^{a^2})$
10          $wellanchored \leftarrow$ false          $O(v \cdot 2^{a^2})$
11          for $a \in anchors(b)$                   $O(v \cdot 2^{a^2})$
12            if $mark(a) = B$ then                  $O(v \cdot 2^{a^2} \cdot 2^{a^2})$
13              $wellanchored \leftarrow$ true       $O(v \cdot 2^{a^2} \cdot 2^{a^2})$
14          if not $wellanchored$ then               $O(v \cdot 2^{a^2})$
15            $cascade(b)$                           $O(v)$
                                            Total:   $O(e + v \cdot 2^{2a^2})$
```

Figure 9.3: Algorithm for cascading the consequences of inserting specialisation points

changes the mark of its argument to D and cascades the effect.

Its complexity is calculated in steps:

1. Consider line 3; it changes a non-D mark (due to line 2) to a D mark; this can be done at most v times, making lines 3, 4 and 7 $O(v)$.

2. As line 4 is executed at most once for each node, each edge in the graph is drawn from $succ(x)$ at most once, so line 5 is executed at most e times.

3. Each parameter can be an anchor for $O(2^{2^a})$ loops, yielding the complexities for lines 8–14.

4. If *we assume that cascade is only ever called with arguments that are not marked D*, line 1 and 2 are executed exactly as many times as line 3, i.e. $O(v)$ times.

5. This assumtion holds for lines 6 and 15, so they too are executed $O(v)$ times.

The call loop anchoring and cascading step is iterated at most s times, as each iteration inserts at least one specialisation point, so the complexity of this step is $O(s^2 \cdot 2^{a^2} \cdot 2^s) + O(e + v \cdot 2^{2a^2}) = O(s^2 \cdot 2^{a^2} \cdot 2^s + e + v \cdot 2^{2a^2})$. In terms of program size, this is $O(s^2 \cdot 2^{a^2} \cdot 2^s + n + f + q \cdot a + s + (n + 2f) \cdot 2^{2a^2}) = O(s^2 \cdot 2^{a^2} \cdot 2^s + q \cdot a + (n + f) \cdot 2^{2a^2})$.

The entire binding-time analysis can now be estimated by adding the complexities of the individual steps. It is reasonable to assume that the complexity variables f, n, s, q are proportional to the program size, p. Further, it can be argued that the maximum function arity a should be regarded as a constant: twice as long programs do not normally have functions of twice the arity.

Summing up, the complexity measured in terms of the individual complexity variables, in terms of only p and a, and finally regarding a as constant are shown in Table 9.1 on the next page. All these ghastly expressions are of course a result of conservative and worst-case approximations, and we can make several observations about them:

- Terminating programs are detected in time $O(p^3)$, and disregarding call-site loop costs and specialisation point cascading, the BTA is also computed in time $O(p^3)$.

- Care should be taken in designing good heuristics for computing $\mathcal{E}^\mu$, creating the SDG and DDG and for computing the loop-costs, so that the worst-case complexity of $O(p^3)$ can be avoided for typical programs.

	General	*p and a only*	*a const*
syntax	$O((p+fa)\log a+(f+s)\log f)$	$O(p^2\cdot\log a+p\cdot\log p)$	$O(p^2)$
computing $\mathcal{E}^\mu$	$O(p\cdot q\cdot a\cdot n)$	$O(p^3\cdot a)$	$O(p^3)$
creating SDG	$O(p+q\cdot a^5\cdot 2^{a^2})$	$O(p\cdot a^5\cdot 2^{a^2})$	$O(p)$
finding SCCs	$O(n+f+q\cdot a+s)$	$O(p\cdot a)$	$O(p)$
loop-costs	$O(n\cdot a^4\cdot 2^{3a^2}\cdot(n^2+2^{a^2}))$	$O(p^3 2^{3a^2}a^4+p2^{4a^2}a^4)$	$O(p^3)$
creating LDG	$O((n+f)\cdot 2^{a^2}\cdot a^2)$	$O(p\cdot 2^{a^2}\cdot a^2)$	$O(p)$
BA	$O((n+f)(qa+s+(n+f)2^{a^2}a^2))$	$O(p^2\cdot 2^{a^2}\cdot a^2)$	$O(p^2)$
creating DDG	$O((f+q\cdot n+p)\cdot(a+f))$	$O(p^3+p^2\cdot a)$	$O(p^3)$
call loop costs	$O(fa^4 2^{3a^2}2^{3s}s(f^2+2^{a^2}2^s))$	$O(2^{4p}\cdot p^2\cdot 2^{4a^2}\cdot a^4)$	$O(2^{4p}p^2)$
sp cascading	$O(s^2 2^{a^2}2^s+qa+(n+f)2^{2a^2})$	$O(2^p p^2 2^{a^2}+p2^{2a^2})$	$O(2^p p^2)$

Table 9.1: Worst-case complexity for the various phases

- In fact, approximating the squared maximum SCC size u^2 in the complexity of computing the loop-cost (cf. Figure 9.1 on page 121) to be proportional with p^2 is rather conservative. Roughly, u corresponds to the largest set of mutually recusive functions in the program; in typical programs most SCCs will be singleton, and few SCCs are expected to have more than a handful of nodes. Thus it can be argued that the complexity of computing the total loop-costs is linear in practise.

- Further, the factor of f in the complexity for creating the $g_{\mathbf{R}} \to f_i$ edges in the DDG is a worst-case approximation which is based on function arguments containing nested calls to all functions in the program. In practise this can be regarded as a small constant, yielding an $O(p^2)$ complexity for creating the DDG.

- The maximum function arity occurs exponentially in several places. If we look at the algorithms, we see that the factors $2^{c\cdot a^2}$ arise from the sets of sibling edges and all their possible combinations of source and destination nodes. However, in typical real-world programs—at least those written by humans—a function parameter does not interact with *all* the other parameters, so it can be argued that a is a "very" small integer.

- Even though the arity a is considered constant, the algorithms should be designed carefully, to avoid introducing large constant factors that could become a problem in practice.
- The exponential factor 2^s for computing call-site augmented total costs and performing specialisation point cascading is due to the sets of call sites that must be created and checked. This could be a problem in practice and should be addressed in future research.

9.2.2 Improving semi-ring computation speed for `dp` nodes

We will not attempt to address the problem of the exponential complexity in depth, but instead leave that for future research. Rather, we will make some observations concerning the algorithms previously described, and suggest some ad-hoc optimisations without formally proving their correctness.

First we note that the exponential complexity *is only related to specialisation-point insertion, i.e. only the computations for* `dp` *nodes.* This also implies that all cost triples of interest have increasing first components: $(\uparrow, S, \sigma)$.

Second, it is clear from the preceding discussion that the problem lies in calculating the semi-ring product (cf. Figure 9.2 on page 122), where we have three nested for loops:

```
⋮
for $(\delta_1, S_1, \sigma_1) \in \gamma_1$
  for $(\delta_2, S_2, \sigma_2) \in \gamma_2$
    ⋮
    for $(\delta', S', \sigma') \in A$
      ⋮
```

each of which draw elements from costs of size $O(2^{2^a} \cdot 2^s)$. Thus, it seems sensible to try and keep the typical size of a cost, especially the number of different call site set components, small.

Third, observe how the results of the computation, the total costs, are used: each total cost $\{(\uparrow, S_1, \sigma_1), \ldots, (\uparrow, S_n, \sigma_n)\}$ is searched for an i where there is no anchor $(u \xrightarrow{\downarrow} u) \in S_i$. In this case, an $s \in \sigma_i$ is selected for specialisation-point insertion. This has several implications:

- Cost triplets $(\uparrow, \{(u \xrightarrow{\downarrow} u), \ldots\}, \sigma)$ containing anchors are not interesting for the final result.

- During computation of the total costs we traverse products of sets of cost triplets $\{c_1, \ldots, c_m\} \times \{c'_1, \ldots, c'_n\}$ and compute the union of a function $\bigcup_{i,j} f(c_i, c'_j)$. We can say c_i *supersedes* c_k if $f(c_i, c'_j)$ is "at least as interesting" as $f(c_k, c'_j)$ for all j. In other words, whenever c_k can cause the insertion of a specialisation point $s \in X$, then c_i is certain also to cause insertion of a specialisation point $s' \in X$. During the computations, we can then discard triplets from a set that are superseded by others from the same set. For instance,

- For a total cost $\{\ldots, (\uparrow, S_1, \sigma), \ldots, (\uparrow, S_2, \sigma), \ldots\}$, where $S_1 = \{e_1, \ldots, e_{i-1}, g_y \xrightarrow{\downarrow} f_x, e_{i+1}, \ldots, e_n\}$ and $S_2 = \{e_1, \ldots, e_{i-1}, g_y \xrightarrow{\mp} f_x, e_{i+1}, \ldots, e_n\}$, we can remove the $(\uparrow, S_1, \sigma)$ triplet. Most important though,

- For a cost with two triplets $\{\ldots, (\uparrow, S, \sigma_1), \ldots, (\uparrow, S, \sigma_2), \ldots\}$ where $\sigma_1 \supseteq \sigma_2$, we can remove $(\uparrow, S, \sigma_1)$.

In general, we can define "c_1 is superseded by c_2," as a partial ordering $c_1 \preceq c_2$ by

$$\begin{aligned} (\delta_1, S_1, \sigma_1) \preceq (\delta_2, S_2, \sigma_2) \quad &\Leftrightarrow \quad \delta_1 \sqsubseteq \delta_2 \wedge S_1 \succeq S_2 \wedge \sigma_1 \supseteq \sigma_2, \text{ where} \\ S_1 \succeq S_2 \quad &\Leftrightarrow \quad \forall g_y \xrightarrow{\delta_2} f_x \in S_2 \; \exists g_y \xrightarrow{\delta_1} f_x \in S_1 : \delta_2 \sqsubseteq \delta_1 \end{aligned}$$

and then use this to modify the semi-ring product algorithm in Figure 9.2 on page 122. As the result of the product is always used in a union with another set, we enter this set explicitly as a parameter, as shown in Figure 9.4 on the next page. The trick to keep the triplet set A small is to remove any triplets from A that are superseded by a new cost triplet that is to be added, and only add it if it is not superseded by a triplet already in A.

```
$A \cup (\gamma_1 \otimes \gamma_2) =$
  for $(\delta_1, S_1, \sigma_1) \in \gamma_1$
    for $(\delta_2, S_2, \sigma_2) \in \gamma_2$
      $\vdots$  (compute result $(\delta^{\uparrow}, S, \sigma)$ as in Figure 9.2)
      (* remove any $(\delta', S', \sigma')$ from $A$ where $(\delta^{\uparrow}, S, \sigma) \succ (\delta', S', \sigma')$ *)
      (* add $(\delta^{\uparrow}, S, \sigma)$ to $A$ unless some $(\delta', S', \sigma') \in A$ supersedes it *)
      $leqexists \leftarrow$ false
      for $(\delta', S', \sigma') \in A$
        if $(\delta^{\uparrow}, S, \sigma) \succ (\delta', S', \sigma')$ then $A \leftarrow A \setminus \{(\delta', S', \sigma')\}$
        if $(\delta^{\uparrow}, S, \sigma) \preceq (\delta', S', \sigma')$ then $leqexists \leftarrow$ true
      if not $leqexists$ then $A \leftarrow A \cup \{(\delta^{\uparrow}, S, \sigma)\}$
  return $A$
```

Figure 9.4: Optimised semi-ring product algorithm

Call site sets represented as bit vectors can simply be compared in linear time using bitlogical operations. However, computing the sibling edge set relations $S_1 \succ S_2$ and $S_1 \preceq S_2$ efficiently requires that we keep them sorted, e.g. lexicographically by (u, v, δ) where $S = \{u_1 \xrightarrow{\delta_1} v_1, \ldots, u_n \xrightarrow{\delta_n} v_n\}$, and also that for every node pair (u, v) there is at most one $u \xrightarrow{\delta} v$ in each sibling edge set. This can be accomplished by removing $u \xrightarrow{\ddagger} v$ from a set S if $u \xrightarrow{\downarrow} v \in S$.

When these conditions hold, the function deciding for two sibling edge sets S_1 and S_2 whether $S_1 \prec S_2$, $S_1 = S_2$, $S_1 \succ S_2$, or S_1 is unrelated to S_2 can be computed as shown in Figure 9.5 on the following page.

With this optimised algorithm we conjecture that the specialisation point insertion is performed in polynomial time.

```
(* decide $S_1 \prec S_2, S_1 = S_2, S_1 \succ S_2$ or $S_1$ is unrelated to $S_2$ *)
$compare(S_1, S_2) =$
  $leq \leftarrow$ true    $ltexists \leftarrow$ false
  $geq \leftarrow$ true    $gtexists \leftarrow$ false
  $eq \leftarrow$ true
  $List\ es_1 \leftarrow sorted(S_1)$
  $List\ es_2 \leftarrow sorted(S_2)$
  while not $empty(es_1) \wedge$ not $empty(es_2)$
    $e_1 \equiv (u_1 \xrightarrow{\delta_1} v_1) \leftarrow head(es_1)$
    $e_2 \equiv (u_2 \xrightarrow{\delta_2} v_2) \leftarrow head(es_2)$
    if $u_1 = u_2 \wedge v_1 = v_2$ then
      if $\delta_1 \sqsubset \delta_2$ then
        $eq \leftarrow$ false    $geq \leftarrow$ false    $ltexists \leftarrow$ true
      else if $\delta_1 \sqsupset \delta_2$ then
        $eq \leftarrow$ false    $leq \leftarrow$ false    $gtexists \leftarrow$ true
    else
      $eq \leftarrow$ false
    if $e_1 \leq e_2$ then $es_1 \leftarrow tail(es_1)$
    if $e_1 \geq e_2$ then $es_2 \leftarrow tail(es_2)$
  if not $empty(es_1)$ then
    $eq \leftarrow$ false    $geq \leftarrow$ false
  if not $empty(es_2)$ then
    $eq \leftarrow$ false    $leq \leftarrow$ false
  if $leq \wedge ltexists$ then return $\prec$
  else if $eq$ then return $=$
  else if $geq \wedge gtexists$ then return $\succ$
  else return incomparable
```

Figure 9.5: Algorithm for computing the supersede relation on sibling edge sets

9.3 Results

A prototype of the analyses described in this paper has been implemented in Haskell, which is ideal for prototyping because the semantic equations given in the text in many cases can be directly used as the Haskell program. Thus, efficient datastructures have not been used everywhere, and for this reason we will not report on actual running times. However, using the speed-improved semi-ring computation (cf. Section 9.2.2 on page 127), binding-time analysing the slowest example (*int-loop*) takes less than half a minute.

We have collected a suite of example programs in Appendix B, including several from the work of Lindenstrauss and Sagiv (1997). They are written in Scheme syntax (Abelson et al., 1998) intended as input to the Similix specialiser (Bondorf, 1990; Bondorf & Jørgensen, 1993), but are translated into a Haskell-like syntax before they are passed to the prototype analyser.

The example programs are all first order, so functions like `fold` and `map` use fixed functions for folding and mapping. Wherever it is vital for termination we have written natural numbers in unary notation, i.e. as list lengths, as the builtin integer type is not a well-founded domain. Furthermore, as we lose all size information for functions returning two values pointed to by a cons cell, we have slightly rewritten the sorting functions. For instance, `merge (split xs)` is changed into `splitmerge xs [] []`, where `splitmerge` splits `xs` into two lists and calls `merge` on them.

The results of the termination and binding-time analyses, given binding-time patterns for each goal function, are shown in Tables 9.2–9.5. By looking at each program, we have listed the optimal analysis result, i.e. the fewest generalisations and specialisation point insertions necessary to guarantee termination of specialisation, and whether the prototype is able to achieve this or is more conservative.

Even though the sorting functions have been rewritten as previously described, the analyser is not able to detect that they terminate. Looking at the programs it becomes evident why: during the reordering of the list, our rather crude size approximations lose track of the sizes. It is the `cons` operations found in `splitmerge` of *mergesort,* `part` of *quicksort* and `remove` of *minsort* that make the analyser think there are dangerous parameter increases in the main loops. One could patch on this problem by passing around a mea-

	Termination analysis			*Binding-time analysis*			
		Opt.	*Conser-*	*Goal*		*Opt.*	*Conser-*
Program	*result*	*result*	*vative*	*BT*	*result*	*result*	*vative*
contrived-1	T	T	no	s d			no
contrived-2	QT	QT	no	s d	SP	SP	no
list	T	T	no	s			no
fold	T	T	no	d s			no
				s d	SP	SP	no
map	T	T	no	s			no
naiverev	T	T	no	s			no
deeprev	T	T	no	s			no
append	T	T	no	s d			no
mergelists	T	T	no	s d	SP	SP	no
addlists	T	T	no	s d			no
revapp	T	T	no	s d			no
permute	NT	T	yes	s	SP, G	SP	yes
add	T	T	no	s d	SP, G	SP, G	no
				d s			no
badd	NT	QT	yes	s d	SP	SP	no
mul	T	T	no	s d	SP	SP	no
disjconj	T	T	no	s			no
duplicate	T	T	no	s			no
nestimeql	NT	QT	yes	s	SP, G	SP	yes
evenodd	T	T	no	s			no
lte	T	T	no	s d			no
				d s	SP	SP	no
member	T	T	no	d s			no
				s d	SP	SP	no

T = terminating, QT = quasiterminating, NT = nonterminating,
s = static, d = dynamic,
SP = insert specialisation point(s),
G = generalise variable(s) to ensure termination

Table 9.2: Results of termination and binding-time analyses, part I.

	Termination analysis			*Binding-time analysis*			
		Opt.	*Conser-*	*Goal*		*Opt.*	*Conser-*
Program	*result*	*result*	*vative*	*BT*	*result*	*result*	*vative*
ordered	T	T	no	s			no
overlap	T	T	no	s d	SP	SP	no
select	T	T	no	s			no
subsets	T	T	no	s			no
anchored	T	T	no	s d d s	 SP, G	 SP, G	no no
letexp	NT	NT	no	s s	SP, G	SP, G	no
thetrick	NT	T	yes	s d d s	SP SP, G	SP SP, G	no no
intlookup	QT	QT	no	d s	SP	SP	no
nolexicord	T	T	no	s s s s s s s s s s s d	 SP	 SP	no no
decrease	T	T	no	s			no
equal	QT	QT	no	s	SP	SP	no
increase	NT	NT	no	s	SP, G	SP, G	no
nestdec	T	T	no	s			no
nesteql	QT	QT	no	s	SP	SP	no
nestinc	NT	NT	no	s	SP, G	SP, G	no
sp1	QT	QT	no	s d	SP	SP	no
shuffle	NT	T	yes	s	SP, G		yes
assrewrite	NT	T	yes	s	SP, G		yes
game	T	T	no	s s s			no
vangelder	QT	QT	no	s d	SP	SP	no
power	T	T	no	d s	SP	SP	no

T = terminating, QT = quasiterminating, NT = nonterminating,
s = static, d = dynamic,
SP = insert specialisation point(s),
G = generalise variable(s) to ensure termination

Table 9.3: Results of termination and binding-time analyses, part II.

	Termination analysis			*Binding-time analysis*			
		Opt.	*Conser-*	*Goal*		*Opt.*	*Conser-*
Program	*result*	*result*	*vative*	*BT*	*result*	*result*	*vative*
binom	T	T	no	s d			no
				d s	SP	SP	no
ack	T	T	no	s d	SP	SP	no
gcd-1	T	T	no	s d	SP	SP	no
gcd-2	QT	T	yes	s d	SP	SP	no
mergesort	NT	T	yes	s	SP, G		yes
quicksort	NT	T	yes	s	SP, G		yes
minsort	NT	T	yes	s	SP, G		yes
reach	QT	QT	no	d d s	SP	SP	no
				s d s	SP	SP	no
graphcolour-1	QT	T	yes	d s	SP	SP	no
				s d	SP	SP	no
graphcolour-2	QT	T	yes	d s	SP	SP	no
				s d	SP	SP	no
graphcolour-3	T	T	no	d s	SP	SP	no
				s d	SP	SP	no
match	T	T	no	s d	SP	SP	no
strmatch	T	T	no	s d	SP, G	SP, G	no
				d s			no
turing	NT	NT	no	s d	SP	SP	no
lambdaint	NT	NT	no	s	SP, G	SP, G	no
int-loop	T	T	no	s s d			no
				s d d	SP	SP	no
int-while	NT	NT	no	s d	SP	SP	no

T = terminating, QT = quasiterminating, NT = nonterminating,
s = static, d = dynamic,
SP = insert specialisation point(s),
G = generalise variable(s) to ensure termination

Table 9.4: Results of termination and binding-time analyses, part III.

	Termination analysis			*Binding-time analysis*			
		Opt.	*Conser-*	*Goal*		*Opt.*	*Conser-*
Program	*result*	*result*	*vative*	*BT*	*result*	*result*	*vative*
int-dynscope	NT	NT	no	s d	SP, G	SP, G	no

T = terminating, QT = quasiterminating, NT = nonterminating,
s = static, d = dynamic,
SP = insert specialisation point(s),
G = generalise variable(s) to ensure termination

Table 9.5: Results of termination and binding-time analyses, part IV.

sure of the list lengths (and decreasing them whenever the lists got shorter), but that would not be a natural way to write the sorting functions.

The function for rewriting an expression with an associative operator, *assrewrite*, cannot be proven by the analyser to terminate. This should come as no surprise, as it requires an advanced size measure not only keeping track of the number of cons nodes but also the structure of the syntax tree. Somewhat similarly, to be able to detect the *badd* program to be terminating requires considering the sum of the length of the arguments `x` and `y`, which seems non-trivial.

The two versions of the greatest common divisor function, *gcd–1* and *gcd–2* demonstrate the importance of not "mixing" different variables unnecessarily, cf. Section 10.3.1 on page 144.

The graph colouring program has first been coded up in *graphcolour–1* to use The Trick in the `colorrest` function: The nested call to `colornode` (line 44) returns a coloured node that is represented by a cons pair containing the colour and the node. If the node is dynamic, the colour of the node will also be dynamic (recall we have no partially static data). However, by checking the colour against the finite list of available colours in the call to `colorrest-thetrick` (line 53), we avoid making `ncs` dynamic in the next call (line 62).

However, the analyser is unable to detect that the car of the return value of the call `colornode ncs ...` (line 44) is never greater than that of `ncs`, so `ncs` is lost as an anchor in the mutually recursive call to `colorrest-thetrick` (line 54). We can remedy this by replacing the call to `colornode` by a call to a tail recursive function `colornoderest`, that basically consists of `colornode` followed by the lines following the call to it (lines 46–56).

Using the new program, *graphcolour–2*, the analyser can detect that `ncs` is static and is decreased in the call to `colorrest-thetrick`, but it cannot see that the value of `ncs` is passed back to `colorrest` again. This shows that using The Trick where it is not necessary (i.e. the test is not dynamic) *can in fact be harmful* to the termination analysis. Naturally, as the test is static, we can remove `colorrest-thetrick`, obtaining the program *graphcolour–3*, which is correctly analysed to be terminating.

However, some forms of The Trick, e.g. moving a dynamic test further out into the context, can aid the analysis in detecting termination as can be seen by the different results obtained for function `f` and `g` in program *thetrick*.

The example program *nestimeql* shows one shortcoming of the size approximation function $\mathcal{E}^{\uparrow}$: it cannot detect that the size of the return value of a call `immatcopy x` is the same as the size of `x`, resulting in conservative generalisation. Similar problems occur with the `revapp` call in *permute* (line 14), and the `reverse` call in *shuffle*.

The remaining examples are handled without resulting in overconservative results, notably including several interpreters.

Chapter 10

Conclusion

10.1 Current work

The algorithm presented in this paper has been implemented as a prototype and experiments have shown it to produce reasonably good results. Further experiments will be made to determine how successful it is in correctly determining bounded variation properties and specialisation points.

10.2 Related work

Andersen and Holst (1996) presented a successful analysis which includes the higher-order case. The first-order part of their analysis is very similar to ours: they also employ a size dependency analysis prior to an anchoring algorithm, and their transitive transition closure operation corresponds roughly to our semi-ring algorithm. The main differences are that we clearly separate the modalities by conceptually using both $\mathrm{SDG}^{\uparrow}$ and $\mathrm{SDG}^{\downarrow}$, and we include an algorithm for inserting all necessary specialisation points, an essential ingredient in making specialisation of quasi-terminating programs terminate.

This in turn requires that we handle cases where a BSV variable used as anchor becomes dynamic due to an inserted specialisation point. By stating explicitly in a novel way the loop dependencies in the LDG, we avoid costly reiteration of the semi-ring algorithm.

We also distinguish between $\underset{=}{\uparrow}$ and $\overset{=}{\downarrow}$, due to nested calls (cf. page 27), enabling less conservative size estimates for nested calls like

```
f x y d = if d > 0 then f (max x y) y (d - 1) else x
max u v = if u > v then u else v
```

In their analysis, assuming `d` is dynamic, this would give rise to an increasing transition and generalisation of `x`, whereas we just record an equality transition, leading to no generalisation.

Further, we put more emphasis on describing explicitly our algorithms and show how the worst-case complexity arises, as an aid to developing efficient implementations. It is not clear what the worst-case complexity of their analyses is, but we expect it also to be cubic like ours.

Das' (1998) work concerns mainly termination analysis for partial evaluation of real-world C programs, and is thus not directly comparable with our work. However, it is reasonable to assume that the techniques could be transferred to BTA for functional programs. The main differences between the essence of Das' and our approach is that

- Das deals directly with control dependencies, whereas we emphasize the relationships between increasing and decreasing variables. In fact, it turns out that to avoid too conservative binding-time divisions due to dynamic conditionals, he adds analyses similar to call loop anchoring to detect *grounded loops* and *grounded flow cycles*, which seems to indicate that this is the key property to consider in termination analysis for partial evaluation.

- Applying The Trick, for instance transforming

  $$\begin{array}{lll} & & ss = 10; \\ s = d & \text{into} & \textbf{while}\ (ss \mathbin{!=} d)\ ss = ss - 1; \\ & & s = ss; \end{array}$$

  one has explicitly introduced a dynamic control dependency (from $ss \mathbin{!=} d$) to a BSV variable (ss). This will cause Das' control dependency approach to conservatively generalise the variable, cancelling the effect that was intended with The Trick. His solution is to require the user to explicitly annotate occurrences of The Trick in the code. Although this is a viable approach, we prefer good heuristics that can handle The Trick automatically in typical cases.

- Das only supplies a *conditional* termination guarantee: the resulting binding-time division may lead to static-infinite computations, whereas we detect variables that are truly BSV. As partial evaluation is over-strict, we believe that a full termination guarantee should be given, cf. Section 2.2 on page 29.

- Additionally, the part of Das' work handling functional programs only makes use of weak bounded anchoring, and will thus handle lexicographic ordering conservatively. In fact, his condition for detecting an unsafe, i.e. possibly non-BSV, variable v is (Das, 1998, Algorithm 3, p. 162)

```
  If    v gets a value from a non-BSV variable
  or    (there exists an increasing loop v → ··· → v    and
        there exists no sibling variable w which decreases along all
        loops w → ··· → w)
  then  v must be generalised to ensure termination
```

  This also prevents it from handling the interpreter example, because there exists an increasing loop $\texttt{eval}_{\texttt{ns}} \to \cdots \to \texttt{eval}_{\texttt{ns}}$, but $\texttt{eval}_{\texttt{e}}$ cannot be used as an anchor in Das' condition, because it does not decrease along the loop for interpreting function calls ($\texttt{eval}_{\texttt{e}}$ is reset to some function body which is possibly larger).

- However, in Das' approach, only one SDG graph is used for both the $\text{SDG}^{\uparrow}$ and $\text{SDG}^{\downarrow}$ because only tail-recursive programs are handled. Thus, both edges labeled ↑ and ↓ exist in the same graph, and context-free language reachability allows increasing edges $\xrightarrow{\uparrow}$ to "cancel out" decreasing edges $\xrightarrow{\downarrow}$ in cases where it is safe. This enables a more precise analysis e.g. of functions that return several values packaged with a cons cell which is later taken apart. There does not seems to be anything that in principle prevents an extension of this cancelling effect to SDG$^{\uparrow}$ and SDG$^{\downarrow}$.

10.2.1 Pessimistic vs. optimistic BTA

One aspect that makes our approach stand out is that we start by assuming all variables to be "unsafe," i.e. marked '$\perp$', and we only promote to 'B' when we have a guarantee (an anchor or dominator) that the variable is in fact BSV. This can be termed a 'safe' or 'pessimistic' approach. Other approaches (Andersen & Holst, 1996; Das & Reps, 1996; Das, 1998) are 'optimistic' in the sense that they start by assuming all variables to be BSV, and then re-classify as dynamic variables which seem to be non-BSV. We do not expect there to be any difference in power, i.e. that the safe approach generalises variables too conservatively[1], but this has not been formally shown. One advantage of our safe approach is that one can "bail out" of the promotion process half way and still obtain a correct, albeit more conservative, result.

10.2.2 Partial evaluation of interpreters

Partial evaluation is especially well-suited for specialising away the interpretive overhead in interpreters (Thibault & Consel, 1997; Thibault, Marlet, & Consel, 1997; Jones, 1996), and in this context it is vital to be able to identify as BSV the case where a variable ($\texttt{eval}_\texttt{e}$) is reset to a value computed from another BSV variable ($\texttt{eval}_\texttt{p}$).

10.2.3 Proving termination by lexicographic ordering

Proving termination of programs has also been done by finding a tuple of parameters that can be shown always to decrease in some lexicographic ordering (Nielson & Nielson, 1996), which must somehow be supplied by hand or by other analyses (Giesl, 1995; Brauburger, 1997). The method presented in this paper includes detection of termination by lexicographic ordering,

[1] Das (1998, Example 12) claims to have found an example which is treated differently by the two approaches, but our analysis does in fact give the same result as his.

and this relies on the use of *strong* bounded anchoring (cf. Condition 6.7): weak bounded anchoring does not allow lesser significant parameters in the lexicographic order to be reset to a greater value.

We can show that bounded anchoring is in fact strictly stronger than the lexicographic ordering approach. Consider the following

Example 10.1 (No lexicographic ordering)

```
f dp ($a_1$, $b_1$) ($a_2$, $b_2$) ($a_3$, $b_3$) = ...
  if ... then $f^a$ (dp+1) ($b_1$-1, $a_1$-1) ($a_2$-1, $b_2$-1) ($b_3$-1, $a_3$-1)
         else $f^b$ (dp+1) ($b_1$-1, $a_1$-1) ($b_2$-1, $a_2$-1) ($a_3$-1, $b_3$-1)
```

Note that the parameters have been tupled merely to ease the understanding; they could have all been curried.

It can be shown by bounded anchoring that `f` terminates (i.e. that `dp` is BV):

1. All `a`'s and `b`'s are bounded (there are no increasing operations).
2. The LDG is shown in Figure 10.1. From this we conclude that all `dp` loops are well-anchored, and that `f` terminates.

On the other hand, there exists no tuple of parameters from $\{a_1, a_2, a_3, b_1, b_2, b_3\}$ such that they decrease for all loops $f \longrightarrow f$ in some lexicographic ordering: no single parameter is guaranteed never to increase, and thus no "most significant" parameter exists for a lexicographic ordering.

Although this example shows that bounded anchoring is in some theoretical sense stronger than techniques based on lexicographic ordering, it is doubtful whether this makes a difference for natural programs. What *can* be said is that the present approach is very *operational* and *automatic:* there is no need for a human to say under what ordering termination should be proved.

When using bounded anchoring for detecting lexicographic ordering, it is vital to use the strong form of bounded anchoring (cf. Conditions 6.7 and 6.8 on page 90), as there may not be a variable satisfying the weak form of bounded anchoring.

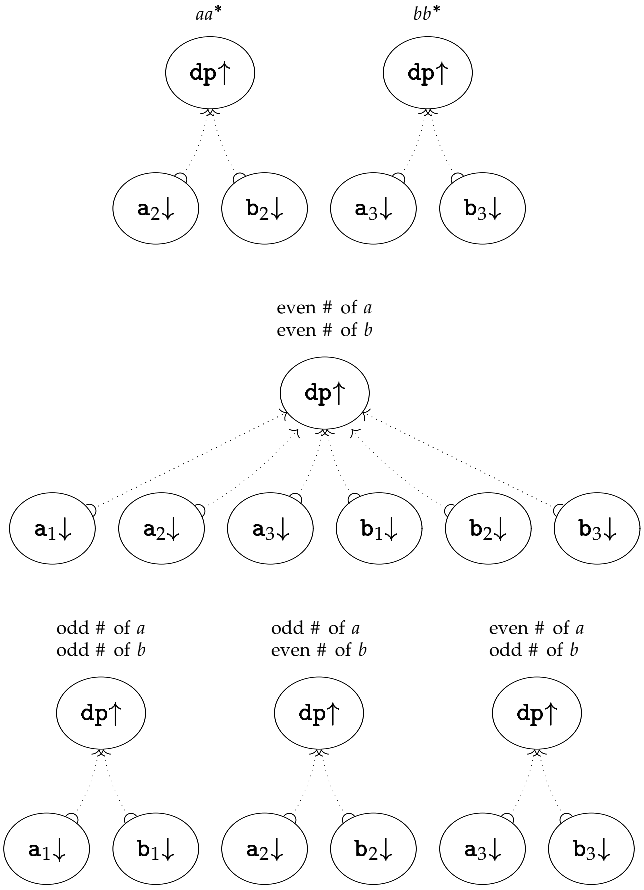


Figure 10.1: LDG for example 10.1

10.3 Future work

We have presented an algorithm for generalising variables for termination reasons, but this is not new: previously, heuristics have been used, e.g. *poor man's generalisation*, for approaching termination. It would be interesting to investigate their results on typical programs to see whether the added safety of our approach does not produce too conservative results, compared to poor man's generalisation.

Similarly, a common way of selecting specialisation point is to choose all dynamic conditionals, unfolding all the original functions. This does not prevent static-infinite computations, but again it would be interesting to perform some comparisons of the strengths of the two strategies on real-world programs.

As previously mentioned, the algorithm for finding a small set of specialisation points necessary for guaranteeing termination has exponential worst-case complexity, and this problem should be addressed in depth in future research.

There are also other directions in which investigations should go:

10.3.1 Extensible loop anchoring

Assuming the domain of the variables to be the nonnegative integers, the algorithm in this paper can detect that the usual definition of greatest common divisor function terminates. Now consider the following definition, where the parameters are swapped at each recursive call:

```
gcd dp x y =  if x = y then x else
              if x < y then gcd^a (dp + 1) (y - x) x else
                            gcd^b (dp + 1)  y (x - y)
```

It is clear that this `gcd` function will terminate on all input. Yet, when we compute the loop approximation and sibling information for `dp` we get:

$$\begin{aligned}
\{1 &: (\uparrow, \{\mathtt{x} \longrightarrow \mathtt{y}, \mathtt{y} \xrightarrow{\downarrow} \mathtt{x}\}), \quad a(ba)^* \\
2 &: (\uparrow, \{\mathtt{y} \longrightarrow \mathtt{x}, \mathtt{x} \xrightarrow{\downarrow} \mathtt{y}\}), \quad b(ab)^* \\
3 &: (\uparrow, \{\mathtt{x} \longrightarrow \mathtt{x}, \mathtt{y} \xrightarrow{\downarrow} \mathtt{y}\}), \quad ab(ab)^*
\end{aligned}$$

$$
\begin{array}{ll}
4 : (\uparrow, \{\mathtt{x} \xrightarrow{\downarrow} \mathtt{x}, \mathtt{y} \longrightarrow \mathtt{y}\}), & ba(ba)^* \\
5 : (\uparrow, \{\mathtt{x} \xrightarrow{\downarrow} \mathtt{x}, \mathtt{y} \xrightarrow{\downarrow} \mathtt{y}\}), & ((a|b)(a|b))^*(aa|bb)((a|b)(a|b))^* \\
6 : (\uparrow, \{\mathtt{x} \xrightarrow{\downarrow} \mathtt{y}, \mathtt{y} \xrightarrow{\downarrow} \mathtt{x}\})\} & ((a|b)(a|b))^*(a|b)(aa|bb)((a|b)(a|b))^*
\end{array}
$$

Here

Loop 1 represents an odd number of calls at site $a, b, a, b, a, \ldots$

Loop 2 represents an odd number of calls at site $b, a, b, a, b, \ldots$

Loop 3 represents an even number of calls at site $a, b, a, b, a, b, \ldots$

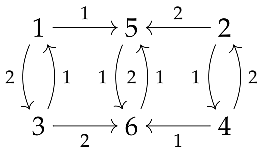
Loop 4 represents an even number of calls at site $b, a, b, a, b, a, \ldots$

Loop 5 represents an even number of calls greater than two, with two identical consecutive calls

Loop 6 represents an odd number of calls greater than two, with two identical consecutive calls.

Looking at this approximation set, we see that neither loop 1 nor loop 2 nor loop 6 is anchored in any sibling *loop*, so we would normally conclude that we cannot guarantee termination.

But let's look at how the individual loop approximations are computed; they are computed by extending existing loops with a 1 or 2 path according to this graph which is easily obtained as a by-product of computing the loop approximation:



This graph is interpretated thus: when trying to extend path 4 with path 1 or path 2, you either obtain 6 or 2, etc.

Now consider `dp` loops 3, 4 and 5: they are immediately anchored. For loops 1, 2 and 6, no matter which call you extend them with (1 or 2), you obtain a loop which already *is* anchored! Thus we could extend the present work by concluding that all **dp** loops are "extensibly" anchored, and so the program is guaranteed to terminate!

Note that this is a strict extension of Carsten Kehler's work (Holst, 1991), as that would not classify **gcd** as quasiterminating for static **dp**.

10.3.2 More precise size dependencies

The present algorithm is based on a rather crude approximation of the program values, using only ↑ and ↓ arrows, and is overly conservative in many cases, e.g. **cdr (cons x y)** which could be considered as {⇞(x)}, but is approximated to {↑(x),↑(y)}. Das (1998), Das and Reps (1996) have addressed this problem, using context-free language reachability, and an extension similar to this certainly seems necessary for treating any real-world examples. It could perhaps also be based on Hughes et. al's (1996) sized types.

Andersen and Holst (1996) also introduce a refinement such that the result of the dependency analysis for the two branches of **if**-expressions can "flow together again." Consider

```
f x y   = g x x y
g u v w = if w then cdr u else cdr v
```

Using a first order object to describe size dependencies, we get {↓(u)} and {↓(v)} for the **if**-branches. When computing g's return value size, the only safe description would be {}, because we must take the intersection of the branch dependencies for the result to be correct. This leads to the overly conservative approximation {} for f—the less conservative approximation {↓(x)} is in fact also correct. Andersen and Holst resolve this problem by introducing disjunctive size descriptions which allows describing **g** by (↓(u) ∨ ↓(v)), and subsequently ↓(x) for f.

In our presentation, we describe size dependency approximations by *second order* objects: given an expression *e*, $\mathcal{E}^{\mu}[\![e]\!]$ returns a *function* that takes an environment and finally returns the size dependency set, cf. Section 5.3. This feature allows us to obtain the same good results as Andersen and Holst obtain, because the intersection operation is postponed until the actual variable names (in the above case x and y) are known.

10.3.3 Other extensions

It is unclear whether the graph-based method presented in this paper is extensible to the higher order case, and what effects imperative constructs would have on the boundedness conditions. Also, some research has lately had success with systematic propagation of static values across dynamic contexts (Hatcliff & Danvy, 1996), and it would be interesting to see what effects those techniques would have on the present algorithm.

An obviously necessary extension for the analyses to be useful in practise would be to handle integers. This could be realised in a fairly straightforward manner by adding a domain analysis to find some bounds on the values that each variable can be assigned during evaluation. Extending the termination analysis to cope with pointers or imperative constructs that can introduce cyclic data structures and aliasing would require some advanced analysis (Fradet & Le Métayer, 1997; Ghiya & Hendren, 1996) to infer the "shapes" of the data structures (e.g. a list constructed using pointers), before they could be used as anchors.

10.4 Conclusion

In this paper we have shown how the key concept of *bounded variation* can be used to develop an analysis for automatically detecting program termination in a first-order functional language simply by adding an extra depth parameter to each function. We have developed analyses that safely detect variables of bounded variation using the central *bounded anchoring* condition, and we have proven (most parts of) them correct on a semantic basis.

We have shown that the techniques can easily be extended to ensure termination of off-line partial evaluation by looking for variables of *bounded static variation*. Depth parameters are of special interest here because they provide a safe approximation to a set of specialisation points that are necessary for termination.

We have presented algorithms that implement the analyses using reasonably efficient graph algorithms, obtaining a worst-case complexity of $O(p^3)$, and at most $O(p^2)$ for typical programs, where p is the program size. Experiments with a prototype implementation have shown that the analyses work well on several smaller programs, and especially interpreters are handled well. Due to a rather crude size approximation, we are unable to prove termination of some of the example programs, mainly sorting algorithms that perform a lot of destructing and constructing operations.

This paper thus contributes towards making off-line partial evaluators automatic and useful tools, even for users that know little about specialiser termination problems. But this is not all—also for on-line partial evaluation the techniques presented here can improve both the degree and speed of specialisation. Another important contribution is, we hope, a better understanding of the problems occurring in the attempt to ensure termination of partial evaluation.

References

Abelson, H., Dybvig, R. K., Haynes, C. T., Rozas, G. J., IV, N. I. A., Friedman, D. P., Kohlbecker, E., Steele Jr., G. L., Bartley, D. H., Halstead, R., Oxley, D., Sussman, G. J., Brooks, G., Hanson, C., Pitman, K. M., & Wand, M. (1998). Revised report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation, 11*(1), 7–105.

Aho, A. V., Hopcroft, J. E., & Ullman, J. D. (1975). *The design and analysis of computer algorithms.* Addison-Wesley.

Andersen, L. O. (1994). *Program analysis and specialization for the C programming language.* Unpublished doctoral dissertation, DIKU, University of Copenhagen, Denmark, Copenhagen, Denmark.

Andersen, P. H., & Holst, C. K. (1996). Termination analysis for offline partial evaluation of a higher order functional language. In *Proceedings of the third international static analysis symposium (SAS).*

Arts, T., & Giesl, J. (1997). Automatically proving termination where simplification orderings fail. In *Proceedings of the 7th international joint conference on the theory and practice of software development (TAPSOFT'97)* (Vol. 1214). Lille, France: Springer-Verlag.

Birkedal, L., & Welinder, M. (1993). *Partial evaluation of standard ML.* Unpublished master's thesis, DIKU, University of Copenhagen, Denmark, Copenhagen, Denmark.

Bondorf, A. (1990). *Self-applicable partial evaluation.* Unpublished doctoral dissertation, DIKU, University of Copenhagen, Denmark, Copenhagen, Denmark.

Bondorf, A. (1993, May). *Similix 5.0 Manual.* (Included in Similix distribution, ftp://ftp.diku.dk/pub/diku/semantics/similix/Similix.tar.gz)

Bondorf, A., & Jørgensen, J. (1993). *Efficient analysis for realistic off-line partial evaluation: Extended version* (Tech. Rep. No. 93/4). Copenhagen, Denmark: DIKU, University of Copenhagen, Denmark.

Boyer, R. S., & Moore, J. S. (1979). *A computational logic.* New York: Academic Press.

Brauburger, J. (1997). Automatic termination analysis for partial functions using polynomial orderings. In *Static analysis symposium* (Vol. 1302, pp. 330–344). Springer-Verlag.

Codish, M., & Taboch, C. (1997). A semantic basis for termination analysis of logic programs and its realization using symbolic norm constraints. In *Proceedings of the sixth international conference on algebraic and logic programming.*

Consel, C. (1993). A tour of Schism: a partial evaluation system for higher-order applicative languages. In ACM (Ed.), *Proceedings of the ACM SIGPLAN symposium on partial evaluation and semantics-based program manipulation. PEPM'93* (pp. 145–154). New York, NY, USA: ACM Press.

Consel, C., Hornof, L., Noel, F., & Noye, J. (1996). A uniform approach for compile-time and run-time specialization. *Lecture Notes in Computer Science*, *1110*, 54–72.

Cormen, T. H., Leiserson, C. E., & Rivest, R. L. (1990). *Introduction to algorithms.* Cambridge, Massachusetts, USA: MIT Press.

Danvy, O., Malmkjær, K., & Palsberg, J. (1995). The essence of eta-expansion in partial evaluation. *LISP and Symbolic Computation*, *8*(3), 209–227.

Das, M. (1998). *Partial evaluation using dependence graphs.* Unpublished doctoral dissertation, University of Wisconsin-Madison.

Das, M., & Reps, T. (1996). *BTA termination using CFL-reachability* (Tech. Rep. No. 1329). Computer Science Department, University of Wisconsin-Madison.

Dershowitz, N. (1987). Termination of rewriting. In J.-P. Jouannaud (Ed.), *Rewriting techniques and applications* (pp. 69–115). Academic Press. (Reprinted from *Journal of Symbolic Computation*)

Diestel, R. (1997). *Graph theory.* New York: Springer-Verlag. (Translation of *Graphentheorie*, Springer-Verlag, 1996)

Fradet, P., & Le Métayer, D. (1997). Shape types. In *ACM symposium on principles of programming languages* (pp. 27–39). Paris, France.

Ghiya, R., & Hendren, L. J. (1996). Is it a tree, a DAG, or a cyclic graph? a shape analysis for heap-directed pointers in C. In *ACM symposium on principles of programming languages* (pp. 1–15). Florida.

Giesl, J. (1995). Termination analysis for functional programs using term orderings. In *Proceedings of the second international static analysis symposium (SAS'95)* (Vol. 983). Glasgow, Scotland: Springer-Verlag.

Glenstrup, A. J., & Jones, N. D. (1996). BTA algorithms to ensure termination of off-line partial evaluation. In *Perspectives of system informatics: Proceedings of the Andrei Ershov second international memorial conference* (Vol. 1181, pp. 273–284). Springer-Verlag.

Hatcliff, J., & Danvy, O. (1996). A computational formalization for partial evaluation. *Mathematical Structures in Computer Science, special issue.*

Henglein, F., & Tofte, M. (1993). An introduction to operational semantics of programming languages. In *Datalogi 2.1 kursusbog* (Vol. 1, pp. 7–109). Copenhagen, Denmark: *DIKU*tryk*. (Based on lecture notes by Robin Milner)

Holst, C. K. (1991). Finiteness analysis. In J. Hughes (Ed.), *Functional programming languages and computer architectures* (pp. 473–495). Cambridge, Massachusetts, USA: Springer-Verlag.

Hughes, J., Pareto, L., & Sabry, A. (1996). Proving the correctness of reactive systems using sized types. In *ACM symposium on principles of programming languages* (pp. 410–423). St. PetersBurg FLA USA.

Jones, N. D. (1988). Automatic program specialization: A re-examination from basic principles. In D. Bjørner, A. P. Ershov, & N. D. Jones (Eds.), *Partial evaluation and mixed computation* (pp. 225–282). Amsterdam: Elsevier.

Jones, N. D. (1996). What *Not* to do when writing an interpreter for specialisation. In O. Danvy, R. Glück, & P. Thiemann (Eds.), *Partial evaluation* (Vol. 1110, pp. 216–237). Springer-Verlag. (International Seminar at Dagstuhl Castle, Germany)

Jones, N. D., Gomard, C. K., & Sestoft, P. (1993). *Partial evaluation and automatic program generation.* Prentice-Hall.

König, D. (1936). *Theorie der endlichen und unendlichen graphen.* Leipzig: Academische Verlagsgesellschaft.

Lafave, L., & Gallagher, J. (1997). *Partial evaluation of functional logic programs in rewriting-based languages* (Tech. Rep. Nos. CSTR–97–001). Bristol, UK: Department of Computer Science, University of Bristol.

Leuschel, M. (1998). *Homeomorphic embedding for online termination* (Tech. Rep. Nos. DSSE–TR–98–11). UK: Department of Electronics and Computer Science, University of Southampton.

Lindenstrauss, N., & Sagiv, Y. (1996, October). *Checking termination of queries to logic programs.* Jerusalem, Israel. (http://www.cs.huji.ac.il/~naomil/)

Lindenstrauss, N., & Sagiv, Y. (1997). *Automatic termination analysis of logic programs (with detailed experimental results).* Jerusalem, Israel. (http://www.cs.huji.ac.il/~naomil/)

Nielsen, K. (1993). *Note on context specialisation.* (Unpublished, DIKU)

Nielson, F., & Nielson, H. R. (1992). *Two-level functional languages.* Cambridge University Press.

Nielson, F., & Nielson, H. R. (1996). Operational semantics of termination types. *Nordic Journal of Computing*, *3*, 144–187.

Ramalingam, G., & Reps, T. (1989). *Semantics of program representation graphs* (Tech. Rep. Nos. TR–900). University of Wisconsin-Madison.

Somogyi, Z., Henderson, F., & Conway, T. (1995). Mercury: an efficient purely declarative logic programming language. In *Proceedings of the australian computer science conference* (p. 499-512). Glenelg, Australia.

Sørensen, M. H., & Glück, R. (1995). An algorithm of generalization in positive supercompilation. In J. Lloyd (Ed.), *Logic programming: Proceedings of the 1995 international symposium* (pp. 465–479). MIT Press.

Speirs, C. (1997). *Termination analysis for logic programs* (Tech. Rep. No. 97/23). Department of Computer Science, University of Melbourne.

Speirs, C., Somogyi, Z., & Søndergaard, H. (1997). *Termination analysis for mercury* (Tech. Rep. No. 97/9). Melbourne, Australia: Department of Computer Science, University of Melbourne.

Steinbach, J. (1995). Simplification orderings: History of results. *Fundamental Informaticae*(24), 47–87.

Thibault, S., & Consel, C. (1997). A framework for application generator design. *ACM SIGSOFT Software Engineering Notes*, *22*(3), 131–135.

Thibault, S., Marlet, R., & Consel, C. (1997). A domain specific language for video device drivers: From design to implementation. In *Proceedings of the conference on domain-specific languages (DSL-97)* (pp. 11–26). Berkeley: USENIX Association.

Thiemann, P. (1998). Aspects of the PGG system: Specialization for standard scheme. In *DIKU international summer school* (Vol. 2, pp. 109–129). Copenhagen, Denmark: DIKU, University of Copenhagen, Denmark.

Thiemann, P. J., & Dussart, D. (1997). Combinator-based program generation. In *ACM symposium on principles of programming languages.* Amsterdam, Holland.

Walther, C. (1988). Argument-bounded algorithms as a basis for automated termination proofs. In *9th international conference on automated deduction* (Vol. 310, pp. 602–621). Argonne, Illinois, USA: Springer-Verlag.

Winston, P. H. (1984). *Artificial intelligence.* Addison-Wesley.

Index

Appendix A

Partial Evaluation Semantics

In this section we give the syntax and semantics of partial evaluation as used in the present paper.

A.1 Two-level syntax

Off-line partial evaluation divides the transformation into two stages: first each expression of the source program is annotated as either "reduce statically" or "defer to run-time," and then specialisation computes the static parts and generates code for the dynamic (run-time) parts. We represent the annotations as is common by a *two-level syntax:*

$$\begin{array}{rccl}
\mathit{AProgram} & \ni & p & ::= \; \mathit{f1}\ x_1^{\beta_{11}} \ldots x_m^{\beta_{1m}} = e_1^{\beta_1}; \ldots ; \mathit{fn}\ x_1^{\beta_{n1}} \ldots x_k^{\beta_{nk}} = e_n^{\beta_n} \\
\mathit{AExpression} & \ni & e & ::= \; x \mid c \mid b^{\beta}\ e_1 \ldots e_n \mid f^{\beta}\ e_1 \ldots e_n \mid \mathbf{lift}^{D}\ e^{S} \\
 & & & \quad\;\; \mid \; \mathbf{if}^{\beta}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 \\
 & & \beta & \in \; \{S, D\}
\end{array}$$

Not all two-level program are sensible; we require as usual that two-level programs be congruent (Jones et al., 1993), i.e. that no expression marked as static (S) depends on the result of an expression marked as dynamic (D).

In this paper we furthermore insist that if the condition of an **if**-expression is dynamic, then the whole expression is. This may seem rather conservative, given the present-day techniques of context-specialisation (Nielsen, 1993; Thiemann & Dussart, 1997; Danvy, Malmkjær, & Palsberg, 1995; Hatcliff & Danvy, 1996), but for simplicity we leave this extension to future work.

A.2 Partial evaluation semantics

We adopt the convention that variables named s and d are intended to be static and dynamic, respectively, and wlog. we assume that all static parameters of a function precede the dynamic ones. As we are now producing not only values but also code, we use <u>underlining</u> to indicate the generation of syntax (Nielson & Nielson, 1992).

We can now define partial evaluation for this two-level language by the partial evaluation operators $\mathcal{PP}$, $\mathcal{PE}$, $\mathcal{PA}$ and $\mathcal{PC}$ that use an extension of the *Value* and *Context* domains:

$$\begin{array}{rcl}
\mathit{PEValue} & = & \mathit{Value} \cup \mathit{Expression} \cup \{\textbf{lift}\ v\} \ni pv \\
\mathit{AContext} & \ni & ve ::= b\ pv_1 \ldots pv_m \bullet e_1 \ldots e_n \mid f\ pv_1 \ldots pv_m \bullet e_1 \ldots e_n \\
 & & \quad \mid \textbf{if}^{\beta} \bullet \textbf{then}\ e_2\ \textbf{else}\ e_3 \mid \textbf{if}^{D} pv_1\ \textbf{then} \bullet \textbf{else}\ e_3 \\
 & & \quad \mid \textbf{if}^{D} pv_1\ \textbf{then}\ pv_2\ \textbf{else} \bullet \mid \textbf{lift} \bullet
\end{array}$$

The definition of the operators is given in Figure A.1; $\mathcal{PE}$ is identical to $\mathcal{E}$ except for the **lift** case, whereas $\mathcal{PA}$ must be extended as we are generating code for both branches of the dynamic **if**-expressions.

For any interesting program the unfolding defined by these operators will not terminate because partial evaluation is *over-strict:* it evaluates both branches of the dynamic conditional (cf. p_{3d}-rules) and unfolds all function calls. To alleviate this problem the two-level syntax is extended with function calls that are marked with a $\star$ as *specialisation points*:

$$\mathit{AExpression} ::= \ldots \mid f^{\star}\ s_1 \ldots s_m\ d_1 \ldots d_n$$

We now introduce a folding scheme whereby we maintain a list of which specialisation points (function names) with which static arguments we have

$$
\begin{array}{lcl}
\mathcal{PP} & : & \mathit{AProgram} \rightarrow \mathit{Value\ List} \rightarrow \mathit{PEValue} \\
\mathcal{PA} & : & \mathit{Stack} \rightarrow \mathit{PEValue} \rightarrow \mathit{PEValue} \\
\mathcal{PC} & : & \mathit{Funname} \rightarrow \mathit{PEValue\ List} \rightarrow \mathit{Stack} \rightarrow \mathit{PEValue} \\
\mathcal{PE} & : & \mathit{AExpression} \rightarrow \mathit{Environment} \rightarrow \mathit{Stack} \rightarrow \mathit{PEValue}
\end{array}
$$

$$
\begin{array}{l}
\mathcal{PP}\ \llbracket f1\ s_1 \ldots s_n\ d_1 \ldots d_m = e_1; \ldots \rrbracket\ [v_1, \ldots, v_n] \\
\rightarrow \mathcal{PC}\ \llbracket f1 \rrbracket\ [v_1, \ldots, v_n, \underline{f1_{n+1}}, \ldots, \underline{f1_{n+m}}]\ [\,]
\end{array}
$$

$$
\begin{array}{ll}
(p_1)\,\mathcal{PE}\ \llbracket c \rrbracket\ \varrho\ s & \rightarrow \mathcal{PA}\ s\ (\mathit{value}\ c) \\
(p_2)\,\mathcal{PE}\ \llbracket x \rrbracket\ \varrho\ s & \rightarrow \mathcal{PA}\ s\ (\varrho\ x) \\
(p_3)\,\mathcal{PE}\ \llbracket \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 \rrbracket\ \varrho\ s & \rightarrow \mathcal{PE}\ \llbracket e_1 \rrbracket\ \varrho\ (\langle \mathbf{if}\ \bullet\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3, \varrho \rangle : s) \\
(p_4)\,\mathcal{PE}\ \llbracket b\ e_1\ e_2 \ldots e_n \rrbracket\ \varrho\ s & \rightarrow \mathcal{PE}\ \llbracket e_1 \rrbracket\ \varrho\ (\langle b\ \bullet\ e_2 \ldots e_n, \varrho \rangle : s) \\
(p_5)\,\mathcal{PE}\ \llbracket f\ e_1\ e_2 \ldots e_n \rrbracket\ \varrho\ s & \rightarrow \mathcal{PE}\ \llbracket e_1 \rrbracket\ \varrho\ (\langle f\ \bullet\ e_2 \ldots e_n, \varrho \rangle : s) \\
(p_6)\,\mathcal{PE}\ \llbracket \mathbf{lift}\, e \rrbracket\ \varrho\ s & \rightarrow \mathcal{PE}\ \llbracket e \rrbracket\ \varrho\ (\langle \mathbf{lift}\ \bullet, \varrho \rangle : s)
\end{array}
$$

$$
\begin{array}{l}
(p_1')\ \ \mathcal{PA}\ [\,]\ pv \rightarrow pv \\
(p_{3s})\,\mathcal{PA}\ (\langle \mathbf{if}^S\ \bullet\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3, \varrho \rangle : s)\ pv \\
\qquad \rightarrow \text{if } pv \text{ then } \mathcal{PE}\ \llbracket e_2 \rrbracket\ \varrho\ s \text{ else } \mathcal{PE}\ \llbracket e_3 \rrbracket\ \varrho\ s \\
(p_{3d})\,\mathcal{PA}\ (\langle \mathbf{if}^D\ \bullet\ \mathbf{then}\ e_2^D\ \mathbf{else}\ e_3^D, \varrho \rangle : s)\ pv \\
\qquad \rightarrow \mathcal{PE}\ \llbracket e_2^D \rrbracket\ \varrho\ (\langle \mathbf{if}^D\ pv\ \mathbf{then}\ \bullet\ \mathbf{else}\ e_3^D, \varrho \rangle : s) \\
(p_{3d}')\,\mathcal{PA}\ (\langle \mathbf{if}^D\ pv_1\ \mathbf{then}\ \bullet\ \mathbf{else}\ e_3^D, \varrho \rangle : s)\ pv \\
\qquad \rightarrow \mathcal{PE}\ \llbracket e_3^D \rrbracket\ \varrho\ (\langle \mathbf{if}^D\ pv_1\ \mathbf{then}\ pv\ \mathbf{else}\ \bullet, \varrho \rangle : s) \\
(p_{3d}'')\,\mathcal{PA}\ (\langle \mathbf{if}^D\ pv_1\ \mathbf{then}\ pv_2\ \mathbf{else}\ \bullet, \varrho \rangle : s)\ pv \\
\qquad \rightarrow \mathcal{PA}\ s\ \underline{\mathbf{if}^D\ pv_1\ \mathbf{then}\ pv_2\ \mathbf{else}\ pv} \\
(p_4')\ \ \mathcal{PA}\ (\langle b\ pv_1 \ldots pv_m \bullet e_2\ e_3 \ldots e_n, \varrho \rangle : s)\ pv \\
\qquad \rightarrow \mathcal{PE}\ \llbracket e_2 \rrbracket\ \varrho\ (\langle b\ pv_1 \ldots pv_m\ pv \bullet e_3 \ldots e_n, \varrho \rangle : s) \text{ for } n > 1 \\
(p_{4s})\,\mathcal{PA}\ (\langle b^S\ pv_1 \ldots pv_m\ \bullet, \varrho \rangle : s)\ pv \rightarrow \mathcal{PA}\ s\ (\mathit{apply}\ b\ [pv_1, \ldots, pv_m, pv]) \\
(p_{4d})\,\mathcal{PA}\ (\langle b^D\ pv_1 \ldots pv_m\ \bullet, \varrho \rangle : s)\ pv \rightarrow \mathcal{PA}\ s\ (\underline{b\ pv_1 \ldots pv_m\ pv}) \\
(p_5')\ \ \mathcal{PA}\ (\langle f\ pv_1 \ldots pv_m \bullet e_2\ e_3 \ldots e_n, \varrho \rangle : s)\ pv \\
\qquad \rightarrow \mathcal{PE}\ \llbracket e_2 \rrbracket\ \varrho\ (\langle f\ pv_1 \ldots pv_m\ pv \bullet e_3 \ldots e_n, \varrho \rangle : s) \text{ for } n > 1 \\
(p_5'')\ \ \mathcal{PA}\ (\langle f\ pv_1 \ldots pv_m\ \bullet, \varrho \rangle : s)\ pv \rightarrow \mathcal{PC}\ \llbracket f \rrbracket\ [pv_1, \ldots, pv_m, pv]\ s \\
(p_5''')\,\mathcal{PC}\ \llbracket f \rrbracket\ [pv_1, \ldots, pv_n]\ s \rightarrow \mathcal{PE}\ \llbracket e^f \rrbracket\ \{f_1 \mapsto pv_1, \ldots, f_n \mapsto pv_n\}\ s \\
(p_6')\ \ \mathcal{PA}\ (\langle \mathbf{lift}\ \bullet, \varrho \rangle : s)\ v \rightarrow \mathcal{PA}\ s\ \underline{v}
\end{array}
$$

Figure A.1: Rewrite rules defining the small-step semantics for partial evaluation

encountered during the transformation. Whenever we reach a specialisation point, we look up the function and its static arguments in the list; if they are not found, we *make a new function definition* with a fresh name and all the dynamic parameters of the specialisation point. We then add the (specialisation point, arguments, new function name)-triple to the list and continue specialising the body of the function. If, on the other hand, we *do* find the function and its static arguments in the list, we immediately return a dynamic value: a piece of code to call the corresponding function definition.

This can all be expressed by passing around[1] a function for looking up functions and static values that have already been seen φ : $\mathit{Funname} \times \mathit{ValueList} \rightarrow \mathit{Funname}$, which in the initial call from $\mathcal{PP}$ is the overall undefined function, and adding the following two fold and memoise rules:

$$
\begin{array}{ll}
(p_{5f}) & \mathcal{PC}\ [\![f^{\star}]\!]\ \varphi\ [s_1,\ldots,s_n,d_1,\ldots,d_m]\ s \\
& \rightarrow \begin{cases} \mathcal{PA}\ \varphi\ s\ \underline{g_{\varphi}\ d_1\ldots d_m}, \text{if } (f,[s_1,\ldots,s_n]) \in \mathit{dom}\ \varphi \\ \mathcal{PE}\ [\![e^f]\!]\ \varrho\ (\varphi + \{(f,[s_1,\ldots,s_n]) \mapsto g\})\ s', \text{otherwise} \end{cases} \\
& \quad \text{where } \begin{array}{lcl} g_{\varphi} & = & \varphi\ (f,[s_1,\ldots,s_n]) \\ \varrho & = & \{f_1 \mapsto s_1,\ldots,f_n \mapsto s_n, \\ & & \ f_{n+1} \mapsto \underline{f_{n+1}},\ldots,f_{n+m} \mapsto \underline{f_{n+m}}\} \\ s' & = & \langle g\ d_1\ldots d_m = \bullet\ f_{n+1}\ldots f_{n+m}, \varrho\rangle : s \\ g & = & \mathit{freshname}() \notin \mathit{rg}\ \varphi \end{array} \\
(p_{5m}) & \mathcal{PA}\ \varphi\ (\langle g\ d_1\ldots d_m = \bullet\ f_{n+1}\ldots f_{n+m}, \varrho\rangle : s)\ v \\
& \rightarrow \mathcal{PA}\ \varphi\ s\ (\underline{g\ d_1\ldots d_m,}\ \textbf{where}\ g\ f_{n+1}\ \ldots f_{n+m} = v)
\end{array}
$$

The **where** clause in rule (p_{6m}) is to be interpreted as "output a definition of function g to the residual program and let '$g\ d_1\ldots d_m$' be the return value of $\mathcal{PA}$." To ease the notation we will normally assume the φ is present without explicitly writing it.

The choice of where to put the specialisation points affects the termination and residual code size properties of the transformation, and we defer the discussion of this topic till Section 7.4.

Although we have expanded the evaluation of **if**-expressions by several

[1] This can of course also be implemented as a global list or function

steps, this semantics still treats all recursion using a stack with the operators being defined in a tail recursive way. Therefore we still have

Lemma A.1
For the two-level semantics defined in Figure A.1 linear compression as defined in Corollary 3.5 holds.

Proof: To show Corollary 3.5 for the partial evaluation operators we only need to show that the call-free evaluation depth is still bounded (Lemma 3.4). But this is easily checked using the depth function

$$|e| = \begin{cases} 3, & \text{if } e \equiv x \text{ or } e \equiv c \\ 1+n+|e_1|+\cdots+|e_n|, & \text{if } e \equiv b\ e_1 \ldots e_n \text{ or } e \equiv f\ e_1 \ldots e_n \\ n+|e_1|+\cdots+|e_n|, & \text{if } e \equiv b\ pv_1 \ldots pv_m \bullet e_1 \ldots e_n \\ n+|e_1|+\cdots+|e_n|, & \text{if } e \equiv f\ pv_1 \ldots pv_m \bullet e_1 \ldots e_n \\ 0, & \text{if } e \equiv g\ d_1 \ldots d_m = \bullet\ f_{n+1} \ldots f_{n+m} \\ 4+|e_1|+|e_2|+|e_3|, & \text{if } e \equiv \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 \\ 2+|e_2|+|e_3|, & \text{if } e \equiv \textbf{if } \bullet \textbf{ then } e_2 \textbf{ else } e_3 \\ 1+|e_3|, & \text{if } e \equiv \textbf{if } pv_1 \textbf{ then } \bullet \textbf{ else } e_3 \\ 0, & \text{if } e \equiv \textbf{if } pv_1 \textbf{ then } pv_2 \textbf{ else } \bullet \\ 0, & \text{if } e \equiv \textbf{lift } \bullet \\ 2+|e_1|, & \text{if } e \equiv \textbf{lift } e_1 \end{cases}$$

$$|s| = \begin{cases} 0, & \text{if } s \equiv [\,] \\ 1+|e|+|s'|, & \text{if } s \equiv \langle e, \varrho \rangle : s' \end{cases}$$

□

We must of course justify our claim that this semantics defines partial evaluation: it does so because it satisfies the mix equation (Jones et al., 1993). To show this, we must be able to identify subsequences that start with an expression and end with the value of this expression, without using objects from the initial stack.

For any stacks s_1, s_2 we therefore say that s_2 is a *suffix* of s_1 iff there exist elements $\xi_1, \ldots, \xi_n$ such that $s_1 = \xi_1 : \cdots : \xi_n : s_2$. If $n > 0$ we say s_2 is a *real suffix* of s_1. For a sequence $\mathcal{PE}\ [\![e_1]\!]\ \varrho_1\ s \rightarrow \cdots \rightarrow \mathcal{PA}\ s\ v$ we say it is *independent* of s if s is a real suffix of all stacks occurring as operands to intermediate $\mathcal{PE}$ and $\mathcal{PA}$ operators in the sequence. Note that such independent sequences are exactly the subsequences we are looking for. The stack *can* in fact be a non-real suffix along the sequence, but only as an argument to operator $\mathcal{PC}$, and this poses no problem, as the stack of the immediately following operator is uniquely determined by the stack and operator preceding $\mathcal{PC}$.

We then define what we require of the lookup function φ for it to be correct with respect to partial evaluation: We say φ is *faithful* iff

$$\begin{aligned}\forall (f, [s_1, \ldots, s_n]) \in dom\ \varphi\ \exists v \in Value : \mathcal{E}\ [\![e^f]\!]\ \varrho_f\ s \to \cdots \to \mathcal{A}\ s\ v \\ \wedge\, \mathcal{E}\ [\![e^g]\!]\ \varrho_g\ s^D \to \cdots \to \mathcal{A}\ s^D\ v,\end{aligned}$$

where $g = \varphi\ (f, [s_1, \ldots, s_n])$, $\varrho_f = \{f_1 \mapsto s_1, \ldots f_n \mapsto s_n, f_{n+1} \mapsto d_1, \ldots, f_{n+m} \mapsto d_m\}$ and $\varrho_g = \{g_1 \mapsto d_1, \ldots, g_m \mapsto d_m\}$, and the sequences are independent of s and s^D.

Now we are able to show that partial evaluation of expressions preserves the faithfulness of φ and that evaluating the resulting partial value on the dynamic input yields the same result as evaluating the original expression on both the static and dynamic input:

Lemma A.2 (Mix equation for Expressions)
For any stack s' and any well-formed expression e, assume that e' is a well-annotated version of e and that $\{s_1, \ldots, s_n\} \cup \{d_1, \ldots, d_m\} = fv\ e$ are classified as static and dynamic, respectively. For $v_1, \ldots, v_{n+m} \in Value$, let $\varrho' = \{s_1 \mapsto v_1, \ldots, s_n \mapsto v_n, d_1 \mapsto \underline{d_1}, \ldots, d_m \mapsto \underline{d_m}\}$. Now if φ_1 is faithful and

$$\mathcal{PE}\ [\![e']\!]\ \varrho'\ \varphi_1\ s' \to \cdots \to \mathcal{PA}\ \varphi_2\ s'\ pv, \qquad (*)$$

is independent of s', then φ_2 is faithful and for any stacks s, s^D there exists a v such that

$$\begin{array}{lcl} \multicolumn{3}{l}{\mathcal{E}\ [\![e]\!]\ \varrho\ s \to \cdots \to \mathcal{A}\ s\ v \ \text{ and }\ \mathcal{E}\ [\![pv]\!]\ \varrho^D s^D \to \cdots \to \mathcal{A}\ s^D v,} \\ \text{where}\ \ \varrho & = & \{s_1 \mapsto v_1, \ldots, s_n \mapsto v_n, d_1 \mapsto v_{n+1}, \ldots, d_m \mapsto v_{n+m}\} \\ \text{and}\ \ \varrho^D & = & \{d_1 \mapsto v_{n+1}, \ldots, d_m \mapsto v_{n+m}\} \end{array}$$

Proof: By induction on the length og the step sequence, checking cases (p_1)–(p_6). Cases (p_1) and (p_2) are obvious, and here we only show the details of the most intricate dynamic cases: (p_3) and (p_5)—the remaining cases are quite similar.

Case (p_3)**:** $e' \equiv \mathbf{if}^D\, e'_1$ **then** e'_2 **else** e'_3**.** To show this, write e as $e \equiv$ **if** e_1 **then** e_2 **else** e_3 and define

$$\begin{array}{rcl}
s'_1 & = & \langle \mathbf{if}\ \bullet\ \mathbf{then}\ e'_2\ \mathbf{else}\ e'_3, \varrho'\rangle : s' \\
s'_2 & = & \langle \mathbf{if}\ pv_1\ \mathbf{then}\ \bullet\ \mathbf{else}\ e'_3, \varrho'\rangle : s' \\
s'_3 & = & \langle \mathbf{if}\ pv_1\ \mathbf{then}\ pv_2\ \mathbf{else}\ \bullet, \varrho'\rangle : s' \\
s_1 & = & \langle \mathbf{if}\ \bullet\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3, \varrho\rangle : s \\
s_1^D & = & \langle \mathbf{if}\ \bullet\ \mathbf{then}\ pv_2\ \mathbf{else}\ pv_3, \varrho^D\rangle : s^D \\
pv & = & \mathbf{if}\ pv_1\ \mathbf{then}\ pv_2\ \mathbf{else}\ pv_3\,.
\end{array}$$

Consider the sequence $(*)$; it can be split into three subsequences:

$$\begin{array}{rll}
\mathcal{PE}\ [\![e']\!]\ \varrho'\ s' \rightarrow \mathcal{PE}\ [\![e'_1]\!]\ \varrho'\ s'_1 \rightarrow \cdots \rightarrow \mathcal{PA}\ s'_1\ pv_1 & & (\sigma_1) \\
\rightarrow \mathcal{PE}\ [\![e'_2]\!]\ \varrho'\ s'_2 \rightarrow \cdots \rightarrow \mathcal{PA}\ s'_2\ pv_2 & & (\sigma_2) \\
\rightarrow \mathcal{PE}\ [\![e'_3]\!]\ \varrho'\ s'_3 \rightarrow \cdots \rightarrow \mathcal{PA}\ s'_3\ pv_3 & & (\sigma_3) \\
\rightarrow \mathcal{PA}\ s'\ pv, & &
\end{array}$$

where s'_1, s'_2, s'_3 are real suffixes of all the intermediate stacks in the respective subsequences.

By induction on σ_1, we have $\mathcal{E}\ [\![e_1]\!]\ \varrho\, s_1 \rightarrow \cdots \rightarrow \mathcal{A}\ s_1\ v_1$. We now consider the case where $v_1 = \mathsf{true}$; the opposite case is analogous. Then we find that φ_2 is faithful and

$$\begin{array}{lcl}
\mathcal{E}\ [\![e]\!]\ \varrho\, s & & \mathcal{E}\ [\![pv]\!]\ \varrho^D\, s^D \\
\rightarrow \mathcal{E}\ [\![e_1]\!]\ \varrho\, s_1 \rightarrow^* \mathcal{A}\ s_1\ v_1 & \text{(induction, } \sigma_1\text{)} & \rightarrow \mathcal{E}\ [\![pv_1]\!]\ \varrho^D\ s_1^D \rightarrow^* \mathcal{A}\ s_1^D\, v_1 \\
\rightarrow \mathcal{E}\ [\![e_2]\!]\ \varrho\, s & (v_1 = \mathsf{true}) & \rightarrow \mathcal{E}\ [\![pv_2]\!]\ \varrho^D\, s^D \\
\rightarrow \cdots \rightarrow \mathcal{A}\ s\ v_2 & \text{(induction, } \sigma_2\text{)} & \rightarrow \cdots \rightarrow \mathcal{A}\ s^D\, v_2
\end{array}$$

Case (p_5)**:** $e' \equiv f^\star\ e'_1 \ldots e'_{n+m}$. We write e as $e \equiv f\ e_1 \ldots e_{n+m}$ and define

$$\begin{array}{rcl}
s'_1 & = & \langle f^\star\ \bullet\ e'_2 \ldots e'_n, \varrho'\rangle : s' \\
 & \vdots & \\
s'_{n+m} & = & \langle f^\star\ pv_1\ pv_2 \ldots e'_{n+m-1}\ \bullet, \varrho'\rangle : s' \\
s_1 & = & \langle f\ \bullet\ e_2 \ldots e_{n+m}, \varrho\rangle : s \\
 & \vdots & \\
s_{n+m} & = & \langle f\ pv_1 \ldots pv_{n+m-1}\ \bullet, \varrho\rangle : s
\end{array}$$

$$\begin{array}{lcl}
s_f & = & \langle g\ pv_{n+1} \ldots pv_{n+m} = \bullet\ f_{n+1} \ldots f_{n+m}, \varrho'_f \rangle : s' \\
s^D_{n+1} & = & \langle g\ \bullet\ pv_{n+2} \ldots pv_{n+m}, \varrho^D \rangle : s^D \\
s^D_{n+m} & = & \langle g\ pv_{n+1} \ldots pv_{n+m-1}\ \bullet, \varrho^D \rangle : s^D
\end{array}$$

$$\begin{array}{lcl}
pv & = & g\ pv_{n+1} \ldots pv_{n+m}, \textbf{ where } g\ f_{n+1} \ldots f_{n+m} = pv^f \\
\varrho'_f & = & \{f_1 \mapsto pv_1, \ldots, f_n \mapsto pv_n, \\
& & \ f_{m+1} \mapsto \underline{f_{m+1}}, \ldots, f_{n+m} \mapsto \underline{f_{n+m}}\} \\
\varrho_f & = & \{f_1 \mapsto v_1, \ldots, f_{n+m} \mapsto v_{n+m}\} \\
\varrho_g & = & \{f_{n+1} \mapsto v_{n+1}, \ldots, f_{n+m} \mapsto v_{n+m}\}.
\end{array}$$

Consider the case where $(f, [pv_1, \ldots, pv_n]) \notin dom\ \varphi$. We can now split $(*)$ into $n+m+1$ subsequences:

$$\begin{array}{ll}
\mathcal{PE}\ [\![e']\!]\ \varrho'\ \varphi_1\ s' & \\
\rightarrow \mathcal{PE}\ [\![e'_1]\!]\ \varrho'\ \varphi_1\ s'_1 \rightarrow^* \mathcal{PA}\ \varphi_2\ s'_1\ pv_1 & (\sigma_1) \\
\vdots & \vdots \\
\rightarrow \mathcal{PE}\ [\![e'_{n+m}]\!]\ \varrho'\ \varphi_{n+m}\ s'_{n+m} \rightarrow^* \mathcal{PA}\ \varphi_{n+m+1}\ s'_{n+m}\ pv_{n+m} & (\sigma_{n+m}) \\
\rightarrow \mathcal{PC}\ [\![f^\star]\!]\ \varphi_{n+m+1}\ [pv_1, \ldots, pv_{n+m}]\ s' & \\
\rightarrow \mathcal{PE}\ [\![e^f]\!]\ \varrho'_f\ \varphi_f\ s_f \rightarrow^* \mathcal{PA}\ \varphi\ s_f\ pv^f \rightarrow \mathcal{PA}\ \varphi\ s'\ pv & (\sigma_f)
\end{array}$$

where again $s'_1, \ldots, s'_{n+m}$ and s_f are real suffixes of all the intermediate stacks in the respective subsequences. By induction $\varphi_i, i = 1, \ldots, n+m+1$ are faithful and as g is correctly defined in the final step, $\varphi_f = \varphi_{n+m+1} + \{(f, [pv_1, \ldots, pv_n]) \mapsto g\}$ and thus φ is faithful.

We now find that

$$\begin{array}{lll}
\mathcal{E}\ \llbracket e \rrbracket\ \varrho\ s & & \\
\quad \to \mathcal{E}\ \llbracket e_1 \rrbracket\ \varrho\ s_1 \to^* \mathcal{A}\ s_1\ v_1 & & \\
\quad \vdots & & \\
\quad \to \mathcal{E}\ \llbracket e_n \rrbracket\ \varrho\ s_n \to^* \mathcal{A}\ s_n\ v_n & & \mathcal{E}\ \llbracket pv \rrbracket\ \varrho^D\ s^D \\
\quad \to \mathcal{E}\ \llbracket e_{n+1} \rrbracket\ \varrho\ s_{n+1} & & \quad \to \mathcal{E}\ \llbracket pv_{n+1} \rrbracket\ \varrho^D\ s^D_{n+1} \\
\quad \to \cdots \to \mathcal{A}\ s_{n+1}\ v_{n+1} & \text{(induction, } \sigma_{n+1}\text{)} & \quad \to \cdots \to \mathcal{A}\ s^D_{n+1}\ v_{n+1} \\
\quad \vdots & \vdots & \quad \vdots \\
\quad \to \mathcal{E}\ \llbracket e_{n+m} \rrbracket\ \varrho\ s_{n+m} & & \quad \to \mathcal{E}\ \llbracket pv_{n+m} \rrbracket\ \varrho^D\ s^D_{n+m} \\
\quad \to \cdots \to \mathcal{A}\ s_{n+m}\ v_{n+m} & \text{(induction, } \sigma_{n+m}\text{)} & \quad \to \cdots \to \mathcal{A}\ s^D_{n+m}\ v_{n+m} \\
\quad \to \mathcal{C}\ \llbracket f \rrbracket\ [v_1, \ldots, v_{n+m}]\ s & & \quad \to \mathcal{C}\ \llbracket g \rrbracket\ [v_{n+1}, \ldots, v_{n+m}]\ s^D \\
\quad \to \mathcal{E}\ \llbracket e^f \rrbracket\ \varrho_f\ s \to^* \mathcal{A}\ s\ v & \text{(induction, } \sigma_f\text{)} & \quad \to \mathcal{E}\ \llbracket pv^f \rrbracket\ \varrho_g\ s^D \to^* \mathcal{A}\ s^D\ v
\end{array}$$

The case where $(f, [pv_1, \ldots, pv_n]) \in \mathit{dom}\ \varphi$ proceeds in a similar way, appealing to the faithfulness of φ. □

We are now able to show the

Proposition A.3 (Mix equation)
For any well-formed program p with any input $s_1, \ldots, s_n$, $d_1, \ldots, d_n$, if p' is a well-annotated version of p then

$$\mathcal{PP}\ \llbracket p' \rrbracket\ [s_1, \ldots, s_n] \to pv \textit{ implies}$$

$$\mathcal{P}\ \llbracket p \rrbracket\ [s_1, \ldots, s_n, d_1, \ldots, d_m] \to \mathcal{E}\ \llbracket pv \rrbracket\ \{f1_{n+1} \mapsto d_1, \ldots, f1_{n+m} \mapsto d_m\}\ [\,]$$

Proof: In the initial call, $\varphi(x) = \bot$ for all x and is thus trivially faithful, and as

$$\begin{array}{l}
\mathcal{PP}\ \llbracket p' \rrbracket\ [s_1, \ldots, s_n] \to \mathcal{PC}\ \llbracket f1 \rrbracket\ [\ldots]\ [\,] \to \\
\mathcal{PE}\ \llbracket e^{f1} \rrbracket\ \varrho'\ [\xi] \to^* \mathcal{PA}\ [\xi]\ e^g \to \mathcal{PA}\ [\,]\ pv \to pv,
\end{array}$$

we find by the preceding lemma that

$$\mathcal{P}\ \llbracket p \rrbracket\ [s_1, \ldots, s_n, d_1, \ldots, d_m] \to \mathcal{C}\ \llbracket f1 \rrbracket\ [\ldots]\ [\,] \to \mathcal{E}\ \llbracket e^{f1} \rrbracket\ \varrho\ [\,] \to^* \mathcal{A}\ [\,]\ v \to v$$

and

$$\begin{array}{l}
\mathcal{E}\ \llbracket pv \rrbracket\ \varrho^D\ [\,] \to \mathcal{E}\ \llbracket g\ f1_{n+1} \cdots f1_{n+m} \rrbracket\ \varrho^D\ [\,] \\
\to^* \mathcal{E}\ \llbracket e^g \rrbracket\ \{g_1 \mapsto d_1, \ldots, g_m \mapsto d_m\}\ [\,] \to^* \mathcal{A}\ [\,]\ v \to v
\end{array}$$

□

Appendix B

Examples

Note: In Scheme, True is represented by #t and False by #f.

B.1 Basic data manipulation

Contrived test program (contrived-1)

```
;; A contrived example from Arne Glenstrup's Master's Thesis
;; Numbers represented by list length
(define (contrived−1 a b) (f a (cons 1 (cons 1 a)) a b))
(define (f x y z d)
  (if (and (> z 'zero) (> d 'zero))
      (f (cons 1 x) z (cdr z) (cdr d))
      (if (> y 'zero) (g x (cdr y) d) x)))
(define (g u v w)
  (if (> w 'zero)
      (f (cons 1 u)
         (if (equal? (h v 'zero) 'zero) v (dec v))
         (cdr v)
         (cdr w))
      u))
(define (h r s)
  (if (> r 'zero)
      (h (cdr r) 42)
      (if (> s 'zero) (h r (cdr s)) r)))
(define (dec n) (cdr n))
```

Contrived test program, resetting h_r (contrived-2)

```
;; A contrived example from Arne Glenstrup's Master's Thesis
;; Numbers represented by list length
(define (contrived-2 a b) (f a (cons 1 (cons 1 a)) a b))
(define (f x y z d)
  (if (and (> z 'zero) (> d 'zero))
      (f (cons 1 x) z (cdr z) (cdr d))
      (if (> y 'zero) (g x (cdr y) d) x)))
(define (g u v w)
  (if (> w 'zero)
      (f (cons 1 u)
         (if (equal? (h v 'zero) 'zero) v (dec v))
         (cdr v)
         (cdr w))
      u))
(define (h r s)
  (if (> r 'zero)
      (h (cdr r) 42)
      (if (> s 'zero) (h 17 (cdr s)) r)))
(define (dec n) (cdr n))
```

List predicate (list)

```
;; The predicate for checking whether the argument is a list
(define (goal x) (list? x))
(define (list? xs) (if (pair? xs) (list? (cdr xs)) (null? xs)))
```

List fold of fixed operator (fold)

```
;; The fold operators, using a fixed operator, op
(define (fold a xs) (cons (foldl a xs) (cons (foldr a xs) '())))
(define (foldl a xs)
  (if (pair? xs)
      (foldl (op a (car xs)) (cdr xs))
      a))

(define (foldr a xs)
  (if (pair? xs)
      (op (car xs) (foldr a (cdr xs)))
      a))

(define (op x1 x2) (+ x1 x2))
```

List map of fixed function (map)

```
 ;; The map function with fixed function f
(define (goal xs) (map xs))
(define (map xs)
  (if (equal? xs '())
      '()
      (cons (f (car xs)) (map (cdr xs)))))

(define (f x) (* x x))
```

Naïve list reverse (naiverev)

```
 ;; Naive reverse function
(define (goal xs) (naiverev xs))
(define (naiverev xs)
  (if (equal? xs '())
      xs
      (app (naiverev (cdr xs)) (cons (car xs) '()))))

(define (app xs ys)
  (if (equal? xs '()) ys (cons (car xs) (app (cdr xs) ys))))
```

Deep list reverse (deeprev)

```
 ;;; Recursively reverse all list elements in a data structure
 ;;; Example: (deeprev '((1 2 3) 4 5 6 (8 (9 10 11)) . 12))
 ;;;          ===>((3 2 1) 4 5 6 ((11 10 9) 8) . 12)
```

```
(define (goal x) (deeprev x))
(define (deeprev x)
  (if (pair? x)
      (deeprevapp x '())
      x))

(define (deeprevapp xs rest)
  (if (pair? xs)
      (deeprevapp (cdr xs) (cons (deeprev (car xs)) rest))
      (if (equal? xs '())
          rest
          (revconsapp rest xs))))

(define (revconsapp xs r)
  (if (pair? xs) (revconsapp (cdr xs) (cons (car xs) r)) r))
```

List append (append)

```
(define (goal x y) (append x y))
(define (append xs ys)
  (if (equal? xs '()) ys (cons (car xs) (append (cdr xs) ys))))
```

List merge (mergelists)

```
;;; Merge two lists
(define (goal xs ys) (merge xs ys))
(define (merge xs ys)
  (if (equal? xs '())
      ys
      (if (equal? ys '())
          xs
          (if (<= (car xs) (car ys))
              (cons (car xs) (merge (cdr xs) ys))
              (cons (car ys) (merge xs (cdr ys)))))))
```

List adding (addlists)

```
;;; Add two lists elementwise
(define (goal xs ys) (addlist xs ys))
(define (addlist xs ys)
  (if (pair? xs)
      (cons (+ (car xs) (car ys)) (addlist (cdr xs) (cdr ys)))
      '()))
```

Reverse append (revapp)

```
;;; Reverse list and append to rest
(define (goal x y) (revapp x y))
(define (revapp xs rest)
  (if (equal? xs '())
      rest
      (revapp (cdr xs) (cons (car xs) rest))))
```

List permutations (permute)

```
;; Compute all the permutations of a list
(define (goal xs) (permute xs))
(define (permute xs)
  (if (equal? xs '())
      '(())
      (select (car xs) '() (cdr xs))))

;; Select x as the first element and cons it onto
;; permutations of the remaining list represented by
;; the list of elements before x (reversed) and the
;; list of elements after x. Finally, recurse by moving
;; on to the next element in postfix
(define (select x revprefix postfix)
  (mapconsapp x (permute (revapp revprefix postfix))
              (if (equal? postfix '())
                  '()
                  (select (car postfix)
                          (cons x revprefix)
                          (cdr postfix)))))

;; Map '(cons x' onto the list of lists xss and append the rest
(define (mapconsapp x xss rest)
  (if (equal? xss '())
      rest
      (cons (cons x (car xss)) (mapconsapp x (cdr xss) rest))))

;; Reverse xs and append the rest
(define (revapp xs rest)
  (if (equal? xs '())
      rest
      (revapp (cdr xs) (cons (car xs) rest))))
```

Unary addition (add)

```
;; Add two numbers unarily represented as '(s s s ... s)
(define (goal x y) (add x y))
(define (add x y) (if (equal? y '()) x (add (cons 1 x) (cdr y))))
```

Bad addition function (badd)

```
(define (goal x y) (badd x y))
(define (badd x y)
  (if (equal? y '()) x (badd '(1) (badd x (cdr y)))))
```

Unary multiplication (mul)

```
;;; Unary multiplication and addition, e.g. (mul '(s s z) '(s s s z))
(define (goal x y) (mul x y))
(define (mul x y) (if (equal? x '()) '() ( add (mul (cdr x) y) y)))
(define (add x y) (if (equal? x '()) y (add (cdr x) (cons 's y))))
```

Disjunctive and conjunctive expression predicates (disjconj)

```
;;; Predicates for disjunctive and conjunctive terms p
(define (disjconj p) (disj? p))
(define (disj? p)
  (if (pair? p)
      (if (equal? 'Or (car p))
          (and (conj? (cadr p)) (disj? (cddr p)))
```

```
            (conj? p))
        (conj? p)))

(define (conj? p)
   (if (pair? p)
        (if (equal? 'And (car p))
            (and (disj? (cadr p)) (conj? (cddr p)))
            (bool? p))
        (bool? p)))

(define (bool? p) (or (equal? 'F p) (equal? 'T p)))
```

List duplication (duplicate)

```
 ;;; Compute a list where each element is duplicated
(define (goal x) (duplicate x))
(define (duplicate xs)
   (if (equal? xs '())
        '()
        (cons (car xs) (cons (car xs) (duplicate (cdr xs))))))
```

Immaterial list “copy” (nestimeql)

```
 ;; Using an immaterial "copy" as recursive argument
(define (goal x) (nestimeql x))
(define (nestimeql x)
   (if (equal? x '()) 42 (nestimeql (immatcopy x))))
(define (immatcopy x)
   (if (equal? x '()) '() (cons '0 (immatcopy (cdr x)))))
```

Even/odd predicate (evenodd)

```
 ;;; Predicate: is x, unarily represented as '(s s s ... s), even/odd?
(define (evenodd x) (even? x))
(define (even? x) (if (null? x) #t (odd? (cdr x))))
(define (odd? x) (if (pair? x) (even? (cdr x)) #f))
```

Less than or equal predicate (lte)

```
;; Less than or equal
(define (goal x y) (and (lte? x y) (even? x)))
(define (lte? x y)
  (if (null? x)
      #t
      (if (and (pair? x) (pair? y))
          (lte? (cdr x) (cdr y))
          #f)))

(define (even? x)
  (if (null? x)
      #t
  (if (null? (cdr x))
      #f
      (even? (cdr (cdr x))))))
```

Member predicate (member)

```
;; The member function
(define (goal x xs) (member? x xs))
(define (member? x xs)
  (if (equal? xs '())
      (if (equal? x (car xs))
          #t
          (member? x (cdr xs)))
      #f))
```

Ordered list predicate (ordered)

```
;; Predicate that checks whether a list is ordered
(define (goal xs) (ordered? xs))
(define (ordered? xs)
  (if (pair? xs)
      (if (pair? (cdr xs))
          (if (<= (car xs) (cadr xs))
              (ordered? (cddr xs))
              #f)
```

```
            #t)
        #t))
```

Set overlap predicate (overlap)

```
;; Predicate for checking whether there is an overlap of two sets
(define (goal xs ys) (overlap? xs ys))
;(define (has-a-or-b? xs) (overlap? xs (cons 'a (cons 'b '()))))
(define (overlap? xs ys)
  (if (pair? xs)
      (if (member? (car xs) ys)
          #t
          (overlap? (cdr xs) ys))
      #f))
(define (member? x xs)
  (if (pair? xs)
      (if (equal? (car xs) x)
          #t
          (member? x (cdr xs)))
      #f))
```

Element selection (select)

```
;; Compute a list of lists . Each list is computed by picking out an
;; element of the original list and consing it onto the rest of the list
(define (select xs)
  (if (equal? xs '())
      '()
      (selects (car xs) '() (cdr xs))))

(define (selects x revprefix postfix)
  (cons (cons x (revapp revprefix postfix))
        (if (equal? postfix '())
            '()
            (selects (car postfix) (cons x revprefix) (cdr postfix)))))

;; Reverse xs and append to rest
(define (revapp xs rest)
  (if (equal? xs '()) rest (revapp (cdr xs) (cons (car xs) rest))))
```

List subsets generation (subsets)

```
;; Compute all subsets
(define (goal xs) (subsets xs))
(define (subsets xs)
  (if (pair? xs)
      (let* ((subs (subsets (cdr xs))))
        (mapconsapp (car xs) subs subs))
       ′(())))

;; map ‘(cons x′ ont the list of lists xss, and append rest
(define (mapconsapp x xss rest)
  (if (pair? xss)
      (cons (cons x (car xss)) (mapconsapp x (cdr xss) rest))
      rest))
```

Parameter anchoring (anchored)

```
;; Parameter y anchored in parameter x
(define (goal x y) (anchored x y))
(define (anchored x y)
  (if (equal? x ′()) y (anchored (cdr x) (cons 1 y))))
```

Let expression (letexp)

```
;; Testing the let construction
(define (goal x y) (letexp x y))
(define (letexp x y) (let* ((z (cons 1 x))) (letexp z y)))
```

The Trick (thetrick)

```
;; The trick : pulling out the conditional into the context
(define (goal x y) (cons (f x y) (cons (g x y) ′())))
(define (f x y)
  (if (equal? y ′())
      42
      (f (if (lt x ′(1)) x       (cdr x))
         (if (lt x ′(1)) (cdr y) (cons 1 y)))))
```

```
(define (g x y)
  (if (equal? y '())
      42
      (if (lt x '(1))
                              (g x       (cdr y))
                              (g (cdr x) (cons 1 y)))))
(define (lt x y)
  (if (equal? y '())
      #f
      (if (equal? x '()) #t (lt (cdr x) (cdr y)))))
```

Interpreter lookup engine (intlookup)

```
;; The function call case of an interpreter
;; Function number represented as list length
(define (run e p) (intlookup e p))
(define (intlookup e p) (intlookup (lookup e p) p))
(define (lookup fnum p)
  (if (equal? fnum '()) (car p) (lookup (cdr fnum) (cdr p))))
```

No simple lexicographic ordering (nolexicord)

```
;; Example not termination—provable by simple lexicographical ordering
(define (goal a1 b1 a2 b2 a3 b3) (nolexicord a1 b1 a2 b2 a3 b3))
(define (nolexicord a1 b1 a2 b2 a3 b3)
  (if (equal? a1 '())
      42
      (if (equal? a1 b1)
          (nolexicord
            (cdr b1) (cdr a1) (cdr a2) (cdr b2) (cdr b3) (cdr a3))
          (nolexicord
            (cdr b1) (cdr a1) (cdr b2) (cdr a2) (cdr a3) (cdr b3)))))
```

Decreasing loop (decrease)

```
(define (goal x) (decrease x))
(define (decrease x) (if (equal? x '()) 42 (decrease (cdr x))))
```

Stable loop (equal)

```
(define (goal x) (equal x))
(define (equal x) (if (equal? x '()) 42 (equal x)))
```

Increasing loop (increase)

```
(define (goal x) (increase x))
(define (increase x) (if (equal? x '()) 42 (increase (cons 1 x))))
```

Decreasing loop with nested call (nestdec)

```
;; Parameter decrease by nested call in recursion
(define (goal x) (nestdec x))
(define (nestdec x) (if (equal? x '()) 17 (nestdec (dec x))))
(define (dec x) (if (equal? x '(1)) (cdr x) (dec (cdr x))))
```

Stable loop with nested call (nesteql)

```
;; Parameter equality by nested call in recursion
(define (goal x) (nesteql x))
(define (nesteql x) (if (equal? x '()) 17 (nesteql (eql x))))
(define (eql x) (if (equal? x '()) x (eql x)))
```

Increasing loop with nested call (nestinc)

```
;; Parameter increase by nested call in recursion
(define (goal x) (nestinc x))
(define (nestinc x) (if (equal? x '()) 17 (nestinc (inc x))))
(define (inc x) (if (equal? x '()) '(1) (cons 1 (inc (cdr x)))))
```

Loops requiring specialisation point (sp1)

```
;; Mutual recursion requiring specialisation points
(define (sp1 x y) (f x y))
(define (f x y) (if (equal? x '()) (g x y) (h x y)))
(define (g x y) (if (equal? x '()) (h x y) (r x y)))
(define (h x y) (if (equal? x '()) (h x y) (f x y)))
(define (r x y) x)
```

Shuffle list (shuffle)

```
;; Shuffle List
(define (goal xs) (shuffle xs))
(define (shuffle xs)
  (if (equal? xs '())
      '()
      (cons (car xs) (shuffle (reverse (cdr xs))))))
(define (reverse xs)
  (if (equal? xs '())
      xs
      (append (reverse (cdr xs)) (cons (car xs) '()))))
(define (append xs ys)
  (if (equal? xs '())
      ys
      (cons (car xs) (append (cdr xs) ys))))
```

Rewrite expression with associative operator (assrewrite)

```
;;; Rewrite expression with associative operator 'op'

;;;      a -> a1    b -> b1    c -> c1
;;; -----------------------------------------
;;; '(op (op a b) c) -> '(op a1 (op b1 c1))

;;; a != 'op  a -> a1  b -> b1            a != '(op ...)
;;; ------------------------------   ------------
;;;   '(op a b) -> '(op a1 b1)               a -> a

(define (assrewrite exp) (rewrite exp))
(define (rewrite exp)
  (if (and (pair? exp)
           (equal? 'op (car exp)))
      (let* ((opab (cadr exp)))
        (if (and (pair? opab)
                 (equal? 'op (car opab)))
            (let* ((a1 (rewrite (cadr opab)))
                   (b1 (rewrite (caddr opab)))
                   (c1 (rewrite (caddr exp))))
```

```
                (rewrite (cons
                           (car exp ) ; op
                           (cons
                            a1
                             (cons
                              (cons
                               (car opab ) ; op
                               (cons b1 (cons c1 (cdddr opab ))))
                              (cdddr exp ))))))
              (cons (car exp ) ; op
                    (cons
                     (rewrite (cadr exp ))   ; a
                     (cons
                      (rewrite (caddr exp )) ; b
                      (cdddr exp ))))))
        exp))
```

Game (game)

```
;; The game function from Manuvir Das' PhD Thesis (p. 137)
(define (goal p1 p2 moves) (game p1 p2 moves))
(define (game p1 p2 moves)
  (if (equal? moves '())
      (cons p1 p2)
      (if (equal? (car moves) 'swap)
          (game p2 p1 (cdr moves))
          (if (equal? (car moves) 'capture)
              (game (cons (car p2) p1) (cdr p2) (cdr moves))
              'error ))))
```

Van Gelder example (vangelder)

```
;;; Following is an example due to Allen Van Gelder.
;;; Note that in the following example there is a
;;; cycle involving q, p, r, t, and q again, such that
;;; nothing gets smaller along that cycle.

;;; e(a,b).
;;; q(X,Y)       :- e(X,Y).
;;; q(X,f(f(X))) :- p(X,f(f(X))), q(X,f(X)).
;;; q(X,f(f(Y))) :- p(X,f(Y)).
;;;
;;; p(X,Y)          :- e(X,Y).
;;; p(X,f(Y))       :- r(X,f(Y)), p(X,Y).
;;;
;;; r(X,Y)          :- e(X,Y).
;;; r(X,f(Y))       :- q(X,Y), r(X,Y).
;;; r(f(X),f(X)) :- t(f(X),f(X)).
;;;
;;; t(X,Y)          :- e(X,Y).
;;; t(f(X),f(Y)) :- q(f(X),f(Y)), t(X,Y).

(define (goal x y) (q x y))
(define (e a b) (and (equal? a 'a) (equal? b 'b)))
(define (q x y)
  (if (e x y) #t
      (if (and (pair? y) (equal? (car y) 'f)
               (pair? (cdr y)) (equal? (cadr y) 'f))
          (if (and (p x y) (q x (cdr y))) #t
              (p x (cdr y)))
          #f)))
(define (p x y)
  (if (e x y) #t
      (if (equal? 'f (car y))
          (and (r x y) (p x (cdr y)))
          #f)))
(define (r x y)
  (if (e x y) #t
      (if (and (pair? y) (equal? (car y) 'f))
          (if (and (q x (cdr y)) (r x (cdr y)))
              #t
              (if (and (pair? x) (equal? (car x) 'f))
                  (t x y)
                  #f))
          #f)))
(define (t x y)
  (if (e x y) #t
      (if (and (pair? x) (equal? (car x) 'f)
                (pair? y) (equal? (car y) 'f))
            (and (q x y) (t (cdr x) (cdr y)))
            #f)))
```

B.2 Simple functions

Power function (power)

```
;; Power function: x to the nth power
;; ( numbers represented by list length)
(define (goal x n) (power x n))
(define (power x n)
  (if (equal? n '()) '(1) (mult x (power x (cdr n)))))
(define (mult x y)
  (if (equal? y '()) '() (add x (mult x (cdr y)))))
(define (add x y)
  (if (equal? y '()) x (cons 1 (add x (cdr y)))))
```

Binomial function (binom)

```
;;; Binomial function, numbers represented by list length
(define (goal n k) (binom n k))
(define (binom n k)
  (if (equal ? '() n)
      '(1)
      (if (equal ? '() k)
          '(1)
          (+ (binom (cdr n) (cdr k)) (binom (cdr n) k)))))
```

Ackermann's function (ack)

```
;;; Ackermann's function, numbers represented by list length
(define (goal m n) (ack m n))
(define (ack m n)
  (if (equal ? '() m)
      (cons 1 n)
      (if (equal ? '() n)
          (ack (cdr m) '(1))
          (ack (cdr m) (ack m (cdr n))))))
```

Greatest common divisor (gcd-1)

```
;; Greatest common divisor, numbers represented by list length
(define (goal x y) (gcd x y))
(define (gcd x y)
  (if (or (equal? x '()) ( equal? y '()))
      'error
      (if (equal? x y)
          x
          (if (gt x y) (gcd (monus x y) y) (gcd x (monus y x))))))
(define (gt x y)
  (if (equal? x '())
      #f
      (if (equal? y '()) # t (gt (cdr x) (cdr y)))))
(define (monus x y)
  (if (equal? (lgth y) 1)
      (cdr x)
      (monus (cdr x) (cdr y))))
(define (lgth x) (if (equal? x '()) 0 (+ 1 ( lgth (cdr x)))))
```

Greatest common divisor, swapping x and y (gcd-2)

```
;; Greatest common divisor, numbers represented by list length
(define (goal x y) (gcd x y))
(define (gcd x y)
  (if (or (equal? x '()) ( equal? y '()))
      'error
      (if (equal? x y)
          x
          (if (gt x y) (gcd y (monus x y)) (gcd (monus y x) x)))))
(define (gt x y)
  (if (equal? x '())
      #f
      (if (equal? y '()) # t (gt (cdr x) (cdr y)))))
(define (monus x y)
  (if (equal? (lgth y) 1)
      (cdr x)
      (monus (cdr x) (cdr y))))
(define (lgth x) (if (equal? x '()) 0 (+ 1 ( lgth (cdr x)))))
```

B.3 Sorting

Mergesort (mergesort)

```
;; Mergesort
(define (goal xs) (mergesort xs))
(define (mergesort xs)
  (if (pair? xs)
      (if (pair? (cdr xs))
          (splitmerge xs '() '())
          xs)
      xs))

(define (splitmerge xs xs1 xs2)
  (if (pair? xs)
      (splitmerge (cdr xs) (cons (car xs) xs2) xs1)
      (merge (mergesort xs1) (mergesort xs2))))

(define (merge xs1 xs2)
  (if (pair? xs1)
      (if (pair? xs2)
          (if (<= (car xs1) (car xs2))
              (cons (car xs1) (merge (cdr xs1) xs2))
              (cons (car xs2) (merge xs1 (cdr xs2))))
          xs1)
      xs2))
```

Quicksort (quicksort)

```
;; Quicksort
(define (goal xs) (quicksort xs))
(define (quicksort xs)
  (if (pair? xs)
      (if (pair? (cdr xs))
          (part (car xs) xs (cons (car xs) '()) '())
          xs)
      xs))

(define (part x xs xs1 xs2)
  (if (pair? xs)
      (if (> x (car xs))
          (part x (cdr xs) (cons (car xs) xs1) xs2)
          (if (< x (car xs))
              (part x (cdr xs) xs1 (cons (car xs) xs2))
              (part x (cdr xs) xs1 xs2)))
      (app (quicksort xs1) (quicksort xs2))))

(define (app xs ys)
  (if (pair? xs)
      (cons (car xs) (app (cdr xs) ys))
      ys))
```

Minimum sort (minsort)

```
 ;;; Minimum sort: remove minimum and cons it onto the rest, sorted.
(define (goal xs) (minsort xs))
(define (minsort xs)
  (if (pair? xs)
      (appmin (car xs) (cdr xs) xs)
       '()))

(define (appmin min rest xs)
  (if (pair? rest)
      (if (< (car rest) min)
          (appmin (car rest) (cdr rest) xs)
          (appmin min (cdr rest) xs))
      (cons min (minsort (remove min xs)))))

(define (remove x xs)
  (if (pair? xs)
      (if (equal? x (car xs))
          (cdr xs)
          (cons (car xs) (remove x (cdr xs))))
       '()))
```

B.4 Larger algorithms

Graph reachability (reach)

```
;;; How can node v be reached from node u in a directed graph.
;;; Graph example : '((a . b) (a . d) (b . d) (c . a))
(define (goal u v edges) (reach u v edges))
(define (reach u v edges)
  (if (member? (cons u v) edges)
      (cons (cons u v) '())
      (via u v edges edges)))

(define (via u v rest edges)
  (if (equal? rest '())
      '()
      (if (equal? u (caar rest))
          (let * ((path (reach (cdar rest) v edges)))
            (if (equal? path '())
                (via u v (cdr rest) edges)
                (cons (car rest) path)))
          (via u v (cdr rest) edges))))

(define (member? x xs)
  (if (equal? xs '()) # f
      (if (equal? x (car xs)) # t (member? x (cdr xs)))))
```

Graph colouring (graphcolour-1)

```
;;; Colour graph G with colours cs so that neighbors have different colours
;;; The graph is represented as a list of nodes with adjacency lists
;;; Example:
;;; '((a . (b c d)) (b . (a c e)) (c . (a b d e f)) (d . (a c f))
;;;     (e . (b c f)) (f . (c d e)))
(define (graphcolour G cs)
  (let * ((ns G)) ; to speed up: (ns (sortnodesbyarity G))
    (reverse
     (colorrest cs cs
                (cons (colornode cs (car ns) '()) '())
                (cdr ns)))))
```

```

;;; Colour a node by appending a colour list to the node. The head of
;;; the list is the chosen colour, the tail are the yet untried
;;; colours. If impossible, return nil.
;;; Example of coloured node: '((red blue yellow) . (a . (b c d)))
(define (colornode cs node colorednodes)
  (if (pair? cs)
      (if (possible (car cs) (cdr node) colorednodes)
          (cons cs node)
          (colornode (cdr cs) node colorednodes))
      '()))

;;; Can we use color with these adjacent nodes and current coloured nodes?
(define (possible color adjs colorednodes)
  (if (pair? adjs)
      (if (equal? color (colorof (car adjs) colorednodes))
          #f
          (possible color (cdr adjs) colorednodes))
      #t))

;;; Return colour of node. If no colour yet, return nil.
(define (colorof node colorednodes)
  (if (pair? colorednodes)
      (if (equal? (cadar colorednodes) node)
          (caaar colorednodes)
          (colorof node (cdr colorednodes)))
      '()))

;;; Colour the first node of rest with colours from ncs, and
;;; colour remaining nodes. If impossible, return nil.
(define (colorrest cs ncs colorednodes rest)
  (if (pair? rest)
      (let* ((colorednode (colornode ncs (car rest) colorednodes)))
        (if (pair? colorednode)
            (let* ((colored (colorrest cs cs
                                       (cons colorednode colorednodes)
                                       (cdr rest))))
              (if (pair? colored)
                  colored
                  ; if remaining nodes are not colourable , and there
                  ( if ( pair ? ( car colorednode )) ; are colours left ,
                      ( colorrest −thetrick
                       cs cs ( cdr ( car colorednode )) ; try next colour
                       colorednodes rest )
                       '()))
            '()))
      colorednodes ))

( define ( colorrest −thetrick cs1 cs ncs colorednodes rest )
  ( if ( equal ? cs1 ncs)
      ( colorrest cs cs1 colorednodes rest )
      ( colorrest −thetrick ( cdr cs1 ) cs ncs colorednodes rest )))

( define ( reverse xs ) ( revapp xs '()))
( define (revapp xs rest )
  ( if ( pair ? xs)
      (revapp ( cdr xs ) ( cons ( car xs ) rest ))
      rest ))
```

Graph colouring, tail recursive (graphcolour-2)

```
 ;;; Colour graph G with colours cs so that neighbors have different
 ;;; colours ( slightly tail recursive version )
 ;;; The graph is represented as a list of nodes with adjacency lists
 ;;; Example:
 ;;; '(( a . ( b c d )) ( b . ( a c e )) ( c . ( a b d e f )) ( d . ( a c f ))
 ;;;    (e . ( b c f )) ( f . ( c d e )))
( define (graphcolour G cs)
  ( let ∗ (( ns G )) ; to speed up : ( ns ( sortnodesbyarity G))
    ( reverse
     ( colorrest cs cs
                 (cons ( colornode cs ( car ns ) '()) '())
                 (cdr ns )))))

 ;;; Colour a node by appending a colour list to the node . The head of
 ;;; the list is the chosen colour , the tail are the yet untried
 ;;; colours . If impossible , return nil .
 ;;; Example of coloured node : '(( red blue yellow ) . ( a . ( b c d )))
```

```
(define (colornode cs node colorednodes)
  (if (pair? cs)
      (if (possible (car cs) (cdr node) colorednodes)
          (cons cs node)
          (colornode (cdr cs) node colorednodes))
      '()))

;;; Can we use color with these adjacent nodes and current coloured nodes?
(define (possible color adjs colorednodes)
  (if (pair? adjs)
      (if (equal? color (colorof (car adjs) colorednodes))
          #f
          (possible color (cdr adjs) colorednodes))
      #t))

;;; Return colour of node. If no colour yet, return nil.
(define (colorof node colorednodes)
  (if (pair? colorednodes)
      (if (equal? (cadar colorednodes) node)
          (caaar colorednodes)
          (colorof node (cdr colorednodes)))
      '()))

;;; Colour the first node of rest with colours from ncs, and
;;; colour remaining nodes. If impossible, return nil.
(define (colorrest cs ncs colorednodes rest)
  (if (pair? rest)
      (colornoderest cs ncs (car rest) colorednodes rest)
      colorednodes))

;;; Like colornode, only continue with colouring the rest
(define (colornoderest cs ncs node colorednodes rest)
  (if (pair? ncs)
      (if (possible (car ncs) (cdr node) colorednodes)
          (let* ((colored (colorrest cs cs
                                     (cons (cons ncs node) colorednodes)
                                     (cdr rest))))
            (if (pair? colored)
                colored
```

```
                  ; if remaining nodes are not colourable , and
                  ( if ( pair ? ncs ) ; there are colours left ,
                      ( colorrest −thetrick
                       cs cs ( cdr ncs ) ; try next colour
                       colorednodes rest )
                      ′())))
            ( colornoderest cs ( cdr ncs ) node colorednodes rest ))
        ′()))

( define ( colorrest −thetrick cs1 cs ncs colorednodes rest )
  ( if ( equal ? cs1 ncs)
      ( colorrest cs cs1 colorednodes rest )
      ( colorrest −thetrick ( cdr cs1 ) cs ncs colorednodes rest )))

( define ( reverse xs ) ( revapp xs ′()))
( define (revapp xs rest )
  ( if ( pair ? xs)
      (revapp ( cdr xs ) ( cons ( car xs ) rest ))
      rest ))
```

Graph colouring, tail recursive without The Trick (graphcolour-3)

```
 ;;; Colour graph G with colours cs so that neighbors have different
 ;;; colours ( slightly tail recursive version )
 ;;; The graph is represented as a list of nodes with adjacency lists
 ;;; Example:
 ;;; ′(( a . ( b c d )) ( b . ( a c e )) ( c . ( a b d e f )) ( d . ( a c f ))
 ;;;     (e . ( b c f )) ( f . ( c d e )))
( define (graphcolour G cs)
  ( let ∗ (( ns G )) ; to speed up : ( ns ( sortnodesbyarity G))
    ( reverse
     ( colorrest cs cs
                (cons ( colornode cs ( car ns ) ′()) ′())
                (cdr ns )))))

 ;;; Colour a node by appending a colour list to the node . The head of
 ;;; the list is the chosen colour , the tail are the yet untried
 ;;; colours . If impossible , return nil .
 ;;; Example of coloured node : ′(( red blue yellow ) . ( a . ( b c d )))
```

```
(define (colornode cs node colorednodes)
  (if (pair? cs)
      (if (possible (car cs) (cdr node) colorednodes)
          (cons cs node)
          (colornode (cdr cs) node colorednodes))
      '()))

;;; Can we use color with adjacent nodes and current coloured nodes?
(define (possible color adjs colorednodes)
  (if (pair? adjs)
      (if (equal? color (colorof (car adjs) colorednodes))
          #f
          (possible color (cdr adjs) colorednodes))
      #t))

;;; Return colour of node. If no colour yet, return nil.
(define (colorof node colorednodes)
  (if (pair? colorednodes)
      (if (equal? (cadar colorednodes) node)
          (caaar colorednodes)
          (colorof node (cdr colorednodes)))
      '()))

;;; Colour the first node of rest with colours from ncs, and
;;; colour remaining nodes. If impossible, return nil.
(define (colorrest cs ncs colorednodes rest)
  (if (pair? rest)
      (colornoderest cs ncs (car rest) colorednodes rest)
      colorednodes))

;;; Like colornode, only continue with colouring the rest
(define (colornoderest cs ncs node colorednodes rest)
  (if (pair? ncs)
      (if (possible (car ncs) (cdr node) colorednodes)
          (let* ((colored (colorrest cs cs
                                     (cons (cons ncs node) colorednodes)
                                     (cdr rest))))
            (if (pair? colored)
                colored
```

```
                    ; if remaining nodes are not colourable , and
                    (if (pair? ncs) ; there are some colours left ,
                        (colorrest
                         cs (cdr ncs) ; try next colour
                         colorednodes rest)
                         '())))
              (colornoderest cs (cdr ncs) node colorednodes rest))
          '()))

(define (reverse xs) (revapp xs '()))
(define (revapp xs rest)
  (if (pair? xs)
      (revapp (cdr xs) (cons (car xs) rest))
      rest))
```

Type inference for the typed λ calculus (typeinf)

```
;;; Type inference for the Typed Lambda Calculus

;;; e ::= (' var . x)              variable x
;;;     | ('apply . (e1 . e2))     apply abstraction e1 to expression e2
;;;     | ('lambda . (x . e1))     make lambda abstraction

;;; t ::= (' tyvar . a)
;;;     | ('arrow . (t1 . t2))

;;; ———————————————————————————————————————

;;; (define inittenv 1); tenv simply holds the next fresh type variables
(define (typeinf inittenv e); infer the type of e
  (let* ((atenv (freshtvar inittenv)))
    (car (etype '() (cdr atenv) e (car atenv)))))

(define (freshtvar tenv) (cons (cons 'Tvar tenv) (+ tenv 1)))

(define (vtype venv x); return the type of x as found in environment
  (if (equal? x (caar venv))
      (cdar venv)
      (vtype (cdr venv) x)))
```

```
(define (tsubst a t t1); substitute t for occ's of type var a in t1
  (if (equal? 'Tvar (car t1))
      (if (equal? a (cdr t1)) t t1)
  (if (equal? 'Arr (car t1))
      (cons 'Arr (cons (tsubst a t (cadr t1)) (tsubst a t (cddr t1))))
  (error 'tsubst-t1))))

(define (subst venv a t); substitute t for occurrences of type
  (if (pair? venv)        ; var a in the variable environment
      (cons (cons (caar venv) (tsubst a t (cdar venv)))
            (subst (cdr venv) a t))
      '()))

(define (unify venv t1 t2); unify types t1 and t2, returning
                           ;(<new venv> . <unified type>)
  (if (equal? 'Tvar (car t1))
      (cons (subst venv (cdr t1) t2) t2)
  (if (equal? 'Arr (car t1))
      (if (equal? 'Tvar (car t2))
          (cons (subst venv (cdr t2) t1) t1)
      (if (equal? 'Arr (car t2))
          (let* ((venv1tx1 (unify venv (cadr t1) (cddr t1)))
                 (venv2tx2
                  (unify (car venv1tx1) (cadr t2) (cddr t2))))
            (cons (car venv2tx2)
                  (cons 'Arr (cons (cdr venv1tx1) (cdr venv2tx2)))))
      (error 'unify-t2)))
  (error 'unify-t1))))

(define (etype venv tenv e t); infer type of e, unified with type t
  (if (equal? 'Var (car e))   ; using variable environment
                              ; and tyvar generator
        (let* ((venv1t1 (unify venv (vtype venv (cdr e)) t)))
          (cons (cdr venv1t1) (cons (car venv1t1) tenv)))
    (if (equal? 'App (car e))
        (let* ((atenv1 (freshtvar tenv))
               (t2venv2tenv2
                (etype venv (cdr atenv1) (cadr e) (car atenv1)))
```

```
                  (t1venv3tenv3
                   (etype (cadr t2venv2tenv2)
                          (cddr t2venv2tenv2)
                          (cddr e)
                          (cons 'Arr (cons (car t2venv2tenv2) t))))
                  (t1 (car t1venv3tenv3)))
             ; t1 == ('Arr . (? . t'))
             (cons (cddr t1) (cdr t1venv3tenv3)))
       (if (equal? 'Lam (car e))
         (let* ((atenv1 (freshtvar tenv))
                (t1venv2tenv2
                 (etype (cons (cons (cadr e) (car atenv1)) venv)
                        (cdr atenv1) (cddr e) (car atenv1)))
                (venv3t3
                 (unify (cadr t1venv2tenv2)
                        (cons 'Arr
                              (cons (vtype (cddr t1venv2tenv2) (cadr e))
                                    (car t1venv2tenv2)))
                        t)))
           (cons (cdr venv3t3) (cons (cdar venv3t3) (cddr t1venv2tenv2))))
         (error 'Error-in-lambda-expression e)))))
```

Simple pattern matching (match)

```
;; Simple pattern matcher
(define (match p s) (loop p s p s))
(define (loop p s pp ss)
  (if (equal? p '())
      #t
  (if (equal? s '())
      #f
  (if (equal? (car p) (car s))
      (loop (cdr p) (cdr s) pp ss)
      (loop pp (cdr ss) pp (cdr ss))))))
```

Simple string matching (strmatch)

```
;;; Naive pattern string matcher
(define (strmatch patstr str)
  ; strmatch returns a list of indeces indicating the
  ; positions in str where patstr occurs
  (domatch (patstring->list patstr) (string->list str) 0))

(define (domatch patcs cs n)
  (if (pair? cs)
      (if (prefix patcs cs)
          (cons n (domatch patcs (cdr cs) (+ n 1)))
          (domatch patcs (cdr cs) (+ n 1)))
      (if (equal? patcs cs) (cons n '()) '())))

(define (prefix precs cs)
  (if (pair? precs)
      (if (pair? cs)
          (and (equal? (car precs) (car cs))
               (prefix (cdr precs) (cdr cs)))
          #f)
      #t))
```

Regular expression matching (rematch)

```
 ;;; Regular expression pattern matcher
 ;;;
 ;;; pat ::= "." | character | "\" character
 ;;;       | pat "*" | (pat) | pat ... pat
 ;;; When parsed, this is represented by:
 ;;; pat ::= ('dot) | ('char c) | ('star pat) | ('seq pat ... pat)
(define (rematch patstr str)
  ; if str matches patstr, match returns a pair consisting of
  ; the prefix of str which matches the pattern and
  ; the remaining part of str, else match returns #f
  (let* ((matchrest (domatch (parsepat patstr) (string->list str))))
    (if (pair? matchrest)
        (cons (list->string (reverse (car matchrest)))
              (list->string (cdr matchrest)))
        matchrest)))

(define (parsepat patstr) (parsep (string->list patstr) '() '()))

```

```
(define (parsep patchars seq stack)
  (if (pair? patchars)
      (if (equal? #\. (car patchars))
          (parsep-dot patchars seq stack)
      (if (equal? #\* (car patchars))
          (parsep-star patchars seq stack)
      (if (equal? #\( (car patchars))
          (parsep-openb patchars seq stack)
      (if (equal? #\) (car patchars))
          (parsep-closeb patchars seq stack)
      (if (equal? #\\ (car patchars))
          (parsep-char (cdr patchars) seq stack)
      (parsep-char patchars seq stack))))))
      ; else (pair? patchars)
      (if (pair? stack)
          (error "unmatched '(' in pattern")
          (cons 'seq (reverse seq)))))

(define (parsep-dot patchars seq stack)
  (parsep (cdr patchars) (cons (cons 'dot '()) seq) stack))

(define (parsep-star patchars seq stack)
  (if (pair? seq)
      (parsep
       (cdr patchars)
       (cons (cons 'star (cons (car seq) '()))
             (cdr seq))
       stack)
      (parsep
       (cdr patchars)
       (cons (cons 'char (cons #\* '())) '())
       stack)))

(define (parsep-openb patchars seq stack)
  (parsep (cdr patchars) '() (cons seq stack)))

(define (parsep-closeb patchars seq stack)
  (if (pair? stack)
      (parsep
       (cdr patchars)
       (cons (cons 'seq (reverse seq))
             (car stack))
       (cdr stack))
      (error "unmatched ')' in pattern")))

(define (parsep−char patchars seq stack)
  (if (pair? patchars)
      (parsep (cdr patchars)
                (cons (cons 'char (cons (car patchars) '()))
                      seq)
                stack)
      (parsep patchars
                (cons (cons 'char (cons #\\ '()))
                      seq)
                stack)))

; domatch∗ cs must match on as much of cs as possible,
; Assume cs = cs1 ++ cs2, where cs1 has been matched. Then
; ((reverse cs1) . cs2) is returned

(define (domatch pat cs)
  (if (pair? pat)
      (if (equal? (car pat) 'dot) (domatch−dot cs)
      (if (equal? (car pat) 'char) (domatch−char cs (cadr pat))
      (if (equal? (car pat) 'star) (domatch−star cs (cadr pat) '())
      (if (equal? (car pat) 'seq) (domatch−seq cs '() (cdr pat))
      (error "unknown pattern data" pat)))))
      (cons '() cs)))

(define (domatch−dot cs)
  (if (pair? cs) (cons (cons (car cs) '()) (cdr cs)) 'nomatch))

(define (domatch−char cs c)
  (if (pair? cs)
      (if (equal? (car cs) c)
          (cons (cons (car cs) '()) (cdr cs))
          'nomatch)
      'nomatch))
```

```

(define (domatch−star cs pat init)
  ; init holds the chars already star−matched
  (if (pair? cs)
      (let∗ ((first (domatch pat cs)))
        (if (pair? first)
            (domatch−star (cdr first) pat (append (car first) init))
            (cons init cs)))
      (cons init cs)))

(define (domatch−seq cs rest pats)
  ; domatch−seq matches first pattern on cs = match ++ cs' and the
  ; remaining patterns on cs' ++ rest
  (if (pair? pats)
      (let∗ ((first (domatch (car pats) cs)))
        (if (pair? first)
            (let∗ ((next
                     (domatch−seq
                      (append (cdr first) rest) '() (cdr pats))))
              (if (pair? next)
                  (cons (append (car next) (car first)) (cdr next))
                  ; first match was too long, try matching fewer chars
                  (if (pair? (car first))
                      (domatch−seq
                        (reverse (cdar first))
                        (cons (caar first) (append (cdr first) rest))
                        pats)
                      'nomatch))) ; even shortest possible first match
                                  ; (empty string) doesn't lead to match
            'nomatch)) ; first match failed
      (cons '() (append cs rest)))) ; no patterns left to
                                    ; match: success!
```

B.5 Interpreters

Turing machine (turing)

```
;;; Turing machine interpreter

;;; instrs ::= '(instr . instrs)
;;;        | '()
;;; instr ::= '(Halt)           ; Stop interpretation
;;;        | '(Write . x)      ; Write x onto the tape at current pos
;;;        | '(Left)           ; Move pos left, extend tape if needed
;;;        | '(Right)          ; Move pos right, extend tape if needed
;;;        | '(Goto . i)       ; Continue at instruction i
;;;        | '(IfGoto x . i) ; If current pos contains x, goto i
(define (run prog tapeinput) (turing prog '() tapeinput prog))
(define (turing instrs revltape rtape prog)
 (if (pair? instrs)
   (if (equal? 'Halt (caar instrs))
     rtape
   (if (equal? 'Write (caar instrs))
     (turing (cdr instrs)
         revltape (cons (cdar instrs) (cdr rtape)) prog)
   (if (equal? 'Left (caar instrs))
     (if (pair? revltape)
       (turing (cdr instrs)
           (cdr revltape)
           (cons (car revltape) rtape) prog)
       (turing (cdr instrs)
           '()
           (cons 'Blank rtape) prog))
   (if (equal? 'Right (caar instrs))
     (if (pair? rtape)
       (turing (cdr instrs)
           (cons (car rtape) revltape)
           (cdr rtape) prog)
       (turing (cdr instrs)
           (cons 'Blank revltape)
           '() prog))
   (if (equal? 'Goto (caar instrs))
     (turing (lookup (cdar instrs) prog) revltape rtape prog)
   (if (equal? 'IfGoto (caar instrs))
     (if (equal? (car rtape) (cadar instrs))
       (turing
        (lookup (cddar instrs) prog) revltape rtape prog)
```

```
                (turing (cdr instrs) revltape rtape prog))
            rtape
            ))))))
        rtape
        ))

(define (lookup i instrs)
  (if (= i 1) instrs (lookup (- i 1) (cdr instrs))))
```

Expression parser (parsexp)

```
;;; Parse a list of atoms as an expression. Return remaining list.
;;; e.g. '("5" "*" "(" "3" "+" "2" "*" "4" ")")
(define (parsexp xs) (expr xs))
(define (expr xs)
  (let* ((rs1 (term xs)))
    (if (equal? '() rs1)
        '()
        (if (member? (car rs1) '("+" "-"))
            (let* ((rs2 (expr (cdr rs1))))
              (if (equal? '() rs2) rs1 rs2))
            rs1))))

(define (term xs)
  (let* ((rs1 (factor xs)))
    (if (equal? rs1 '())
        '()
        (if (member? (car rs1) '("*" "/"))
            (let* ((rs2 (term (cdr rs1))))
              (if (equal? '() rs2) rs1 rs2))
            rs1))))

(define (factor xs)
  (if (equal? "(" (car xs))
      (let* ((rs1 (expr xs)))
        (if (and (not (equal? rs1 '()))
                 (equal? ")" (car rs1)))
            (cdr rs1)
            (atom xs)))
      (atom xs )))

( define  ( member? x xs)
  ( if  ( pair ? xs)
      ( if  ( equal ? x ( car  xs ))
          #t
          (member? x (cdr  xs )))
      #f ))

( define  ( atom xs ) ( if  ( pair ? xs ) ( cdr  xs  ) '()))
```

Lambda expression interpreter (lambdaint)

```
 ;; Reducer for  the  lambda  calculus
 ;;    Representation :
 ;;      R [[n]]        = (1 0 ... 0)    n  zeros
 ;;      R [[\n.e ]] = (2  R [[n]]  R [[e ]])
 ;;      R [[e  e ']] = (3  R [[e ]]  R [[e ']])
( define  (lambdaint  e ) ( red  e ))
( define  ( red  e ) ;  reduce  lambda  expression  e
  ( if  ( isvar ? e)
      e
  ( if  ( islam ? e)
      e
      ( let * (( f  ( red  (app−>e1 e ))) ( a  ( red  ( app−>e2 e ))))
        ( if  ( islam ? f)
            (red  ( subst  ( lam−>var f) a  ( lam−>body f)))
            (mkapp f  a  ))))))

( define  ( subst  x  a  e)
  ( if  ( isvar ? e)
      ( if  ( equal ? x  e ) a  e)
      ( if  ( islam ? e)
          ( if  ( equal ? x  ( lam−>var e))
              e
              (mklam (lam−>var e) (subst  x  a  ( lam−>body e))))
          (mkapp (subst  x  a  ( app−>e1 e)) ( subst  x  a  ( app−>e2 e ))))))

( define  ( isvar ? e ) ( equal ? ( car  e ) 1))
```

```
(define (islam? e) (equal? (car e) 2))

(define (mklam n e) (cons 2 (cons n (cons e '()))))
(define (lam->var e) (cadr e))
(define (lam->body e) (caddr e))
(define (mkapp e1 e2) (cons 3 (cons e1 (cons e2 '()))))
(define (app->e1 e) (cadr e))
(define (app->e2 e) (caddr e))
```

LOOP interpreter (int-loop)

```
 ;; Small 1st order interpreter for LOOP programs
(define (run p l input)
  (let* ((f0 (car (car p)))
         (ef (lookbody f0 p))
         (nf (lookname f0 p)))
    (eeval ef (cons nf '()) ( cons input '()) l p)))

(define (eeval e ns vs l p)
  (if (equal? (car e) 1); constants
      (cdr e)
  (if (equal? (car e) 2); variable
      (lookvar (cdr e) ns vs)
  (if (equal? (car e) 3); basefcn
      (let* ((v1 (eeval (car (cdr (cdr e))) ns vs l p))
             (v2 (eeval (car (cdr (cdr (cdr e)))) ns vs l p)))
        (apply (car (cdr e)) v1 v2))
  (if (equal? (car e) 4)  ; if
      (if (equal? (eeval (car (cdr e)) ns vs l p) 'T)
          (eeval (car (cdr (cdr e))) ns vs l p)
          (eeval (car (cdr (cdr (cdr e)))) ns vs l p))
  (if (equal? (car e) 5)  ; ==
      (if (equal? (eeval (car (cdr e)) ns vs l p)
                  (eeval (car (cdr (cdr e))) ns vs l p))
          'T
          'F)
                               ; call
  (let* ((ef (lookbody (car (cdr e)) p))
         (nf (lookname (car (cdr e)) p))
```

```
           (v  (eeval (car (cdr (cdr e))) ns vs l p)))
     (if (equal? l '()) '()
           (eeval ef (cons nf '()) (cons v '()) (cdr l) p )))))))))

(define (lookvar x ns vs)
  (if (equal? x (car ns)) (car vs) (lookvar x (cdr ns) (cdr vs))))

(define (lookbody f p)
  (if (equal? (car (car p)) f)
      (car (cdr (cdr (car p))))
      (lookbody f (cdr p))))

(define (lookname f p)
  (if (equal? (car (car p)) f)
      (car (cdr (car p)))
      (lookname f (cdr p))))

(define (apply op v1 v2)
  (if (equal? op 5) ; equal
      (if (equal? v1 v2) 'T 'F)
                      ; cons
      (cons v1 v2)))
```

WHILE interpreter (int-while)

```
;; Small 1st order interpreter with static scoping
(define (run p input)
  (let* ((f0 (car (car p)))
         (ef (lookbody f0 p))
         (nf (lookname f0 p)))
     (eeval ef (cons nf '()) (cons input '()) p)))

(define (eeval e ns vs p)
  (if (equal? (car e) 1) ; constants
      (cdr e)
  (if (equal? (car e) 2) ; variable
      (lookvar (cdr e) ns vs)
  (if (equal? (car e) 3) ; basefcn
      (let* ((v1 (eeval (car (cdr (cdr e))) ns vs p))
             (v2 (eeval (car (cdr (cdr (cdr e)))) ns vs p)))
        (apply (car (cdr e)) v1 v2))
  (if (equal? (car e) 4)   ; if
      (if (equal? (eeval (car (cdr e)) ns vs p) 'T)
          (eeval (car (cdr (cdr e))) ns vs p)
          (eeval (car (cdr (cdr (cdr e)))) ns vs p))
  (if (equal? (car e) 5)   ; ==
      (if (equal? (eeval (car (cdr e)) ns vs p)
                  (eeval (car (cdr (cdr e))) ns vs p))
          'T
          'F)
                         ; call
  (let* ((ef (lookbody (car (cdr e)) p))
         (nf (lookname (car (cdr e)) p))
         (v  (eeval (car (cdr (cdr e))) ns vs p)))
    (eeval ef (cons nf '()) (cons v '()) p ))))))))

(define (lookvar x ns vs)
  (if (equal? x (car ns)) (car vs) (lookvar x (cdr ns) (cdr vs))))

(define (lookbody f p)
  (if (equal? (car (car p)) f)
      (car (cdr (cdr (car p))))
      (lookbody f (cdr p))))

(define (lookname f p)
  (if (equal? (car (car p)) f)
      (car (cdr (car p)))
      (lookname f (cdr p))))

(define (apply op v1 v2)
  (if (equal? op 5); equal
      (if (equal? v1 v2) 'T 'F)
                    ; cons
      (cons v1 v2)))
```

WHILE interpreter with dynamic scoping (int-dynscope)

```
;; Small 1st order interpreter with dynamic scoping
```

```
(define (run p input)
  (let* ((f0 (car (car p)))
         (ef (lookbody f0 p))
         (nf (lookname f0 p)))
    (eeval ef (cons nf '())( cons input '()) p)))

(define (eeval e ns vs p)
  (if (equal? (car e ) 1); constants
      (cdr e)
  (if (equal? (car e ) 2); variable
      (lookvar (cdr e) ns vs)
  (if (equal? (car e ) 3); basefcn
      (let* ((v1 (eeval (car (cdr (cdr e ))) ns vs p))
             (v2 (eeval (car (cdr (cdr (cdr e )))) ns vs p)))
        (apply (car (cdr e )) v1 v2))
  (if (equal? (car e ) 4)  ; if
      (if (equal? (eeval (car (cdr e )) ns vs p ) 'T)
          (eeval (car (cdr (cdr e ))) ns vs p)
          (eeval (car (cdr (cdr (cdr e )))) ns vs p))
  (if (equal? (car e ) 5)  ;==
      (if (equal? (eeval (car (cdr e )) ns vs p)
                  (eeval (car (cdr (cdr e ))) ns vs p))
          'T
          'F)
                         ; call
  (let* (( ef (lookbody (car (cdr e )) p))
         (nf (lookname (car (cdr e )) p))
         (v  (eeval (car (cdr (cdr e ))) ns vs p)))
    (eeval ef (cons nf ns ) (cons v vs ) p ))))))))

(define (lookvar x ns vs)
  (if (equal? x (car ns ))( car vs )( lookvar x (cdr ns ) (cdr vs ))))

(define (lookbody f p)
  (if (equal? (car (car p )) f)
      (car (cdr (cdr (car p ))))
      (lookbody f (cdr p ))))

(define (lookname f p)
  (if (equal? (car (car p)) f)
      (car (cdr (car p)))
      (lookname f (cdr p))))

(define (apply op v1 v2)
  (if (equal? op 5) ; equal
      (if (equal? v1 v2) 'T 'F)
                    ; cons
      (cons v1 v2)))
```

Appendix C

Detailed Results

C.1 Termination analysis

```
========================================
Program Examples/contrived−1.term ...

Bounded variables:
contrived_1 : a b

f : x y z d

g : u v w

h : r s

dec : n

Program terminates.
========================================
Program Examples/contrived−2.term ...

Bounded variables:
contrived_2 : a b
```

```
f : x y z d

g : u v w

h : r s => function may cause non-termination

dec : n

Program quasi-terminates.
========================================
Program Examples/list.term ...

Bounded variables:
goal : x

listQ : xs

isNull : x

isPair : x

Program terminates.
========================================
Program Examples/fold.term ...

Bounded variables:
fold : a xs

foldl : a xs

foldr : a xs

op : x1 x2

isPair : xs

Program terminates.
========================================
Program Examples/map.term ...

Bounded variables:
goal : xs

map : xs

f : x

Program terminates.
========================================
Program Examples/naiverev.term ...

Bounded variables:
goal : xs

naiverev : xs

app : xs ys

Program terminates.
========================================
Program Examples/deeprev.term ...

Bounded variables:
goal : x

deeprev : x

deeprevapp : xs rest

revconsapp : xs r

isPair : x

Program terminates.
========================================
Program Examples/append.term ...
```

```
Bounded variables:
goal : x y

append : xs ys

Program terminates.
========================================
Program Examples/mergelists.term ...

Bounded variables:
goal : xs ys

merge : xs ys

Program terminates.
========================================
Program Examples/addlists.term ...

Bounded variables:
goal : xs ys

 addlist : xs ys

 isPair : xs

Program terminates.
========================================
Program Examples/revapp.term ...

Bounded variables:
goal : x y

revapp : xs rest

Program terminates.
========================================
Program Examples/permute.term ...

Bounded variables:
```

```
goal  :  xs

permute : =>  function  may cause non−termination

 select   : =>  function  may cause non−termination

mapconsapp : => function  may cause non−termination

revapp  : =>  function  may cause non−termination

Program may not terminate!
========================================
Program Examples/add.term ...

Bounded variables :
goal  :  x y

add  :  x y

Program terminates .
========================================
Program Examples/badd.term ...

Bounded variables :
goal  :  x y

badd  :  x => function  may cause non−termination

Program may not terminate!
========================================
Program Examples/mul.term ...

Bounded variables :
goal  :  x y

mul  : x y

add  :  x y
```

```
Program terminates.
========================================
Program Examples/disjconj.term ...

Bounded variables:
 disjconj : p

disjQ : p

conjQ : p

boolQ : p

 isPair : xs

Program terminates.
========================================
Program Examples/duplicate.term ...

Bounded variables:
goal : x

 duplicate : xs

Program terminates.
========================================
Program Examples/nestimeql.term ...

Bounded variables:
goal : x

 nestimeql : => function may cause non−termination

immatcopy : => function may cause non−termination

Program may not terminate!
========================================
Program Examples/evenodd.term ...
```

```
Bounded variables:
evenodd : x

evenQ : x

oddQ : x

 isPair  : xs

 isNull  : xs

Program terminates.
========================================
Program Examples/lte.term ...

Bounded variables:
goal : x y

lteQ : x y

evenQ : x

 isPair  : xs

 isNull  : xs

Program terminates.
========================================
Program Examples/member.term ...

Bounded variables:
goal : x xs

memberQ : x xs

Program terminates.
========================================
Program Examples/ordered.term ...
```

```
Bounded variables:
goal : xs

orderedQ : xs

 isPair : xs

Program terminates.
========================================
Program Examples/overlap.term ...

Bounded variables:
goal : xs ys

overlapQ : xs ys

memberQ : x xs

 isPair : xs

Program terminates.
========================================
Program Examples/select.term ...

Bounded variables:
 select : xs

 selects : x revprefix postfix

revapp : xs rest

Program terminates.
========================================
Program Examples/subsets.term ...

Bounded variables:
goal : xs

 subsets : xs
```

```
mapconsapp : x  xss   rest

 isPair  :  x

Program terminates.
==========================================
Program Examples/anchored.term  ...

Bounded  variables :
goal  :  x  y

anchored  :  x  y

Program terminates.
==========================================
Program Examples/letexp.term  ...

Bounded  variables :
goal  :  x  y

 letexp  :  y => function  may cause  non—termination

Program may not terminate!
==========================================
Program Examples/thetrick.term  ...

Bounded  variables :
goal  :  x  y

f  :  x => function  may cause  non—termination

g  :  x  y

 lt  :  x  y

Program may not terminate!
==========================================
Program Examples/intlookup.term  ...
```

```
Bounded variables :
run : e p

 intlookup : e p => function may cause non−termination

lookup : fnum p

Program quasi−terminates.
========================================
Program Examples/nolexicord.term ...

Bounded variables :
goal : a1 b1 a2 b2 a3 b3

 nolexicord : a1 b1 a2 b2 a3 b3

Program terminates.
========================================
Program Examples/decrease.term ...

Bounded variables :
goal : x

 decrease : x

Program terminates.
========================================
Program Examples/equal.term ...

Bounded variables :
goal : x

equal : x => function may cause non−termination

Program quasi−terminates.
========================================
Program Examples/increase.term ...
```

```
Bounded variables :
goal  :  x

 increase   : =>  function  may cause non−termination

Program may not terminate!
========================================
Program Examples/nestdec.term  ...

Bounded variables :
goal  :  x

nestdec  :  x

dec  :  x

Program terminates .
========================================
Program Examples/nesteql.term  ...

Bounded variables :
goal  :  x

 nesteql   :  x => function  may cause non−termination

eql  :  x => function  may cause non−termination

Program quasi−terminates.
========================================
Program Examples/nestinc.term  ...

Bounded variables :
goal  :  x

nestinc  : =>  function  may cause non−termination

inc  : =>  function  may cause non−termination

Program may not terminate!
```

```
========================================
Program Examples/sp1.term ...

Bounded variables:
sp1 : x y

f : x y => function may cause non−termination

g : x y => function may cause non−termination

h : x y => function may cause non−termination

r : x y

Program quasi−terminates.
========================================
Program Examples/shuffle.term ...

Bounded variables:
goal : xs

 shuffle : => function may cause non−termination

 reverse : => function may cause non−termination

append : => function may cause non−termination

Program may not terminate!
========================================
Program Examples/assrewrite.term ...

Bounded variables:
 assrewrite : exp

 rewrite : => function may cause non−termination

 isPair :

Program may not terminate!
```

```
========================================
Program Examples/game.term ...

Bounded variables:
goal  :  p1 p2 moves

game : p1 p2 moves

Program terminates.
========================================
Program Examples/vangelder.term  ...

Bounded variables:
goal  :  x y

e  :  a  b

q  :  x y => function  may cause non—termination

p  :  x y => function  may cause non—termination

r  :  x y => function  may cause non—termination

t  :  x y => function  may cause non—termination

 isPair  :  xs

Program quasi—terminates.
========================================
Program Examples/power.term ...

Bounded variables:
goal  :  x n

power  :  x n

mult  :  x y

add  :  x y
```

```
Program terminates.
==============================================
Program Examples/binom.term ...

Bounded variables:
goal  :  n k

binom : n k

Program terminates.
==============================================
Program Examples/ack.term ...

Bounded variables:
goal  :  m n

ack  :  m n

Program terminates.
==============================================
Program Examples/gcd−1.term ...

Bounded variables:
goal  :  x y

gcd  :  x y

gt  :  x y

monus : x y

lgth  :  x

Program terminates.
==============================================
Program Examples/gcd−2.term ...

Bounded variables:
```

```
goal : x y

gcd : x y => function may cause non−termination

gt : x y

monus : x y

lgth : x

Program quasi−terminates.
========================================
Program Examples/mergesort.term ...

Bounded variables :
goal : xs

mergesort : => function may cause non−termination

 splitmerge : => function may cause non−termination

merge : => function may cause non−termination

 isPair :

Program may not terminate!
========================================
Program Examples/quicksort.term ...

Bounded variables :
goal : xs

 quicksort : => function may cause non−termination

part : => function may cause non−termination

app : => function may cause non−termination

 isPair :
```

```
Program may not terminate!
========================================
Program Examples/minsort.term ...

Bounded variables:
goal : xs

minsort : => function may cause non−termination

appmin : => function may cause non−termination

remove : => function may cause non−termination

 isPair :

Program may not terminate!
========================================
Program Examples/reach.term ...

Bounded variables:
goal : u v edges

reach : u v edges => function may cause non−termination

via : u v rest edges => function may cause non−termination

memberQ : x xs

Program quasi−terminates.
========================================
Program Examples/graphcolour−1.term ...

Bounded variables:
graphcolour : g cs

colornode : cs node colorednodes

 possible : color adjs colorednodes
 colorof  :  node  colorednodes

 colorrest   :  cs  ncs  colorednodes   rest  => function  may cause non−termination

  colorrest_thetrick   :  cs1  cs  ncs  colorednodes   rest
                          => function  may cause non−termination

 reverse  :  xs

revapp  :  xs  rest

 isPair  :  x

Program quasi−terminates.
========================================
Program Examples/graphcolour−2.term ...

Bounded variables :
graphcolour  :  g  cs

 colornode  :  cs  node  colorednodes

 possible  :  color  adjs  colorednodes

 colorof  :  node  colorednodes

 colorrest   :  cs  ncs  colorednodes   rest  => function  may cause non−termination

 colornoderest  :  cs  ncs  node  colorednodes   rest
                   => function  may cause non−termination

  colorrest_thetrick   :  cs1  cs  ncs  colorednodes   rest
                          => function  may cause non−termination

 reverse  :  xs

revapp  :  xs  rest
```

```
 isPair  :  x

Program quasi−terminates.
========================================
Program Examples/graphcolour−3.term ...

Bounded variables :
graphcolour  :  g  cs

 colornode  :  cs  node  colorednodes

 possible   :  color   adjs   colorednodes

 colorof  :  node  colorednodes

 colorrest   :  cs  ncs  colorednodes   rest

 colornoderest   :  cs  ncs  node  colorednodes   rest

 reverse  :  xs

revapp  :  xs   rest

 isPair  :  x

Program terminates.
========================================
Program Examples/match.term ...

Bounded variables :
match  :  p  s

loop  :  p  s  pp  ss

Program terminates.
========================================
Program Examples/strmatch.term ...

Bounded variables :
```

```
strmatch : patstr str

domatch : patcs cs n

 prefix : precs cs

 string2list : x

 patstring2list : x

 isPair : xs

Program terminates.
========================================
Program Examples/turing.term ...

Bounded variables:
run : prog tapeinput

turing : instrs prog => function may cause non−termination

lookup : i instrs

 isPair :

Program may not terminate!
========================================
Program Examples/lambdaint.term ...

Bounded variables:
lambdaint : e

red : => function may cause non−termination

subst : => function may cause non−termination

isvarQ :

islamQ :
```

```
mklam :

lam2var :

lam2body :

mkapp :

app2e1 :

app2e2 :

Program may not terminate!
========================================
Program Examples/int-loop.term ...

Bounded variables:
run : p l input

eeval : e ns vs l p

lookvar : x ns vs

lookbody : f p

lookname : f p

apply : op v1 v2

Program terminates.
========================================
Program Examples/int-while.term ...

Bounded variables:
run : p input

eeval : e ns p => function may cause non-termination
```

```
lookvar  :  x  ns

lookbody  :  f  p

lookname  :  f  p

apply  :  op

Program may not terminate!
========================================
Program Examples/int−dynscope.term ...

Bounded variables :
run  :  p  input

eeval  :  e  p => function  may cause non−termination

lookvar  :  x => function  may cause non−termination

lookbody  :  f  p

lookname  :  f  p

apply  :  op

Program may not terminate!
```

C.2 Binding-time analysis

```
========================================
Program Examples/contrived−1.term ...
Result  of  BTA:
contrived_1  :  a:s  b:d

f  :  x:s  y:s  z:s  d:d

g  :  u:s  v:s  w:d
```

```
h : r:s s:s

dec : n:s


No variables made dynamic to ensure termination
=============================================
Program Examples/contrived−2.term ...
Result of BTA:
contrived_2 : a:s b:d

f : x:s y:s z:s d:d

g : u:s v:s w:d

h : r:s s:s   insert SP

dec : n:s


No variables made dynamic to ensure termination
=============================================
Program Examples/list.term ...
Result of BTA:
goal : x:s

listQ : xs:s

isNull : x:s

isPair : x:s


No variables made dynamic to ensure termination
=============================================
Program Examples/fold.term ...
Result of BTA:
fold : a:d xs:s
```

```
foldl  : a:d xs:s

foldr  : a:d xs:s

op : x1:d x2:d

isPair  : xs:s


No variables  made dynamic to ensure  termination
========================================
Program Examples/fold.term  ...
Result of BTA:
fold  : a:s xs:d

foldl  : a:d xs:d   insert SP

foldr  : a:s xs:d   insert SP

op : x1:d x2:d

isPair  : xs:d


No variables  made dynamic to ensure  termination
========================================
Program Examples/map.term ...
Result of BTA:
goal  : xs:s

map : xs:s

f  : x:s


No variables  made dynamic to ensure  termination
========================================
Program Examples/naiverev.term  ...
Result of BTA:
```

```
goal : xs:s

naiverev : xs:s

app : xs:s ys:s


No variables made dynamic to ensure termination
========================================
Program Examples/deeprev.term ...
Result of BTA:
goal : x:s

deeprev : x:s

deeprevapp : xs:s rest:s

revconsapp : xs:s r:s

isPair : x:s


No variables made dynamic to ensure termination
========================================
Program Examples/append.term ...
Result of BTA:
goal : x:s y:d

append : xs:s ys:d


No variables made dynamic to ensure termination
========================================
Program Examples/mergelists.term ...
Result of BTA:
goal : xs:s ys:d

merge : xs:s ys:d   insert SP
```

```
No variables made dynamic to ensure termination
=========================================
Program Examples/addlists.term ...
Result of BTA:
goal : xs:s ys:d

 addlist : xs:s ys:d

 isPair : xs:s


No variables made dynamic to ensure termination
=========================================
Program Examples/revapp.term ...
Result of BTA:
goal : x:s y:d

revapp : xs:s rest :d


No variables made dynamic to ensure termination
=========================================
Program Examples/permute.term ...
Result of BTA:
goal : xs:s

permute : xs:d  insert SP

 select : x:d revprefix :d postfix :d  insert SP

mapconsapp : x:d xss:d rest :d  insert SP

revapp : xs:d rest :d  insert SP


 Variables permute_1 select_1  select_2  select_3  mapconsapp_1
           mapconsapp_2 mapconsapp_3 revapp_1 revapp_2
    made dynamic to ensure termination
```

```
=========================================
Program Examples/add.term ...
Result of BTA:
goal : x:s y:d

add : x:d y:d   insert SP


 Variable add_1 made dynamic to ensure termination
=========================================
Program Examples/add.term ...
Result of BTA:
goal : x:d y:s

add : x:d y:s


No variables made dynamic to ensure termination
=========================================
Program Examples/badd.term ...
Result of BTA:
goal : x:s y:d

badd : x:s y:d   insert SP


No variables made dynamic to ensure termination
=========================================
Program Examples/mul.term ...
Result of BTA:
goal : x:s y:d

mul : x:s y:d

add : x:d y:d   insert SP


No variables made dynamic to ensure termination
=========================================
Program Examples/disjconj.term ...
Result of BTA:
 disjconj : p:s

disjQ : p:s

conjQ : p:s

boolQ : p:s

 isPair : xs:s


No variables made dynamic to ensure termination
========================================
Program Examples/duplicate.term ...
Result of BTA:
goal : x:s

 duplicate : xs:s


No variables made dynamic to ensure termination
========================================
Program Examples/nestimeql.term ...
Result of BTA:
goal : x:s

 nestimeql : x:d  insert SP

immatcopy : x:d  insert SP


 Variables nestimeql_1 immatcopy_1 made dynamic to ensure termination
========================================
Program Examples/evenodd.term ...
Result of BTA:
evenodd : x:s
```

```
evenQ : x:s

oddQ : x:s

isPair : xs:s

isNull : xs:s


No variables made dynamic to ensure termination
=========================================
Program Examples/lte.term ...
Result of BTA:
goal : x:s y:d

lteQ : x:s y:d

evenQ : x:s

isPair : xs:d

isNull : xs:s


No variables made dynamic to ensure termination
=========================================
Program Examples/lte.term ...
Result of BTA:
goal : x:d y:s

lteQ : x:d y:s

evenQ : x:d   insert SP

isPair : xs:d

isNull : xs:d
```

```
No variables made dynamic to ensure termination
========================================
Program Examples/member.term ...
Result of BTA:
goal : x:d xs:s

memberQ : x:d xs:s


No variables made dynamic to ensure termination
========================================
Program Examples/member.term ...
Result of BTA:
goal : x:s xs:d

memberQ : x:s xs:d   insert SP


No variables made dynamic to ensure termination
========================================
Program Examples/ordered.term ...
Result of BTA:
goal : xs:s

orderedQ : xs:s

 isPair : xs:s


No variables made dynamic to ensure termination
========================================
Program Examples/overlap.term ...
Result of BTA:
goal : xs:s ys:d

overlapQ : xs:s ys:d

memberQ : x:s xs:d   insert SP
```

```
isPair : xs:d


No variables made dynamic to ensure termination
========================================
Program Examples/select.term ...
Result of BTA:
 select : xs:s

 selects : x:s revprefix :s postfix :s

revapp : xs:s rest :s


No variables made dynamic to ensure termination
========================================
Program Examples/subsets.term ...
Result of BTA:
goal : xs:s

subsets : xs:s

mapconsapp : x:s xss:s rest :s

isPair : x:s


No variables made dynamic to ensure termination
========================================
Program Examples/anchored.term ...
Result of BTA:
goal : x:s y:d

anchored : x:s y:d


No variables made dynamic to ensure termination
========================================
Program Examples/anchored.term ...
```

```
Result of BTA:
goal : x:d y:s

anchored : x:d y:d   insert SP


Variable anchored_2 made dynamic to ensure termination
========================================
Program Examples/letexp.term ...
Result of BTA:
goal : x:s y:s

letexp : x:d y:s   insert SP


Variable letexp_1 made dynamic to ensure termination
========================================
Program Examples/thetrick.term ...
Result of BTA:
goal : x:s y:d

f : x:s y:d   insert SP

g : x:s y:d   insert SP

lt : x:s y:s


No variables made dynamic to ensure termination
========================================
Program Examples/thetrick.term ...
Result of BTA:
goal : x:d y:s

f : x:d y:d   insert SP

g : x:d y:d   insert SP

lt : x:d y:s
```

```
Variable g_2 made dynamic to ensure termination
=========================================
Program Examples/intlookup.term ...
Result of BTA:
run : e:d p:s

intlookup : e:d p:s   insert SP

lookup : fnum:d p:s


No variables made dynamic to ensure termination
=========================================
Program Examples/nolexicord.term ...
Result of BTA:
goal : a1:s b1:s a2:s b2:s a3:s b3:s

nolexicord : a1:s b1:s a2:s b2:s a3:s b3:s


No variables made dynamic to ensure termination
=========================================
Program Examples/nolexicord.term ...
Result of BTA:
goal : a1:s b1:s a2:s b2:s a3:s b3:d

nolexicord : a1:s b1:s a2:s b2:s a3:d b3:d   insert SP


No variables made dynamic to ensure termination
=========================================
Program Examples/decrease.term ...
Result of BTA:
goal : x:s

decrease : x:s
```

```
No variables made dynamic to ensure termination
========================================
Program Examples/equal.term ...
Result of BTA:
goal : x:s

equal : x:s  insert SP


No variables made dynamic to ensure termination
========================================
Program Examples/increase.term ...
Result of BTA:
goal : x:s

 increase : x:d  insert SP


 Variable increase_1 made dynamic to ensure termination
========================================
Program Examples/nestdec.term ...
Result of BTA:
goal : x:s

nestdec : x:s

dec : x:s


No variables made dynamic to ensure termination
========================================
Program Examples/nesteql.term ...
Result of BTA:
goal : x:s

 nesteql : x:s  insert SP

eql : x:s  insert SP
```

```
No variables made dynamic to ensure termination
========================================
Program Examples/nestinc.term ...
Result of BTA:
goal : x:s

nestinc : x:d   insert SP

inc : x:d   insert SP


Variables nestinc_1 inc_1 made dynamic to ensure termination
========================================
Program Examples/sp1.term ...
Result of BTA:
sp1 : x:s y:d

f : x:s y:d   insert SP

g : x:s y:d   insert SP

h : x:s y:d   insert SP

r : x:s y:d


No variables made dynamic to ensure termination
========================================
Program Examples/shuffle.term ...
Result of BTA:
goal : xs:s

shuffle : xs:d   insert SP

reverse : xs:d   insert SP

append : xs:d ys:d   insert SP
```

```
 Variables   shuffle_1   reverse_1  append_1 append_2 made dynamic to ensure  termination
========================================
Program Examples/assrewrite.term  ...
Result  of  BTA:
 assrewrite   :  exp:s

 rewrite  :  exp:d    insert  SP

 isPair  :  xs:d


 Variables   rewrite_1   isPair_1  made dynamic to  ensure  termination
========================================
Program Examples/game.term ...
Result  of  BTA:
goal  :  p1:s  p2:s  moves:s

game : p1:s  p2:s  moves:s


No  variables   made dynamic to  ensure  termination
========================================
Program Examples/vangelder.term  ...
Result  of  BTA:
goal  :  x:s  y:d

e  :  a:s  b:d

q  :  x:s  y:d    insert  SP

p  :  x:s  y:d    insert  SP

r  :  x:s  y:d    insert  SP

t  :  x:s  y:d    insert  SP

 isPair  :  xs:d
```

```
No variables made dynamic to ensure termination
========================================
Program Examples/power.term ...
Result of BTA:
goal : x:d n:s

power : x:d n:s

mult : x:d y:d   insert SP

add : x:d y:d   insert SP


No variables made dynamic to ensure termination
========================================
Program Examples/binom.term ...
Result of BTA:
goal : n:s k:d

binom : n:s k:d


No variables made dynamic to ensure termination
========================================
Program Examples/binom.term ...
Result of BTA:
goal : n:d k:s

binom : n:d k:s   insert SP


No variables made dynamic to ensure termination
========================================
Program Examples/ack.term ...
Result of BTA:
goal : m:s n:d
```

```
ack : m:s n:d  insert SP


No variables made dynamic to ensure termination
========================================
Program Examples/gcd−1.term ...
Result of BTA:
goal : x:s y:d

gcd : x:d y:d  insert SP

gt : x:d y:d  insert SP

monus : x:d y:d  insert SP

lgth : x:d  insert SP


No variables made dynamic to ensure termination
========================================
Program Examples/gcd−2.term ...
Result of BTA:
goal : x:s y:d

gcd : x:d y:d  insert SP

gt : x:d y:d  insert SP

monus : x:d y:d  insert SP

lgth : x:d  insert SP


No variables made dynamic to ensure termination
========================================
Program Examples/mergesort.term ...
Result of BTA:
goal : xs:s
```

```
mergesort : xs:d   insert SP

splitmerge : xs:d xs1:d xs2:d   insert SP

merge : xs1:d xs2:d   insert SP

isPair : xs:d


Variables mergesort_1 splitmerge_1 splitmerge_2 splitmerge_3
          merge_1 merge_2 isPair_1 made dynamic to ensure termination
=======================================
Program Examples/quicksort.term ...
Result of BTA:
goal : xs:s

quicksort : xs:d   insert SP

part : x:d xs:d xs1:d xs2:d   insert SP

app : xs:d ys:d   insert SP

isPair : xs:d


Variables quicksort_1 part_1 part_2 part_3 part_4 app_1 app_2
          isPair_1 made dynamic to ensure termination
=======================================
Program Examples/minsort.term ...
Result of BTA:
goal : xs:s

minsort : xs:d   insert SP

appmin : min:d rest:d xs:d   insert SP

remove : x:d xs:d   insert SP

isPair : xs:d
```

```
 Variables  minsort_1 appmin_1 appmin_2 appmin_3 remove_1 remove_2
            isPair_1  made dynamic to ensure  termination
=========================================
Program Examples/reach.term  ...
Result  of  BTA:
goal  :  u:d  v:d  edges:s

reach  :  u:d  v:d  edges:s    insert  SP

via  :  u:d  v:d  rest:s  edges:s    insert  SP

memberQ : x:d  xs:s


No  variables  made dynamic to ensure  termination
=========================================
Program Examples/reach.term  ...
Result  of  BTA:
goal  :  u:s  v:d  edges:s

reach  :  u:s  v:d  edges:s    insert  SP

via  :  u:s  v:d  rest:s  edges:s    insert  SP

memberQ : x:d  xs:s


No  variables  made dynamic to ensure  termination
=========================================
Program Examples/graphcolour−1.term ...
Result  of  BTA:
graphcolour  :  g:d  cs:s

colornode  :  cs:s  node:d  colorednodes:d

 possible  :  color:s  adjs:d  colorednodes:d    insert  SP
 colorof  :  node:d  colorednodes :d    insert  SP

 colorrest   :  cs :s  ncs:s  colorednodes :d  rest :d    insert  SP

  colorrest_thetrick    :  cs1 :s  cs :s  ncs:d  colorednodes :d  rest :d    insert  SP

 reverse  :  xs:d

revapp  :  xs :d  rest :d    insert  SP

 isPair  :  x:d


No  variables  made dynamic to ensure  termination
========================================
Program Examples/graphcolour−1.term ...
Result  of  BTA:
graphcolour  :  g:s  cs :d

 colornode  :  cs :d  node:s  colorednodes :d    insert  SP

 possible   :  color :d  adjs :s  colorednodes :d

 colorof  :  node:s  colorednodes :d    insert  SP

 colorrest   :  cs :d  ncs:d  colorednodes :d  rest :s    insert  SP

  colorrest_thetrick    :  cs1 :d  cs :d  ncs:d  colorednodes :d  rest :s    insert  SP

 reverse  :  xs:d

revapp  :  xs :d  rest :d    insert  SP

 isPair  :  x:d


No  variables  made dynamic to ensure  termination
========================================
Program Examples/graphcolour−2.term ...
```

```
Result of BTA:
graphcolour : g:d cs:s

colornode : cs:s node:d colorednodes:s

possible : color:s adjs:d colorednodes:d   insert SP

colorof : node:d colorednodes:d   insert SP

colorrest : cs:s ncs:s colorednodes:d rest:d   insert SP

colornoderest : cs:s ncs:s node:d colorednodes:d rest:d   insert SP

colorrest_thetrick : cs1:s cs:s ncs:s colorednodes:d rest:d   insert SP

reverse : xs:d

revapp : xs:d rest:d   insert SP

isPair : x:d


No variables made dynamic to ensure termination
========================================
Program Examples/graphcolour−2.term ...
Result of BTA:
graphcolour : g:s cs:d

colornode : cs:d node:s colorednodes:s   insert SP

possible : color:d adjs:s colorednodes:d

colorof : node:s colorednodes:d   insert SP

colorrest : cs:d ncs:d colorednodes:d rest:s   insert SP

colornoderest : cs:d ncs:d node:s colorednodes:d rest:s   insert SP

colorrest_thetrick : cs1:d cs:d ncs:d colorednodes:d rest:s   insert SP
```

```
reverse  :  xs:d

revapp  :  xs:d  rest :d   insert  SP

isPair  :  x:d


No  variables  made dynamic to ensure  termination
========================================
Program Examples/graphcolour−3.term ...
Result  of  BTA:
graphcolour  :  g:d  cs:s

colornode  :  cs:s  node:d  colorednodes :s

possible  :  color :s  adjs :d  colorednodes :d   insert  SP

colorof  :  node:d  colorednodes :d   insert  SP

colorrest  :  cs:s  ncs:s  colorednodes :d  rest :d   insert  SP

colornoderest  :  cs:s  ncs:s  node:d  colorednodes :d  rest :d   insert  SP

reverse  :  xs:d

revapp  :  xs:d  rest :d   insert  SP

isPair  :  x:d


No  variables  made dynamic to ensure  termination
========================================
Program Examples/graphcolour−3.term ...
Result  of  BTA:
graphcolour  :  g:s  cs:d

colornode  :  cs:d  node:s  colorednodes :s   insert  SP
```

```
 possible  :  color :d  adjs :s  colorednodes :d

 colorof  :  node:s  colorednodes :d   insert  SP

 colorrest  :  cs :d  ncs:d  colorednodes :d  rest :s   insert  SP

 colornoderest  :  cs :d  ncs:d  node:s  colorednodes :d  rest :s   insert  SP

 reverse  :  xs:d

revapp  :  xs:d  rest :d   insert  SP

 isPair  :  x:d


No  variables  made dynamic to ensure  termination
========================================
Program Examples/match.term ...
Result  of  BTA:
match  :  p:s  s :d

loop  :  p:s  s :d  pp:s  ss :d   insert  SP


No  variables  made dynamic to ensure  termination
========================================
Program Examples/strmatch.term ...
Result  of  BTA:
strmatch  :  patstr :s  str :d

domatch  :  patcs :s  cs :d  n:d   insert  SP

 prefix  :  precs :s  cs :d

  string2list  :  x:d

  patstring2list  :  x:s

 isPair  :  xs:d
 Variable  domatch_3 made dynamic to ensure  termination
========================================
Program Examples/strmatch.term ...
Result  of  BTA:
strmatch  :  patstr :d  str :s

domatch  :  patcs :d  cs :s  n:s

 prefix   :  precs :d  cs :s

  string2list   :  x:s

  patstring2list   :  x:d

 isPair  :  xs :d


No  variables  made dynamic to ensure  termination
========================================
Program Examples/turing.term ...
Result  of  BTA:
run  :  prog:s  tapeinput :d

turing  :  instrs :s  revltape :d  rtape :d  prog:s    insert  SP

lookup  :  i :s  instrs :s

 isPair  :  xs :d


No  variables  made dynamic to ensure  termination
========================================
Program Examples/lambdaint.term ...
Result  of  BTA:
lambdaint  :  e:s

red  :  e:d    insert  SP
```

```
subst : x:d a:d e:d   insert SP

isvarQ : e:d

islamQ : e:d

mklam : n:d e:d

lam2var : e:d

lam2body : e:d

mkapp : e1:d e2:d

app2e1 : e:d

app2e2 : e:d


 Variables red_1 subst_1 subst_2 subst_3 isvarQ_1 islamQ_1 mklam_1
           mklam_2 lam2var_1 lam2body_1 mkapp_1 mkapp_2 app2e1_1
           app2e2_1 made dynamic to ensure termination
========================================
Program Examples/int−loop.term ...
Result of BTA:
run : p:s l:s input:d

eeval : e:s ns:s vs:d l:s p:s

lookvar : x:s ns:s vs:d

lookbody : f:s p:s

lookname : f:s p:s

apply : op:s v1:d v2:d
```

```
No variables made dynamic to ensure termination
========================================
Program Examples/int-loop.term ...
Result of BTA:
run : p:s l:d input:d

eeval : e:s ns:s vs:d l:d p:s   insert SP

lookvar : x:s ns:s vs:d

lookbody : f:s p:s

lookname : f:s p:s

apply : op:s v1:d v2:d


No variables made dynamic to ensure termination
========================================
Program Examples/int-while.term ...
Result of BTA:
run : p:s input:d

eeval : e:s ns:s vs:d p:s   insert SP

lookvar : x:s ns:s vs:d

lookbody : f:s p:s

lookname : f:s p:s

apply : op:s v1:d v2:d


No variables made dynamic to ensure termination
========================================
Program Examples/int-dynscope.term ...
Result of BTA:
run : p:s input:d
```

eeval : e:s ns:d vs:d p:s insert SP

lookvar : x:s ns:d vs:d insert SP

lookbody : f:s p:s

lookname : f:s p:s

apply : op:s v1:d v2:d

Variables eeval_2 lookvar_2 made dynamic to ensure termination