Termination Analysis and Specialization-Point Insertion in Off-line Partial Evaluation

ARNE JOHN GLENSTRUP and NEIL D. JONES
DIKU, University of Copenhagen

Recent research suggests that the goal of fully automatic and reliable program generation for a broad range of applications is coming nearer to feasibility. However, several interesting and challenging problems remain to be solved before it becomes a reality. Solving them is also *necessary,* if we hope ever to elevate software engineering from its current state (a highly-developed handiwork) into a successful branch of engineering, capable of solving a wide range of new problems by systematic, well-automated and well-founded methods.

A key problem in all program generation is *termination* of the generation process. This paper focuses on off-line partial evaluation and describes recent progress towards automatically solving the termination problem, first for individual programs, and then for specializers and "generating extensions," the program generators that most offline partial evaluators produce.

The technique is based on *size-change graphs* that approximate the changes in parameter sizes at function calls. We formulate a criterion, *bounded anchoring,* for detecting parameters known to be bounded during specialization: a bounded parameter can act as an *anchor* for other parameters. Specialization points necessary for termination are computed by adding a parameter that tracks call depth, and then selecting a specialization point in every call loop where it is unanchored. By generalizing all unbounded parameters, we compute a binding-time division which together with the set of specialization points *guarantees* termination.

Contributions of this paper include a proof, based on the operational semantics of partial evaluation with memoization, that the analysis guarantees termination; and an in-depth description of safety of the increasing size approximation operator required for termination analysis in partial evaluation.

Initial experiments with a prototype shows that the analysis overall yields binding-time divisions that can achieve a high degree of specialization, while still guaranteeing termination.

The paper ends with a list of challenging problems whose solution would bring the community closer to the goal of broad-spectrum, fully automatic and reliable program generation.

Categories and Subject Descriptors: F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*partial evaluation*; *program analysis*; F.3.3 [**Logics and Meanings of Programs**]: Studies of Program Constructs—*program and recursion schemes*; I.2.2 [**Artificial Intelligence**]: Automatic Programming—*program transformation*; D.1.2 [**Programming Techniques**]: Automatic Programming; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*specification techniques*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*recursion*; D.3.4 [**Programming Languages**]: Processors—*optimization*; *compilers*; *interpreters*

General Terms: Algorithms, experimentation, theory

Additional Key Words and Phrases: Binding-time analysis, quasitermination, size-change graphs, termination

This work was partly supported by the Danish Narural Science Research Council.
Authors' address: DIKU, Universitetsparken 1, DK–2100 Copenhagen Ø, Denmark
Permission to make digital/hard copy of all or part of this material without fee for personal or classroom use provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.
© 2003 ACM ????-????/??/????-???? $00.75

1. ON PROGRAM GENERATION

Program generation is a rather old idea, dating back to the 1960s and seen in many forms since then. Instances include code templates, macro expansion, conditional assembly and the use of if-defs, report generators, partial evaluation/program specialization [Consel and Danvy 1993; Jones et al. 1993; Lloyd and Shepherdson 1991], domain-specific languages (at least, those compiled to an implementation language) [Hudak 1996], program transformation [Burstall and Darlington 1977; Chin et al. 1998; Gallagher and Bruynooghe 1990], and both practical and theoretical work aimed at generating programs from specifications [Hudak 1996; Kieburtz et al. 1996].

Recent years have seen a rapid growth of interest in generative, automatic approaches to program management, updating, adaptation, transformation, and evolution, e.g., [Kieburtz et al. 1996; McNamee et al. 2001; Taha 2000]. The motivations are well-known: persistence of the "software crisis" and dreams of achieving industrial-style automation, automatic adaptation of programs to new contexts, and automatic optimization.

The overall goal is to increase efficiency of software production by making it possible to write fewer but more abstract or more heavily parameterized programs. Ideally, one would like to transform a problem specification automatically into a solution in program form. Benefits of such an approach would include increased reliability in software production: The software developer can debug/test/verify a small number of high-level, compact programs or specifications; and then generate from these as many machine-near, efficiently executable, problem-specific programs as needed, all guaranteed to be faithful to the specification from which they were derived.

1.1 What are programs generated from?

On a small scale, programs can be generated from *problem parameters* (e.g., device characteristics determine the details of a device driver). Ideally, and on a larger scale, programs could be generated from *user-oriented problem descriptions or specifications.* Dreams of systematically transforming specifications into solutions were formulated in the 1970s by Dijkstra, Gries, Hoare, Manna, etc.

This approach has been tried out in real-world practice and some serious problems have been encountered. First, it is *very hard* to see whether a program actually solves the problem posed by a description or specification. Second, it seems impossible for specifications of manageable size to be complete enough to give the "whole picture" of what a program should do for realistic problem contexts. While there have been recent and impressive advances in bridging some gaps between specifications and programs by applying model-checking ideas to software [Corbett et al. 2000], they are incomplete: the specifications used nearly always describe only certain misbehaviors that cannot be accepted, rather than total program correctness.

Executable specifications. It has proven to be quite difficult to build a program from an unexecutable, nonalgorithmic specification, e.g., expressed in first-order or temporal logic. For realistic problems it is often much more practical to use an *executable specification.* Such a specification is, in essence, also a program; but is written on a higher level of abstraction than in an implementation language, and usually has many "don't-care" cases.

Consequence: Many examples of "generative software engineering" or "transforming specifications into programs" or "program generation" can be thought of as transformation from one high-level program (specification-oriented) to another lower-level one (execution-oriented). In other words, much of generative software engineering can be done by (variations on) *compilation* or *program transformation*.

This theme has been seen frequently, first in early compilers, then in use of program transformation frameworks for optimizations, then in partial evaluation, and more recently in multi-stage programming languages.

Partial evaluation for program generation. This paper concerns some improvements needed in order to use partial evaluation as a technology for automatically transforming an executable problem specification into an efficient stand-alone solution in program form. There have been noteworthy successes and an enormous literature in partial evaluation: see [Sestoft 2001] and the PEPM series, as well as the Asia-PEPM and SAIG conferences. However, its use for large-scale program generation has been hindered by the need, given an executable problem specification, to do "hand tuning" to ensure *termination of the program generation process*: that the partial evaluator will yield an output program for every problem instance described by the current problem specification.

1.2 Fully automatic program transformation: an impossible dream?

Significant speedups have been achieved by program transformation on a variety of interesting problems. Pioneers in the field include Bird, Boyle, Burstall, Darlington, Dijkstra, Gries, the Kestrel group, Meertens and Paige; more recent researchers include De Moor, Liu and Pettorossi. As a result of this work, some significant common principles for deriving efficient programs have become clear:

—Optimize by *changing the times* at which computations are performed (code motion, preprocessing, specialization, multi-staged programming languages [Aho et al. 1986; Consel and Danvy 1993; Ganz et al. 2001; Jones et al. 1993; Hatcliff et al. 1999; Taha 1999a; Taha et al. 2001; Taha and Sheard 2000]).

—Don't solve *the same sub-problem* repeatedly. A successful practical example is the XSB logic programming system [Sagonas et al. 1994], based on *memoization*: Instead of recomputing results, store them for future retrieval when first computed.

—Avoid *multiple passes* over same data structure (tupling, deforestation [Burstall and Darlington 1977; Chin et al. 1998; Glück and Sørensen 1996; Wadler 1988]).

—Use an abstract, *high-level specification language*. SETL [Cai et al. 1991] is a good example, with small mathematics-like specifications (sets, tuples, finite mappings, fixpoints) that are well-suited to automatic transformation and optimization of algorithms concerning graphs and databases.

The pioneering results already obtained in program transformation are excellent academic work that include the systematic reconstruction of many state-of-the-art fundamental algorithms seen in textbooks. Such established principles are being used to develop algorithms in newer fields, e.g., computational geometry and biocomputation. On the other hand, these methods seem ill-suited to large-scale, heterogeneous computations: tasks that are broad rather than deep.

Summing up. Program transformation is a promising field of research but its potential in practical applications is still far from being realized. Much of the earlier work has flavor of a "fine art," practiced by highly gifted researchers on one small program at a time. Significant automation has not yet been achieved for the powerful methods capable of yielding superlinear speedups. The less ambitious techniques of partial evaluation have been more completely automated, but even there, termination remains a challenging problem.

This problem setting defines the focus of this paper. For simplicity, in the remainder of this article we use a simple first-order functional language.

1.3 Requirements for success in generative software engineering

The major bottlenecks in generative software engineering have to do with humans. For example, most people cannot and should not have to understand automatically generated programs—e.g., to debug them—because they did not write them themselves. A well-known analogy: the typical user does not and probably cannot read the output of parser generators such as Yacc and Lex, or the code produced by any commercial compiler.

If a user is to trust programs he/she did not write, a *firm semantic basis* is needed, to ensure that user intentions match the behavior of the generated programs. Both intentions and behavior must be clearly and precisely defined. How can this be ensured?

First, the source language or specification language must have a precisely understood semantics (formal or informal; but tighter than, say, C++), so the user can know exactly what was specified. Second, evidence (proof, testing, etc.) is needed that the output of the program generator always has the same semantics as specified by its input. Third, familiar software quality demands must be satisfied, both by the program generator itself and the programs that it generates.

Desirable properties of a program generator thus include:

(1) *High automation level.* The process should
   —accept any well-formed input, i.e., not commit generation-time failures like "can't take tail of the empty list"
   —issue sensible error messages, so the user is not forced to read output code when something goes wrong
(2) The *code generation process* should
   —terminate for all inputs
   —be efficient enough for the usage context, e.g., the generation phase should be very fast for run-time code generation, but need not be for highly-optimizing compilation
(3) The *generated program*
   —should be efficient enough for the usage context, e.g., fast generated code in a highly-optimizing compilation context
   —should be of predictable complexity, e.g., not slower than executing the source specification

—should have an acceptable size (no code explosion)
—may be required to terminate.

1.4 On the meaning of "automatic" in program transformation

The goals above are central to the field of automatic program transformation. Unfortunately, this term seems to mean quite different things to different people, so we now list some variations as points on an "automation" spectrum:

(1) Hand work, e.g., program transformation done and proven correct on paper by gifted researchers. Examples: recursive function theory [Péter 1976], work by McCarthy and by Bird.
(2) Interactive tools for individual operations, e.g., call folding and unfolding. Examples: early theorem-proving systems, the Burstall-Darlington program transformation system [Burstall and Darlington 1977].
(3) Tools with automated strategies for applying transformation operations. Human interaction to check whether the transformation is "on target." Examples: Chin, Khoo, Liu [Chin et al. 1998; Liu 2000].
(4) Hand-placed program annotations to guide a transformation tool. Examples: Chambers' and Engeler's DyC and Tick C [Grant et al. 2000; Poletto et al. 1999]. After annotation, transformation is fully automatic, requiring no human interaction.
(5) Tools that automatically recognize and avoid the possibility of nontermination during program transformation, e.g., homeomorphic embedding. Examples: Sørensen, Glück, Leuschel [Leuschel 1998; Sørensen and Glück 1995].
(6) Computer-placed program annotations to guide transformation, e.g., binding-time annotation [Bondorf 1991; Consel 1993; Consel and Danvy 1993; Glenstrup and Jones 1996; Glück et al. 1995; Jones et al. 1993; Hatcliff et al. 1999; Thiemann 1997]. After annotation, transformation is fully automatic, requiring no human interaction.

In the following we use the term "automatic" mostly with the meaning of point 6. This paper's goal is automatically to perform the annotations that will ensure termination of program specialization. We will ignore issues concerning code duplication, elimination and reordering, as they are important issues in themselves treated elsewhere [Jones et al. 1993].

2. PARTIAL EVALUATION AND PROGRAM GENERATION

Partial evaluation [Consel and Danvy 1993; Danvy et al. 1996; Jones et al. 1993; Hatcliff et al. 1999; Lloyd and Shepherdson 1991; Mogensen 2000] is an example of automatic program transformation. While speedups are more limited than sometimes realizable by hand transformations based on deep problem or algorithmic knowledge, the technology is well-automated. It has already proven its utility in several different contexts:

—Scientific computing ([Berlin and Weise 1990; Glück et al. 1995], etc.);
—Extending functionality of existing languages by adding binding-time options ([Grant et al. 2000], etc.);

—Functional languages ([Bondorf 1991; Consel 1993; Hughes 1996; Launchbury 1991; Thiemann 1999], etc.);

—Logic programming languages ([Gallagher and Bruynooghe 1990; Gallagher 1993; Leuschel and Bruynooghe 2002; Lloyd and Shepherdson 1991], etc.);

—Compiling or other transformation by specializing interpreters ([Birkedal and Welinder 1994; Bondorf 1991; Glenstrup et al. 1999; Jones et al. 1993; Jones 1996; Mogensen 1988], etc.);

—Optimization of operating systems, e.g., remote procedure calls, device drivers, etc. ([Consel and Noël 1996; McNamee et al. 2001; Poletto et al. 1999], etc.);

2.1 Equational definition of a parser generator

To get started, and to connect partial evaluation with program generation, we first exemplify our notation for program runs on an example. (Readers familiar with the field may wish to skip to Section 3.)

Parser generation is a familiar example of program generation. The construction of a parser from a grammar by a parser generator, and running the generated parser, can be described by the following two program runs:[1]

```
parser      :=  ⟦parse-gen⟧ [grammar]
parsetree   :=  ⟦parser⟧ [inputstring]
```

The combined effect of the two runs can be described by a nested expression:

```
parsetree := ⟦⟦parse-gen⟧ [grammar]⟧ [inputstring]
```

2.2 What goals does partial evaluation achieve?

A partial evaluator [Consel and Danvy 1993; Jones et al. 1993; Hatcliff et al. 1999] is a *program specializer:* Suppose one is given a subject program p, expecting two inputs, and the first of its input data, in1 (the "static input"). The effect of specialization is to construct a new program p_{in1} that, when given p's remaining input in2 (the "dynamic input"), yields the same result that p would have produced if given both inputs.

The process of partial evaluation is shown schematically in Figure 1.

Example 1. The standard toy example for partial evaluation is the program power, computing x^n with code:

```
f(n,x) = if n=0 then 1 else if odd(n) then x*f(n-1,x) else f(n/2,x)**2
```

This example illustrates that partial evaluation in effect does an aggressive constant propagation across function calls. Specialization to static input n = 13 yields a residual program that runs several times faster than the general one above:

```
f_13(x) = x*((x*(x**2))**2)**2
```

[1]Notation: ⟦p⟧ [in1,...,ink] is a partial value: the result yielded by running program p on input values in1,...,ink if this computation terminates, else the *undefined* value $\perp$.

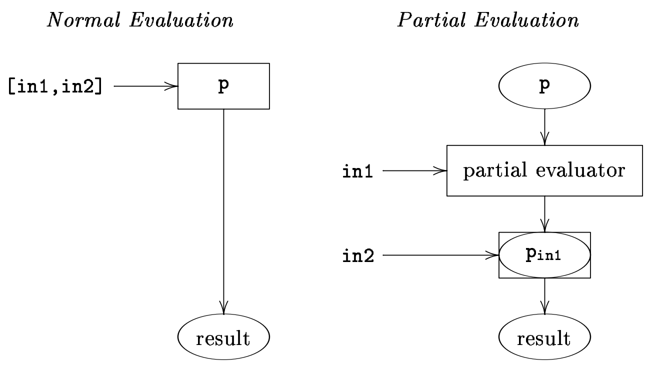


Fig. 1. Comparison of normal and partial evaluation. Boxes represent programs, ovals represent data objects. The *residual program* is $\mathtt{p}_{\mathtt{in1}}$.

Example 2. (Ackermann's function) If calls are not unfolded, then specialized functions are generated in the specialized program, as seen by this example of specialization to static input m=2:

Source program

```
ack(m,n) = if m=0 then n+1 else
    if n=0 then ack(m-1,1) else
            ack(m-1,ack(m,n-1))
```

Specialized program (for m = 2*)*

```
ack_2(n) = if n=0 then
         ack_1(1) else
     ack_1(ack_2(n-1))
ack_1(n) = if n=0 then
         ack_0(1) else
     ack_0(ack_1(n-1))
ack_0(n) = n+1
```

2.2.1 *Equational definition of a partial evaluator.* Correctness of $\mathtt{p}_{\mathtt{in1}}$ as described by Figure 1 can be described equationally as follows:[2]

Definition 2.2.1. A partial evaluator spec is a program such that for all programs p and input data in1, in2:

$$[\![\mathtt{p}]\!]\,[\mathtt{in1},\mathtt{in2}] = [\![\,[\![\mathtt{spec}]\!]\,[\mathtt{p},\mathtt{in1}]\,]\!]\,[\mathtt{in2}]$$

Writing $\mathtt{p}_{\mathtt{in1}}$ for $[\![\mathtt{spec}]\!]\,[\mathtt{p},\mathtt{in1}]$, the equation can be restated without nested semantic parentheses $[\![\,]\!]$:

$$[\![\mathtt{p}]\!]\,[\mathtt{in1},\mathtt{in2}] = [\![\mathtt{p}_{\mathtt{in1}}]\!]\,[\mathtt{in2}]$$

Program $\mathtt{p}_{\mathtt{in1}}$ is often called the "residual program" of p with respect to in1, as it specifies remaining computations that were not done at specialization time.

2.2.2 *Desirable properties of a partial evaluator.*

Efficiency. This is the main goal of partial evaluation. Run out:=$[\![\mathtt{p}]\!]$ [in1,in2] is often slower than the run out:=$[\![\mathtt{p}_{\mathtt{in1}}]\!]$ [in2], as seen in the two examples above.

[2]An equation $[\![\mathtt{p}]\!]\,[\mathtt{in1},\ldots,\mathtt{ink}] = [\![\mathtt{q}]\!]\,[\mathtt{in1},\ldots,\mathtt{ink}]$ expresses equality of partial values: if either side terminates, then the other side does too, and they yield the same value.

Said another way: It is slower to run the *general* program p on [in1,in2] than it is to run the *specialized* residual program $\mathtt{p}_{\mathtt{in1}}$ on in2.

The reason is that, while constructing $\mathtt{p}_{\mathtt{in1}}$, some of p's operations that depend only on in1 have been *precomputed*, and some function calls have been *unfolded* or "inlined." These actions are never done while running $[\![\mathtt{p}_{\mathtt{in1}}]\!]$ [in2]; but they will be performed *every time* $[\![\mathtt{p}]\!]$ [in1,in2] is run. This is especially relevant if p is run often, with in1 changing less frequently than in2.

Correctness. First and foremost, a partial evaluator should be correct. Today's state of the art is that most partial evaluators give correct residual programs when they terminate, but sometimes loop infinitely or sometimes give code explosion. A residual program, once constructed, will in general terminate at least as often as the program from which it was derived.

Completeness. Maximal speedup will be obtained if *every computation* depending only on in1 is performed. For instance the specialized Ackermann program above, while faster than the original, could be improved yet more by replacing the calls ack_1(1) and ack_0(1), respectively, by constants 3 and 2.

Termination. Ideally:

—The specializer should terminate for all inputs [p,in1].

—The generated program $\mathtt{p}_{\mathtt{in1}}$ should terminate on in2 just in case p terminates on [in1,in2].

These termination properties are easier to state than to achieve, as there is an intrinsic conflict between demands for completeness and termination of the specializer. Nonetheless, significant progress has been made in the past few years [Das 1998; Das and Reps 1996; Glenstrup 1999; Glenstrup and Jones 1996; Song and Futamura 2000]. A key is carefully to select which parts of the program should be computed at specialization time. (In the usual jargon: which parts are considered as "static.")

2.2.3 *A breakthrough: the generation of program generators.* For the first time in the mid 1980s, a breakthrough was achieved in practice that had been foreseen by Japanese and Russian researchers early in the 1970s: the automatic generation of compilers and other program generators by self-application of a specializer. The following definitions [Futamura 1999a; Futamura 1999b] are called the *Futamura projections:*

Generic Futamura projections

1.	$\mathtt{p}_{\mathtt{in1}}$	:=	$[\![\mathtt{spec}]\!]$ [p,in1]
2.	p-gen	:=	$[\![\mathtt{spec}]\!]$ [spec,p]
3.	cogen	:=	$[\![\mathtt{spec}]\!]$ [spec,spec]

Consequences:

A.	$[\![\mathtt{p}]\!]$ [in1,in2]	=	$[\![\mathtt{p}_{\mathtt{in1}}]\!]$ [in2]
B.	$\mathtt{p}_{\mathtt{in1}}$	=	$[\![\mathtt{p\text{-}gen}]\!]$ [in1]
C.	p-gen	=	$[\![\mathtt{cogen}]\!]$ [p]

Program p-gen is called p's "generating extension." By B it is a *generator of specialized versions of* p, that transforms static parameter in1 into specialized program

$\mathtt{p_{in1}}$. Further, by C and B the program called cogen behaves as a *generator of program generators.*

Consequence A is immediate from Definition 2.2.1. Proof of Consequence B is by simple algebra:

⟦p-gen⟧ [in1]	=	⟦⟦spec⟧ [spec,p]⟧ [in1]	By definition of p-gen
	=	⟦spec⟧ [p,in1]	By Definition 2.2.1
	=	$\mathtt{p_{in1}}$	By definition of $\mathtt{p_{in1}}$

and consequence C is proven by similar algebraic reasoning, left to the reader.

Efficiency gained by self-application of a partial evaluator. Compare the two ways to specialize program p to static input in1:

```
I.     p_in1  :=  ⟦spec⟧ [p,in1]
II.    p_in1  :=  ⟦p-gen⟧ [in1]
```

Way I is in practice usually several times slower than Way II. This is for exactly the same reason that $\mathtt{p_{in1}}$ is faster than p: Way I runs spec, which is a *general* program that is able to specialize *any program* p to any in1. On the other hand, Way II runs p-gen: a *specialized* program, only able to produce specialized versions of this particular p.

By exactly the same reasoning, computing p-gen:=⟦spec⟧ [spec,p] is often several times slower than using the compiler generator p-gen:=⟦cogen⟧ [p].

Writing cogen *instead of* spec. In a sense, any program generator is a “generating extension,” but without an explicitly given general source program p. To see this, consider the two runs of the parser generator example:

```
parser      :=  ⟦parse-gen⟧ [grammar]
parsetree   :=  ⟦parser⟧ [inputstring]
```

Here in principle a universal parser (e.g., Earley’s parser) satisfying:

```
parsetree = ⟦universal-parser⟧ [grammar, inputstring]
```

could have been used to obtain parse-gen as a generating extension:

```
parse-gen := ⟦cogen⟧ [universal-parser]
```

Further, modern specializers such as C-mix [Glenstrup et al. 1999], Tempo [Consel and Noël 1996], PGG [Thiemann 1999] and SML-mix [Birkedal and Welinder 1994] are cogen programs written directly, rather than built using self-application of spec as described above and in the literature [Jones et al. 1993]. (Descriptions of the idea can be found in [Birkedal and Welinder 1994; Holst and Launchbury 1991; Thiemann 1997].) Although not built by self-application, the net effect is the same: programs can be specialized; and a program may be converted into a generating extension with respect to its static parameter values.

2.2.4 *Front-end compilation: an important case of the Futamura projections.* Suppose that we now have *two* languages: the specializer’s input-output language

L, and another language S that is to be implemented. Let $[\![\_]\!]^S$ be the "semantic function" that assigns meanings to S-programs.

Definition 2.2.2. Program `interp` is an *interpreter* for language S written in language L if for any S-program `source` and input `in`,

$$[\![\texttt{source}]\!]^S[\texttt{in}] = [\![\texttt{interp}]\!][\texttt{source,in}]$$

(See Figure 2 in Section 2.2.5 for an example interpreter.) We now apply the Futamura projections with some renaming: Replace program `p` by `interp`, static input `in1` by S-program `source`, dynamic input `in2` by `in`, and `p-gen` by `compiler`.

Compilation by the Futamura projections

1.	`target`	:=	$[\![\texttt{spec}]\!]$ `[interp,source]`
2.	`compiler`	:=	$[\![\texttt{spec}]\!]$ `[spec,interp]`
3.	`cogen`	:=	$[\![\texttt{spec}]\!]$ `[spec,spec]`

After this renaming, the "Consequences" of Section 2.2.3 become:

A.	$[\![\texttt{interp}]\!]$ `[source,in]`	=	$[\![\texttt{target}]\!]$ `[in]`
B.	`target`	=	$[\![\texttt{compiler}]\!]$ `[source]`
C.	`compiler`	=	$[\![\texttt{cogen}]\!]$ `[interp]`

Program `target` deserves its name, since it is an L-program with the same input-output behavior as S-program `source`:

$$[\![\texttt{source}]\!]^S[\texttt{in}] = [\![\texttt{interp}]\!][\texttt{source,in}] = [\![\texttt{target}]\!][\texttt{in}]$$

Further, by the other consequences program `compiler` = `interp-gen` transforms `source` into `target` and so really is a compiler from S to L; and `cogen` is a *compiler generator* that transforms interpreters into compilers.

The compilation application of specialization imposes some natural demands on the quality of the specializer:

—The compiler `interp-gen` *must terminate* for all source program inputs.

—The target programs should be *free of all source code* from program `source`.

2.2.5 *Example of compiling by specializing an interpreter.* Consider the simple interpreter `interp` given by the pseudocode in Figure 2.[3] An interpreted source program (to compute the factorial function $n!$) might be:

```
pg = ((f (n x) (if =(n,0) then 1 else *(x, call f(-(n,1), x)))))
```

Let `target` = $[\![\texttt{spec}]\!]$ `[interp,pg]` be the result of specializing `interp` to static source program `pg`. Any reasonable compiler should "specialize away" all source code. In particular, all computations involving syntactic parameters `pg`, `e` and `ns` should be done at specialization time (considered as "static.")

[3]The interpreter is written as a first-order functional program, using Lisp "S-expressions" as data values. The current binding of names to values is represented by two lists: `ns` for names, and parallel list `vs` for their values. For instance computation of $[\![\texttt{interp}]\!]$ `[pg,[2,3]]` would use initial environment `ns = (n x)` and `vs = (2 3)`.

Operations are `car`, `cdr` corresponding to ML's `hd`, `tl`, with abbreviations such as `cadr(x)` for `car(cdr(x))`.

```
run(prog,d) = [1] eval(lkbody('main',prog),lkparm('main',prog), d, prog)

eval(e,ns,vs,pg) = case e of
  c                  : valueof(c)
  x                  : lkvar(x, ns, vs)
  basefn(e1,...,en)  : apply(basefn,[2] eval(e1,ns,vs,pg), ..., [3] eval(en,ns,vs,pg))
  let x = e1 in e2   : [4] eval(e2,
                                cons(x, ns),
                                cons([5] eval(e1,ns,vs,pg),vs),
                                pg)
  if e1 then e2  else e3: if   [6] eval(e1, ns, vs, pg)
                       then [7] eval(e2, ns, vs, pg)
                       else [8] eval(e3, ns, vs, pg)

  call f(e1,...,en)  : [9] eval(lkbody(f,pg),
                                lkparm(f,pg),
                                list([10] eval(e1,ns,vs,pg), ...,
                                     [11] eval(en,ns,vs,pg)),
                                pg)
lkbody(f,pg) = if caar(pg)=f then caddar(pg) else lkbody(f,cdr(pg))

lkparm(f,pg) = if caar(pg)=f then cadar(pg) else lkparm(f,cdr(pg))

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

Fig. 2. **interp**, an interpreter for a small functional language. Parameter **e** is an expression to be evaluated, **ns** is a list of parameter names, **vs** is a parallel list of values, and **pg** is a program (for an example, see Section 2.2.5.) The **lkparm** and **lkbody** functions find a function's parameter list and its body. **lkvar** looks up the name of a parameter in **ns**, and returns the corresponding element of **vs**.

For instance, when given the source program above, we might obtain a specialized program like this:

```
eval_f(vs) =
    if apply('=, car(vs), 0) then 1 else
    apply('*, cadr(vs),
              eval_f(list(apply('-, car(vs), 1), cadr(vs))))
```

Clearly, running `eval_f(d)` is several times faster than running `run(pg,d)`.[4]

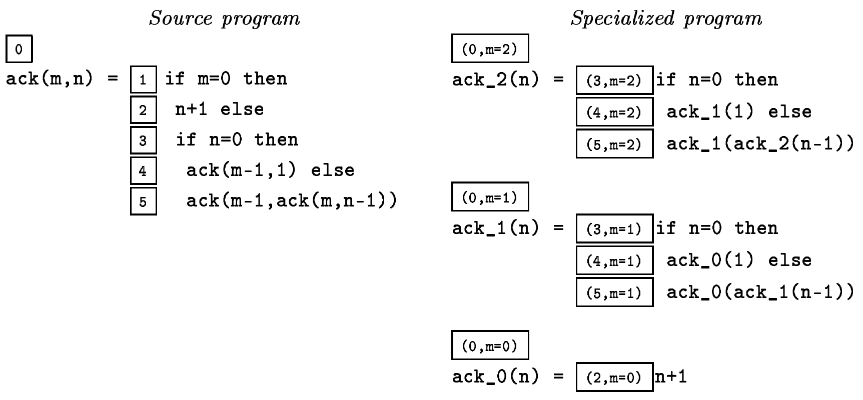
2.3 How does partial evaluation achieve its goals?

Partial evaluation is analogous to memoization, but not the same. Instead of saving a complete *value* for later retrieval, a partial evaluator generates *specialized code*

[4]Still better code can be generated! Since list `vs` always has length 2, it could be split into two components `v1` and `v2`. Further generic "`apply(op,...)`" can be replaced by specialized "`op(...)`". These give a program essentially identical to the interpreter input:

```
eval_f(v1,v2) = if =(v1,0) then 1 else *(v2,eval_f(-(v1,1), v2))
```

This is as good as can reasonably be expected, cf. the discussion of "optimal" specialization in [Jones et al. 1993; Taha et al. 2001].



Source program

```
[0]
ack(m,n) = [1] if m=0 then
           [2]  n+1 else
           [3]  if n=0 then
           [4]   ack(m-1,1) else
           [5]   ack(m-1,ack(m,n-1))
```

Specialized program

```
[(0,m=2)]
ack_2(n) = [(3,m=2)] if n=0 then
           [(4,m=2)]  ack_1(1) else
           [(5,m=2)]  ack_1(ack_2(n-1))

[(0,m=1)]
ack_1(n) = [(3,m=1)] if n=0 then
           [(4,m=1)]  ack_0(1) else
           [(5,m=1)]  ack_0(ack_1(n-1))

[(0,m=0)]
ack_0(n) = [(2,m=0)] n+1
```

Fig. 3. Partial evaluation of Ackermann's function for static m=2

(possibly containing loops) that will be executed at a later stage in time.

The specialization process is easy to understand provided the residual program contains no loops, e.g., the "power" function seen earlier can be specialized by computing values statically when possible, and generating residual code for all remaining operations. But how to proceed if *residual program loops* are needed, e.g., as in the "Ackermann" example, or when specializing an interpreter to compile a source program that contains repetitive constructs? One answer is program-point specialization.

2.3.1 *Program-point specialization.* Most if not all partial evaluators employ some form of the principle: A control point in the specializer's output program corresponds to a pair (pp, *vs*): a source program control point pp, plus some knowledge ("static data") *vs* about of the source program's runtime state. An example is shown for Ackermann's function in Figure 3 (program points are labeled by boxed numbers.) Residual functions ack_2, ack_1, ack_0 correspond to program point 0 (entry to function ack) and known values $m = 2, 1, 0$, respectively.

2.3.2 *Sketch of a generic expression reduction algorithm.* To make the issues clearer and more concrete, let exp be an expression and let *vs* be the known information about program p's run-time state.[5]

The "reduced" or residual expression $\mathtt{exp}^{resid}$ can be computed roughly as follows:

Expression reduction algorithm Reduce(exp, *vs*)*:*

(1) Reduce any subexpressions $\mathtt{e}_1,\ldots,\mathtt{e}_n$ of exp to $\mathtt{e}_1^{resid},\ldots,\mathtt{e}_n^{resid}$.

[5]This knowledge can take various forms, and is often a term with free variables. For simplicity we will assume in the following that at specialization time every parameter is either fully static (totally known) or fully dynamic (totally unknown), i.e., that there is no partially known data.

Some partial evaluators are more liberal, for example, allowing lists of statically known length with dynamic elements. This is especially important in partial evaluation of Prolog programs or for supercompilation [Glück and Sørensen 1996; Leuschel and Bruynooghe 2002; Sørensen and Glück 1995; Turchin 1979].

(2) If $\texttt{exp}$ = "$\texttt{basefn(e}_1\texttt{,...,e}_n\texttt{)}$" and some $\texttt{e}_i^{resid}$ is not fully known, then
$\texttt{exp}^{resid}$ = the **residual expression** "$\texttt{basefn(e}_1^{resid}\texttt{,...,e}_n^{resid}\texttt{)}$", else
$\texttt{exp}^{resid}$ = the **value** $[\![\texttt{basefn}]\!](\texttt{e}_1^{resid}\texttt{,...,e}_n^{resid})$.

(3) If $\texttt{exp}$ = "$\texttt{if e}_0\texttt{ then e}_1\texttt{ else e}_2$" and $\texttt{e}_0^{resid}$ is not fully known, then
$\texttt{exp}^{resid}$ = the residual expression "$\texttt{if e}_0^{resid}\texttt{then e}_1^{resid}\texttt{else e}_2^{resid}$", else
$\texttt{exp}^{resid} = \texttt{e}_1^{resid}$ in case $\texttt{e}_0^{resid}$ = "True", else $\texttt{exp}^{resid} = \texttt{e}_2^{resid}$.

(4) A function call $\texttt{exp}$ = "$\texttt{f(e}_1\texttt{,...,e}_n\texttt{)}$" can either be
residualized: $\texttt{exp}^{resid} = \texttt{f(e}_{i_1}\texttt{,...,e}_{i_m}\texttt{)}$ with some known $\texttt{e}_{i_j}$ omitted[6],
or **unfolded:** $\texttt{exp}^{resid} = \mathit{Reduce}(\text{definition of }\texttt{f}, vs')$
where vs' is the static knowledge about $\texttt{e}_1^{resid}\texttt{,...,e}_n^{resid}$.

This algorithm computes expression values depending only on static data, reduces static "if" constructs to their "then" or "else" branches, and either unfolds or residualizes function calls. The latter enables the creation of loops (recursion) in the specialized program. Any expression computation or "if"-branching that depends on dynamic data is postponed by generating code to do the computation in the specialized program.

Naturally, an expression may be reached several times by the specializer (it may be in a recursive function). If this program point and the current values of the source program, (pp, vs), have been seen before, one of two things can happen: If pp has been identified as a specialization point, the specializer generates a call in the specialized program to the code generated previously for (pp, vs). Otherwise, the call is unfolded and specialization continues as usual.

This sketch leaves several choices unspecified, in particular:

—Whether or not to unfold a function call

—Which among the known function arguments are to be removed (if any)

2.3.3 *Causes of nontermination during specialization.* First, we show an example of nontermination during specialization. A partial evaluator should not indiscriminately compute computable static values and unfold function calls, as this can cause specialization to loop infinitely even when normal evaluation would not do so. For example, the following program computing 2x in "base 1" terminates for any x:

```
double(x)     = dblplus(x,0)
dblplus(u,v) = if u <= 0 then v else dblplus(u-1,v+2)
```

Now suppose x is a dynamic (unknown) program input. A naïve specializer might "reason" that parameter v first has value 0, a constant known at specialization time. Further, if at any stage the specializer knows one value v of v, then it can compute its next value, $v + 2$. Alas, repeatedly unfolding calls to dblplus assigns to v the values $0, 2, 4, 6, \ldots$ causing an infinite loop at specialization time.

The way to avoid this problem is to not unfold the second dblplus call. The effect is to *generalize* v, i.e., to make it dynamic.

[6]If a function call is residualized, a definition of a specialized function must also be constructed by applying *Reduce* to the function's definition.

Definition 2.3.1. A parameter `x` will be called *potentially static* if dynamic program inputs are not needed to compute its value.

Three effects are the main causes of nontermination in partial evaluation:

(1) A loop controlled by a dynamic conditional can involve a potentially static parameter that takes on infinitely many values, generating an infinite set of specialized program control points $\{(\mathrm{pp}, vs_1), (\mathrm{pp}, vs_2), \ldots\}$. Parameter `v` of function `dblplus` illustrates this behavior.

(2) A too liberal policy for unfolding of function calls can cause an attempt to generate an infinitely large specialized program.

(3) As both branches of dynamic conditionals are (partially) evaluated, partial evaluation is *over-strict* in the sense that it may evaluate *more* expressions than normal evaluation would, and thus risk nontermination not present in the source program being specialized.

If there is danger of specializing a function with respect to infinitely many static values, then some of the parameters in question should be generalized, i.e., made dynamic. Instead of specializing the function with respect to *all* potentially static values, some will then become parameters in the specialized program, to be computed at run-time.

We will show how this can be done automatically, before specialization begins (of automation degree 6 in the list of Section 1.4.). A companion goal is to unfold function calls "just enough but not too much."

2.3.4 *Online and offline specialization.*

Partial evaluators fall in two categories, *on-line* and *off-line*, according to the time at which it is decided whether parameters should be generalized or calls should be unfolded (cf. Step 4 of the *Reduce* algorithm).

An online specializer generalizes during specialization. Usually this involves, at each call to a function `f`, comparing the newly-computed static arguments with the values of *all* parameters seen in earlier calls to `f`, in order to detect potentially infinite value sequences [Berlin and Weise 1990; Gallagher 1993; Leuschel 1998; Leuschel and Bruynooghe 2002; Sørensen and Glück 1995; Turchin 1979].

An offline specializer works in two stages. Stage 1, called a *binding-time analysis* (BTA for short) yields an *annotated program* in which:

—each parameter is marked as either "static" or "dynamic,"

—each expression is marked as "reduce" or "specialize," and

—each call is marked as "unfold" or "residualize."

Annotation is done before the static program input is available, and only requires knowing *which* inputs will be known at Stage 2. Stage 2, given the annotated program and the values of static inputs, only needs to obey the annotations; argument comparisons are not needed [Bondorf 1991; Christensen et al. 2000; Consel 1993; Consel and Noël 1996; Das 1998; Glenstrup et al. 1999; Glück et al. 1995; Jones et al. 1993; Hatcliff et al. 1999; Mogensen 1988].

Both kinds of specialization have their merits. Online specialization sometimes does more reduction: For example, in the specialization of Ackermann's function in Figure 3, when the call at site $\boxed{4}$ is unfolded, an online specializer will realize that the value of n is known and compute the function call. In contrast, the binding-time analysis of offline specialization must classify n as dynamic at call site $\boxed{4}$, since its value is not known at call site $\boxed{5}$.[7]

Although an online specializer can sometimes exploit static data better, this extra precision comes at a cost. A major problem is how to determine online *when to stop* unfolding function calls. Comparison of *vs* with previously seen values can result in an extremely slow specialization phase, and the specialized program may be inefficient if specialization is stopped too soon.

An offline specializer is often faster since it needs to do no decision-making online: it only has to obey the annotations. Further, *full self-application,* as in the Futamura projections, has to date only been achieved by using offline methods.

2.4 Multi-level programming languages

Many informal algorithm optimizations work by changing the times at which computations are done, e.g., by moving computations out of loops; by caching values in memory or in files for future reference; or by generating code implicitly containing partial results of earlier computations. Such a change to an algorithm is sometimes called a *staging transformation*, or a *binding-time shift.* The field of *domain-specific languages* [Hudak 1996; Kieburtz et al. 1996] employs similar strategies to improve efficiency. Partial evaluation automates this widely-used principle to gain speedup.

A BTA-annotated program can be thought of as a program in a *two-level language* in which statically annotated parts are executed, while the syntactically similar dynamic parts are in effect templates for generation of code to be executed at a later time. This line of thinking has been developed much further, e.g., by Grant et al. [2000] and, for typed languages, in Sheard's Meta-ML work [Moggi et al. 1999; Sheard 1999; Taha and Sheard 2000], and later in Taha's thesis and subsequent papers [Taha 1999b; Taha 1999a; Taha et al. 2001]. Goals of the work include efficiency improvement by staging computations; understanding how multi-level languages may be designed and implemented; resolution of tricky semantic issues, in particular questions of renaming; and formal proofs to guarantee that multi-level type safety can be achieved. A recent paper applies these ideas to ensure type-safe use of macros [Ganz et al. 2001].

Termination remains, however, a problem to be solved one case at a time by the programmer. Conceivably the ideas of this paper could contribute to a future strongly-terminating multi-level language.

Remark on language evolution. Figure 1 assumes that $\mathtt{p_{in1}}$ is an explicit, printable program. A program in a multi-level language such as Meta-ML can construct program fragments as data values and then run them, but it does not produce stand-alone, separately executable programs.

[7]Some partial evaluators allow more static computation than in in the present paper by using *polyvariant BTA,* in which the binding-time analysis can generate a finite set of different combinations of static and dynamic parameters [Christensen et al. 2000; Consel 1993].

Meta-ML is an interesting example of internalization of a concept *about* programming languages *into* a programming language construct. Analogous developments have happened before: continuations were developed to explain semantics of the **goto** and other control structures, but were quickly incorporated into new functional languages. Multi-level languages perhaps derive similarly from internalizing the fact that one language's semantics can be defined within another language, a concept also seen in Smith, Wand, Danvy and Asai's "reflective tower" languages 3-Lisp, Brown, Blonde and Black [des Rivières and Smith 1984; Friedman and Wand 1984; Wand and Friedman 1988; Danvy and Malmkjær 1988; Bawden 1988; Jefferson and Friedman 1996].

2.5 Challenging problems

2.5.1 *On the state of the art in partial evaluation.* Partial evaluation has been most successful on simple "pure" languages such as Scheme [Bondorf 1991; Bondorf and Jørgensen 1993; Consel 1993; Lawall and Thiemann 1997] and Prolog [De Schreye et al. 1999], and there has been some success on C: the C-mix [Glenstrup et al. 1999] and Tempo systems [Consel and Noël 1996]. Further, Schultz has succeeded in doing Java specialization by translating Java to C, using the Tempo system, and then translating back [Schultz 2001].

There have been a number of practically useful applications of partial evaluation, many exploiting its ability to build program generators [Glück et al. 1995; Jones 1996; McNamee et al. 2001; Lawall and Thiemann 1997; Sperber and Thiemann 2000].

Partial evaluation also has some weaknesses. One is that speedups are at most linear in the subject program's runtime (although the size of the constant coefficient can be a function of the static input data.) Further, its use can be delicate: Obtaining good speedups requires a close knowledge of the program to be specialized, and some "binding-time improvements" may be needed to get expected speedups. (These are hand work, of automation degree 4 in the list of Section 1.4.) Another weakness is that the results of specialization can be hard to predict: While speedup is common, slowdown is also possible, as is the possibility of code explosion or infinite loops at specialization time.

Little success has been achieved to date on partial evaluation of the language C++, largely due to its complex, unclear semantics; or on Java, C#, and other languages with objects, polymorphism, modules and concurrency.

One reason for the limited success in these languages is that partial evaluation requires precomputing as much as is possible, based on partial knowledge of a program's input data. To do this requires anticipating, at specialization time, the space of all possible run-time states. Such analysis can be done by abstract interpretation [Cousot and Cousot 1977; Jones and Nielson 1994] for simple functional or logic programming languages, but becomes much more difficult for more complex or more dynamic program semantics, as both factors hinder the specialization-time prediction of run-time actions.

Languages such as C++, Java and C# seem at the moment to be too complex to be sufficiently precisely analyzed, and to allow reliable assurances that program semantics is preserved under specialization.

2.5.2 *How to make specialization terminate.* Reliable termination properties are essential for highly automatic program manipulation – and are not perfect in existing partial evaluation systems. In fact, few papers have appeared on the subject, exceptions being the promising but unimplemented [Song and Futamura 2000], and works by Das and Glenstrup [Das 1998; Das and Reps 1996; Glenstrup 1999; Glenstrup and Jones 1996]. Although the generation of program generators is a significant accomplishment with promise for wider applications, program generator generation places high demands on predictability in the behavior of specialized output programs, and on the specialization process itself.

Termination of specialization can always be achieved; an extreme way is to make everything dynamic. This is useless, though, because no computations at all occur at specialization time and the specialized program is only a copy of the source program.

Our goal is thus to specialize in a way that maximizes the number of static computations, but nonetheless guarantees termination. As a "stepping stone" in order to achieve the *two-level termination* required for partial evaluation, we first investigate a novel automatic approach to ordinary termination analysis.

3. PROGRAM TERMINATION BY "SIZE-CHANGE ANALYSIS"

Our ultimate goal is to ensure, given program `p`, that program specialization: $\texttt{p}_{\texttt{in1}}$=[[`spec`]] `[p,in1]` will terminate for all inputs `in1`. Before explaining our solution to this subtle problem, we describe an automated *solution to a simpler problem*: how to decide whether a (one-stage) program terminates on all inputs.

The termination problem is interesting in itself, of considerable practical interest, and has been studied by several communities: logic programming [Lindenstrauss et al. 1997], term rewriting [Arts and Giesl 1997], functional programming [Abel and Altenkirch 1999; Lee et al. 2001; Xi 2002] and partial evaluation [Das 1998; Das and Reps 1996; Glenstrup 1999; Glenstrup and Jones 1996; Song and Futamura 2000].

A guarantee of termination is hard to achieve in practice, and undecidable in general (it is the notorious uniform halting problem). Still, dependable positive answers are essential in practice to ensure system liveness properties: that a system will never "hang" in an infinite loop, but continue to offer necessary services and make progress towards its goals. To this end, we have recently had some success in using *size-change analysis* to detect termination [Lee et al. 2001].

3.1 The size-change termination principle

Size-change analysis is based only on local information about parameter values. The starting point is a "size-change graph," G_c, one for each call $c : \texttt{f} \rightarrow \texttt{g}$ from function `f` to function `g` in the program. Graph G_c describes parameter value changes when call c is made (only equalities and decreases). The graphs are derivable from program syntax, using elementary properties of base functions (plus a size analysis to describe the results of nested function calls, see [Chin and Khoo 2002; Hughes et al. 1996; Jones et al. 1993]).

In the following, we shall consider programs written in a first-order functional language with well-founded data[8] and we say that a call sequence $cs = c_1c_2\ldots$ is *valid* for a program if it follows the program's control flow. The key to our termination analyses is the following principle:

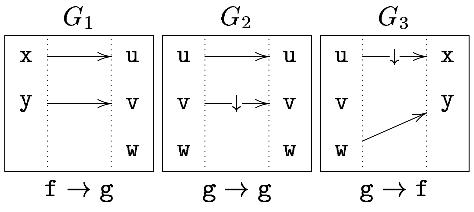
> A program terminates (on any input to any function) if *every valid infinite call sequence* would, if executed, cause an infinite decrease in some parameter values.

Example of size-change termination analysis. Consider a program on natural numbers that uses mutual recursion to compute $2^x \cdot y$.

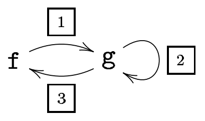
```
f(x,y)   = if x = 0 then y else [1] g(x,y,0)
g(u,v,w) = if v > 0 then [2] g(u,v-1,w+2)
                     else [3] f(u-1,w)
```

Following are the three size-change graphs for this program, along with its call graph and a description of its valid infinite call sequences. There is one size-change graph G_c for each call c, describing both the data flow and the parameter size changes that occur if that call is taken. To keep the analysis simple, we approximate the size changes by either '$\overset{=}{\downarrow}$' (the change does not increase the value) or $\downarrow$ (the change decreases the value); normally we omit the '$\overset{=}{\downarrow}$' labels from graphs for readability. The valid infinite call sequences can be seen from the call graph, and are indicated by ω-regular expressions[9].

Size-change graphs



Call graph



Valid infinite call sequences

$$(12^*3)^\omega + (12^*3)^*12^\omega$$

An informal argument for termination. Examination of the size-change graphs reveals that a call sequence ending in 12^ω causes parameter v to decrease infinitely, but is impossible by the assumption that data values are well-founded. Similarly, a call sequence in $(12^*3)^\omega$ causes parameters x and u to decrease infinitely. The natural numbers are well-formed, so both are impossible. Consequently *no valid infinite call sequence is possible*, so the program terminates.

[8]I.e., no infinitely decreasing value sequences are possible. With the aid of other analyses this assumption may be relaxed so as to cover, for example, the integers.

[9]$*$ indicates any finite number of repetitions, and ω indicates an infinite number of repetitions.

3.2 A termination algorithm

It turns out that the set of valid infinite call sequences that cause infinite descent in some value is finitely describable[10]. Perhaps surprisingly, the infinite descent property is *decidable*, e.g., by automata-theoretic algorithms. We sketch an algorithm operating directly on the size-change graphs without the passage to automata.

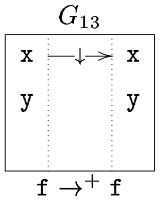
Two size-change graphs $G : \texttt{f} \to \texttt{g}$ and $G' : \texttt{g} \to \texttt{h}$ may be composed in an obvious way to yield a size-change graph $G; G' : \texttt{f} \to \texttt{h}$. Their total effect can be expressed in a single size-change graph, like $G_{13} = G_1; G_3$, shown below. These compositions are used in the central theorem of [Lee et al. 2001]:

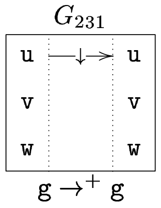
THEOREM 3.2.1. *Let the* closure $\mathcal{S}$ *of the size-change graphs for program* p *be the smallest graph set containing the given size-change graphs, such that* $G; G' \in \mathcal{S}$ *whenever* $\mathcal{S}$ *contains both* $G : \texttt{f} \to \texttt{g}$ *and* $G' : \texttt{g} \to \texttt{h}$. *Then* p *is size-change terminating if and only if every idempotent* $G \in \mathcal{S}$ *(i.e., every graph satisfying* $G = G; G$*) has an* in situ *descent* $\texttt{z} \overset{\downarrow}{\to} \texttt{z}$.

The closure set for the example program is:

$$\mathcal{S} = \{G_1, G_2, G_3, G_{12}, G_{13}, G_{131}, G_{23}, G_{231}, G_{31}, G_{312}\}$$

This theorem leads to a straightforward algorithm. The idempotent graphs in $\mathcal{S}$ are G_2 (above) as well as G_{13} and G_{231} (below). Each of them has an *in situ* decreasing parameter, so no infinite computations are possible.





3.3 Assessment

Compared to other results in the literature, termination analysis based on the size-change principle is surprisingly simple and general: lexicographical orders, indirect function calls and permuted arguments (descent that is not *in-situ*) are all handled automatically and without special treatment, with no need for manually supplied argument orders, or theorem-proving methods not certain to terminate at analysis time. It has recently been proven [Ben-Amram 2002] that the functions computable by size-change terminating are exactly Péter's "multiple recursive" functions [Péter 1976]. This is an enormous class, so the expressive power of size-change terminating programs is quite high.

Converging insights. This termination analysis technique has a history of independent rediscovery. We first discovered the principle while trying to communicate our earlier, rather complex, binding-time analysis algorithms to ensure termination of specialization [Glenstrup 1999; Glenstrup and Jones 1996]. The size-change termination principle arose while trying to explain our methods simply, by starting

[10]It is an ω-regular set, representable by a Büchi automaton.

with one-stage computations as an easier special case [Lee et al. 2001]. To our surprise, the necessary reformulations led to BTA algorithms that were stronger as well as easier to explain.

Independently, Lindenstrauss and Sagiv had devised some more complex graphs to analyze termination of logic programs [Lindenstrauss et al. 1997], based on the same mathematical properties as our method. A third discovery: the Sistla-Vardi-Wolper algorithm for determinization of Büchi automata [Prasad Sistla et al. 1987] resembles our closure construction. This is no coincidence, as the infinite descent condition is quite close to the condition for acceptance of an infinite string by a Büchi automaton.

Termination analysis in practice. Given the importance of the subject, it seems that surprisingly few termination analyses have been implemented. In logic programming, the Termilog analyzer [Lindenstrauss et al. 1997] can be run from a web page; and the Mercury termination analysis [Speirs et al. 1997] has been applied to all the programs comprising the Mercury compiler. In functional programming, analyses by Abel and Altenkirch [1999] and Lee et al. [2001] have both been implemented as part of the AGDA proof assistant [Coquand 2001]. Further, Xi [2002] uses dependent types with user-supplied annotations to verify termination, and Grobauer [2001] refines this approach to verify run-time bounds. A recent paper on termination analysis of imperative programs is written by Colón and Sipma [2002].

3.4 Quasitermination

The following closely related concept will turn out to be central for ensuring that partial evaluation terminates. A quasiterminating program is one that enters only finitely many different states during any one computation. Of course, any terminating program is also quasiterminating.

Definition 3.4.1. Program `p` is *quasiterminating* iff for any input vector $\overline{in}$, the following is a *finite set*:

$$Reach(\overline{in}) = \{(\mathtt{f}, \overline{v}) \mid \text{Computation of } [\![\mathtt{p}]\!][\overline{in}] \text{ calls } \mathtt{f} \text{ with argument } \overline{v}\}$$

This can be re-expressed in terms of properties of individual function parameters:

Definition 3.4.2. Assume program `p` contains the definition $\mathtt{f(x_1,\ldots,x_n)}$ = `expression`. The *variation* of parameter $\mathtt{x}_i$ for program input $\overline{in}$ is the set:

$$Var(\mathtt{x}_i, \overline{in}) = \left\{v_i \;\middle|\; (\mathtt{f}, v_1 \ldots v_n) \in Reach(\overline{in}) \;\; \textit{for some inputs } \overline{in} \;\right\}$$

We say $\mathtt{x}_i$ is of *bounded variation* (written BV, for short,) iff for every input *in* the set $Var(\mathtt{x}_i, \overline{in})$ is finite.

LEMMA 3.4.3. *Program* `p` *is quasiterminating if and only if all its parameters are of bounded variation.*

A simple example: program `double` of Section 2.3.3 is terminating, and thus also quasiterminating. The variations of its parameters:

$$\begin{aligned} Var(\mathtt{x}, x) &= \{x\} \\ Var(\mathtt{u}, x) &= \{x, x-1, \ldots, 1, 0\} \\ Var(\mathtt{v}, x) &= \{0, 2, \ldots, 2x\} \end{aligned}$$

4. GUARANTEEING TERMINATION OF PROGRAM GENERATION

We now show how to extend the size-change principle to an analysis to guarantee termination of specialization in partial evaluation. The results described here are significantly better than those of Chapter 14 in [Jones et al. 1993] and [Glenstrup 1999; Glenstrup and Jones 1996]. Although the line of reasoning is similar, the following results are stronger and more precise, further, a prototype implementation exists.

4.1 Online specialization with guaranteed termination

Most online specializers record the values of the function parameters seen during specialization, so that current static values *vs* can be compared with earlier ones in order to detect potentially infinite value sequences. A rather simple strategy (residualize whenever any function argument value increases) is sufficient to give good results on the Ackermann example seen earlier.

More liberal conditions that guarantee termination of specialization will yield more efficient residual programs. The most liberal conditions known to the authors employ variants of the Kruskal tree condition called "homeomorphic embedding" [Sørensen and Glück 1995]. This test, carried out during specialization, seems expensive but strong. It is evaluated on a variety of test examples in the literature [Leuschel 1998].

4.2 Online specialization with call-free residual programs

The termination-detecting techniques of Section 3 can easily be extended to find a sufficient precondition for online specialization termination to yield call-free residual programs. The idea is to modify the size-change termination principle as follows (recall Section 2.3.3, Definition 2.3.1):

> Suppose every valid infinite call sequence causes an *infinite descent in some potentially static parameter.* Then online program specialization will terminate on any static input, to yield a call-free residual program.

A program passing this test will have no infinite specialization-time call sequences. This means that an online specializer may proceed blindly, computing every potentially static value and unfolding all function calls.

This condition can be tested by a slight extension of the closure algorithm of Section 3:

—Before specialization, identify the set *PS* of potentially static parameters.

—Construct the closure $\mathcal{S}$ as in Section 3.

—Decide this question: Does every idempotent graph $G \in \mathcal{S}$ have an *in situ* decreasing parameter in *PS*?

Strengths:

(1) No comparisons or homeomorphic embedding tests are required, so specialization is quite fast.

(2) The condition succeeds on many programs, e.g.,
—the "power" example of Section 2.2 specialized to static input `n`, or
—Ackermann's function specialized to static `m` and `n`.

(3) If a size-change terminating program has no dynamic inputs, the test above will succeed and residual programs will have the form "`f(_) = constant`".

(4) Further, any program passing this test is free of "static loops," a notorious cause of nonterminating partial evaluation.

Weakness. The methods fails, however, for the Ackermann example with static `m` and dynamic `n`, or for interpreter specialization with static program argument `pg`. The problem is that specialized versions of such programs must contain loops.

4.3 Offline specialization of an interpreter

As mentioned in Section 2.2.4, the property that source syntax is "specialized away" is vital for compiling efficient target programs, and when performing self-application of the specializer to generate compilers. Our main goal is to achieve a high degree of specialization (i.e., a fast specialized program) and at the same time a *guarantee* that the specialization phase terminates.

Consider specializing the simple interpreter `interp` of Figure 2 (Section 2.2.4) to static source program `pg` and dynamic program input `d`. One would hope and expect all source code to be specialized away—in particular, syntactic parameters `pg`, `e` and `ns` should be classified as "static" and so not appear in target programs.

In Figure 2 parameter `vs` is dependent on dynamic program input `d`, and so must appear in the specialized programs. Parameter `pg` always equals static input `prog` (it is only copied in all calls), so any specializer should make it static.

Parameter `e` decreases at every call site except $\boxed{9}$, where it is *reset* (always to a subterm of `pg`). After initialization or this reset, parameter `ns` is only copied, except at call site $\boxed{4}$, where it is *increased.*

A typical *online* specializer might see at call site $\boxed{9}$ that `eval`'s first argument `e` has a value larger than any previously encountered, and thus would consider it dynamic. By the same reasoning, argument `ns` would be considered dynamic at call site $\boxed{4}$ due to the evident risk of unboundedness. Alas, such classifications will yield unacceptably poor target programs.

However the binding-time analysis of an *offline* partial evaluator considers the interpreter program as a whole, and can detect that `eval`'s first argument can only range over subterms of the function bodies in the interpreted program `pg`. This means that parameter `e` can be considered as "static" without risking nontermination, and can thus be specialized away. Further, parameter `ns` will always be a part of the interpreter's program input `pg`. Offline specialization can recognize these facts, and so ensure that *no source code from* `pg` *appears in any target program* produced by specializing this interpreter.

Conclusion: control of the specializer's termination is easier offline than "on the fly." The effect just described is not easily achieved by online methods.

Parameters that are reset to bounded values. We now describe a more subtle automatic "boundedness" analysis that reveals that the set of values `ns` ranges over during specialization is also finite[11]. First, some terminology.

[11]Remark: this is a delicate property, holding only because this `interp` implements "static name binding." Changing the `call` code as follows to implement dynamic name binding would make `ns` necessarily dynamic during specialization:

4.4 Bounded static variation

A *necessary condition for termination of offline specialization* is that *all of the program's static parameters are of bounded variation.* We define this more precisely:

Definition 4.4.1. Assume program p contains the definition `f(x`$_1$`,...,x`$_n$`) = expression`. Parameter $\mathtt{x}_i$ is of *bounded static variation* (BSV for short) if for all static inputs $\overline{sin}$, the following set is finite:

$$StatVar(\mathtt{x}_i, \overline{sin}) = \left\{ v_i \;\middle|\; \begin{array}{l} (\mathtt{f}, v_1 \ldots v_n) \in Reach(\overline{in}) \;\; \textit{for some} \\ \text{inputs } \overline{in} \text{ with } \overline{sin} = \text{staticpart}(\overline{in}) \end{array} \right\}$$

Two illustrative examples both concern the program `double` of Section 2.3.3. Program `double` is terminating; and parameter `u` is of BSV, while `v` is not.

```
double(x)     = [1] dblplus(x,0)
dblplus(u,v) = if u <= 0 then v else [2] dblplus(u-1,v+2)
```

Case 1. Input `x` is static, and

$$\begin{aligned} StatVar(\mathtt{u}, x) &= \{x, x-1, \ldots, 1, 0\} \\ StatVar(\mathtt{v}, x) &= \{0, 2, \ldots, 2x\} \end{aligned}$$

Both are finite for any static input `x`. The BTA can annotate both `u` and `v` as static because `dblplus` will only be called with finitely many value pairs $(\mathtt{u}, \mathtt{v}) = (x, 0)$, $(x-1, 2), \ldots, (0, 2x)$, where x is the initial value of `x`.

Case 2. Input `x` is dynamic, and

$$\begin{aligned} StatVar(\mathtt{u}, \varepsilon) = \{0, 1, 2, \ldots\} \\ StatVar(\mathtt{v}, \varepsilon) = \{0, 2, 4, \ldots\} \end{aligned}$$

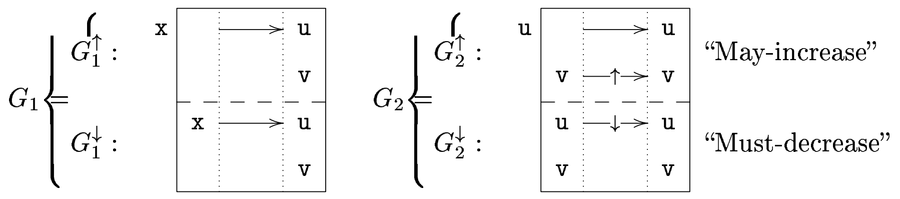
(Here ε is the empty list of static program inputs.) Here `u` must be dynamic since it depends on dynamic input `x`. Further, even though `v` is computed without use of dynamic inputs, the BTA *must not* annotate `v` as static: If it did so, then `dblplus` would be specialized infinitely, to: $\mathtt{v} = 0, \mathtt{v} = 2, \mathtt{v} = 4, \ldots$

4.5 "Must-decrease" and "may-increase" properties

Detecting violations of the BSV condition requires, in addition to the *must-decrease* parameter size properties on which size-change termination is based, *may-increase* properties as well. Abstract interpretation or constraint solving analyses may be used to detect both modalities of size-change behavior, and can be found in the literature [Chin and Khoo 2002; Hughes et al. 1996]. We develop analyses for the current context in Section 5, in the style of Jones et al. [1993, Section 14.3].

A simple extension of the size-change graph formalism uses *two-layered* size-change graphs $G = (G^{\uparrow}, G^{\downarrow})$ where (as before) $G^{\downarrow}$ approximates "must-decrease" properties on which the size-change approach is based, and $G^{\uparrow}$ safely approximates "may-increase" size relations by arcs $\mathtt{x} \xrightarrow{\uparrow} \mathtt{y}$. An example: the `double` program seen before has two-layered size-change graphs:

```
call f(e1,...,en): [9] eval(lkbody(f,pg), append(lkparm(f,pg),ns),...)
```



4.6 Constraints on binding-time analysis

The purpose of binding-time analysis is safely to annotate a program. The central task is to find a so-called *division* β : *ParameterNames* $\to \{S, D, ?\}$ that classifies every function parameter[12] as "static," "dynamic," or "as yet undecided." The desired division should have no ?-values and be "as static as possible" while ensuring that specialization will terminate in all cases. To this end define $\beta \sqsupseteq \beta'$ to hold iff $\beta'(\mathtt{x}) \neq ?$ implies $\beta(\mathtt{x}) = \beta'(\mathtt{x})$ for all parameters x.

Constraints on a division β.

(1) $\beta \sqsupseteq \beta_0$ where β_0 is the *initial division*, mapping each program input parameter to its given binding time and all other parameters to ?.

(2) $\beta(\mathtt{x}) = D$ if x is not of BSV.

(3) $\beta(\mathtt{x}) = D$ if the value of x depends on the value of some y with $\beta(\mathtt{y}) = D$.

(4) $\beta(\mathtt{x}) = D$ if x depends on the result of a residual call.

We now proceed to develop two principles that can be used to assign $\beta(\mathtt{x}) := S$, both using ideas from size-change termination.

4.7 Second and better try: Bounded Domination

A new principle for termination of offline program specialization:

> Let $\beta(\mathtt{x}) = ?$ where β satisfies the constraints in Section 4.6[13]. If no valid infinite call sequence causes x to increase infinitely, then x is of BSV.

This analysis goes considerably farther than the "call-free" method of Section 4.2. Further, it is relatively easy to implement applying familiar graph algorithms. First, define $\mathtt{x} \equiv \mathtt{w}$ iff parameter x depends on w and w depends on x, and define the equivalence class: $[\mathtt{x}] = \{\mathtt{w} \mid \mathtt{x} \equiv \mathtt{w}\}$.

(a) Compute the closure $\mathcal{S}$ of program p's size-change graphs (now two-layered).

(b) Let $\beta := \beta_0$

(c) Let $\beta(\mathtt{x}) := D$ for every x that depends on some y with $\beta(\mathtt{y}) = D$.

(d) Identify, as a candidate for upgrading, any x such that $\beta(\mathtt{x}) = ?$, and $\beta(\mathtt{y}) = S$ whenever x depends on $\mathtt{y} \notin [\mathtt{x}]$.

(e) Reclassify all parameters $\mathtt{z} \in [\mathtt{x}]$ by $\beta(\mathtt{z}) := S$ if no idempotent $G \in \mathcal{S}$ contains $\mathtt{w} \xrightarrow{\uparrow} \mathtt{w}$ for any $\mathtt{w} \in [\mathtt{x}]$.

[12]We assume without loss of generality all functions have distinct parameter names, and that the program contains no calls to its initial function.
[13]Note that x must be potentially static, by Constraint 3.

(f) Iterate Steps d and e to stability.

(g) Change remaining ?'s in β to D.

Explanation: Steps b and c follow constraints 1 and 3. Steps d and e apply the bounded domination principle to classify some equivalence classes [x] as static. Step g follows constraint 2.

For `double` with static input, Step e changes initial $\beta = [\mathtt{x} \mapsto S, \mathtt{u} \mapsto ?, \mathtt{v} \mapsto ?]$ into

$$\beta = [\mathtt{x} \mapsto S, \mathtt{u} \mapsto S, \mathtt{v} \mapsto ?]$$

Ackermann specialization terminates by similar reasoning. Now consider the interpreter example of Figure 2 with initial

$$\beta = [\mathtt{prog} \mapsto S, \mathtt{d} \mapsto D, \mathtt{e} \mapsto ?, \mathtt{ns} \mapsto ?, \mathtt{vs} \mapsto ?, \mathtt{pg} \mapsto ?, \mathtt{f} \mapsto ?, \mathtt{x} \mapsto ?]$$

pg is clearly of BSV because it is copied in all calls and never changed (increased or decreased). Further, call sites 2, 3 and 5–10 only pass values to `e` from substructures of `e` or pg, so the values of `e` are always a substructure of the static input. Thus `e` is of BSV by the Bounded Domination principle and can be annotated "static" (even though at call site $\boxed{9}$ its value can become larger from one call to the next.) This and constraint 3 of Section 4.6 gives:

$$\beta = [\mathtt{prog} \mapsto S, \mathtt{d} \mapsto D, \mathtt{e} \mapsto S, \mathtt{ns} \mapsto ?, \mathtt{vs} \mapsto D, \mathtt{pg} \mapsto S, \mathtt{f} \mapsto S, \mathtt{x} \mapsto S]$$

On the other hand, call site $\boxed{4}$ poses a problem: `ns` can increase, which (so far) will cause it to be annotated "dynamic." This is more conservative than necessary, as the name list `ns` only takes on finitely many values when interpreting any fixed program pg.

4.8 Third and still better try: Bounded Anchoring

A still more general principle allows potentially static parameters that *increase*:

> Suppose $\beta(\mathtt{x}) = ?$ and $\beta(\mathtt{y}) = S$ where β satisfies the constraints of Section 4.6. If every valid infinite call sequence *cs* that infinitely increases x also infinitely *decreases* y, then x is of BSV.

The set of known BSV parameters can iteratively be extended by this principle, starting with ones given by Bounded Domination[14]. Again, it is relatively easy to implement; just replace Steps d and e above by the following:

(d′) Identify, as a candidate for upgrading, any x such that $\beta(\mathtt{x}) = ?$, and $\beta(\mathtt{y}) = S$ whenever x depends on $\mathtt{y} \notin [\mathtt{x}]$.

(e′) Reclassify parameters $\mathtt{z} \in [\mathtt{x}]$ by $\beta(\mathtt{z}) := S$ if every idempotent $G \in \mathcal{S}$ containing $\mathtt{w} \xrightarrow{\uparrow} \mathtt{w}$ for some $\mathtt{w} \in [\mathtt{x}]$ *also* contains $\mathtt{y} \xrightarrow{\downarrow} \mathtt{y}$ for some y with $\beta(\mathtt{y}) = S$.

For the `double` example, Step e′ allows changing $\beta = [\mathtt{x} \mapsto S, \mathtt{u} \mapsto S, \mathtt{v} \mapsto ?]$ into

$$\beta = [\mathtt{x} \mapsto S, \mathtt{u} \mapsto S, \mathtt{v} \mapsto S]$$

[14]In fact, the Bounded Domination principle can be seen as the special case of Bounded Anchoring where the set of call sequences that infinitely increase x is empty.

$$\begin{aligned}
\textit{Program} \ni p &::= d_1 \ldots d_n \\
\textit{Def} \ni d &::= f\, x_1 \ldots x_n = e^f \\
\textit{Expression} \ni e &::= k \mid x \mid \textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3 \mid b\, e_1 \ldots e_n \mid \boxed{c}\, f\, e_1 \ldots e_n
\end{aligned}$$

$$k \in \textit{Constant} \quad x \in \textit{Varname} \quad c \in \textit{CallSite}$$
$$b \in \textit{Basefuncname} = \{\texttt{car}, \texttt{cdr}, \texttt{cons}, \ldots\} \quad f \in \textit{Funname}$$

Fig. 4. Syntax for the first-order functional language treated by the analysis

In the interpreter example, the previous analysis shows that `e` and `pg` are of BSV. The remaining parameter `ns` can increase (call 4), but a call sequence $\ldots\boxed{4}\ldots$ with an *in situ* increase of `ns` also has an *in situ* decrease in `e`. This implies `ns` cannot increase unboundedly and so is also of BSV. Conclusion (as desired):

$$\beta = [\texttt{prog} \mapsto S, \texttt{d} \mapsto D, \texttt{e} \mapsto S, \texttt{ns} \mapsto S, \texttt{vs} \mapsto D, \texttt{pg} \mapsto S, \texttt{f} \mapsto S, \texttt{x} \mapsto S]$$

The only dynamic parameter of function `eval` is `vs`, so target programs obtained by specialization will be free of all source program syntax.

4.9 Specialization point insertion

The discussion above was rather quick and did not cover Constraint 4 of Section 4.6, which concerns the function unfolding policy to be used. The justification of the constraint is that the result of a residual call is not available at specialization time.

There is a clear danger of infinite specialization-time unfolding, for instance even in the `double` example. This can be prevented by doing no unfolding at all, but by Constraint 4 this could force other parameters to be dynamic.

A more liberal unfolding policy can be used, based upon a *specialization-point insertion* analysis to mark a limited set of call sites as specialization points, not to be unfolded. These should be as few as possible, just enough to prevent infinite loop unrolling. This is not explained here, since this section is already long enough. More details, and correctness proofs, are given in the following sections.

5. SEMANTIC FOUNDATIONS FOR TERMINATION ANALYSIS

As mentioned in Section 1.3, it is vital that the program transformations rest on a firm basis of the programming language semantics if the user is to trust the generated programs. In the following sections we will present these foundations and correctness proofs for the termination analysis.

5.1 Syntax and semantics

The analyses presented operate on a strict first-order functional language with the formal syntax definition shown in Figure 4. For a function f we write $f^{(i)}$ for the ith parameter of f, and label each function call site with a unique label c. *CallSite* is the set of labels in the program. The functions defined by definitions $d_1, \ldots, d_n$ are referred to as $f1, \ldots, fn$. The semantics for normal, strict evaluation of this language, shown in Figure 5, is quite standard; all base functions are assumed to terminate, possibly with an error value, for all input. Given a tuple $\vec{v}$, v_i denotes the ith component. We assume given a domain *Value* $\ni v$ and a size relation $\leq\ \in$ *Value* $\times$ *Value* which makes *Value* well-founded. Furthermore, we make use

$$\mathcal{E} : \mathit{Expression} \to \mathit{Value}^* \to \mathit{Value} \cup \{\mathsf{error}, \bot\}$$

$$\begin{aligned}
\mathcal{E} = \mathcal{E}_{\mathsf{e}}(\mathit{fix}(\lambda\phi.\{ &\mathit{f1} \mapsto \lambda v_1 \ldots v_m . \mathcal{E}_{\mathsf{e}}\, \phi[\![e^{f1}]\!](v_1, \ldots, v_m), \ldots, \\
&\mathit{fn} \mapsto \lambda v_1 \ldots v_k\ . \mathcal{E}_{\mathsf{e}}\, \phi[\![e^{fn}]\!](v_1, \ldots, v_k)\}))
\end{aligned}$$

$$\begin{aligned}
&\mathcal{E}_{\mathsf{e}}\, \phi[\![k]\!]\vec{v} && = \mathit{value}\ k \\
&\mathcal{E}_{\mathsf{e}}\, \phi[\![x_i]\!]\vec{v} && = v_i \\
&\mathcal{E}_{\mathsf{e}}\, \phi[\![\mathbf{if}\ \ e_1\ \ \mathbf{then}\ \ e_2\ \ \mathbf{else}\ \ e_3]\!]\vec{v} && = \text{if } v_1' \text{ then } v_2' \text{ else } v_3', && \text{where } v_i' = \mathcal{E}_{\mathsf{e}}\, \phi[\![e_i]\!]\vec{v} \\
&\mathcal{E}_{\mathsf{e}}\, \phi[\![b\ e_1 \ldots e_n]\!]\vec{v} && = \mathit{apply}\ \ b\ v_1' \ldots v_n', && \text{where } v_i' = \mathcal{E}_{\mathsf{e}}\, \phi[\![e_i]\!]\vec{v} \\
&\mathcal{E}_{\mathsf{e}}\, \phi[\![f\ e_1 \ldots e_n]\!]\vec{v} && = \phi\ f\ v_1' \ldots v_n', && \text{where } v_i' = \mathcal{E}_{\mathsf{e}}\, \phi[\![e_i]\!]\vec{v}
\end{aligned}$$

Fig. 5. Normal evaluation semantics. Use of 'if' is strict in its first argument, and applications of functions *apply* and ϕ are strict, returning $\bot$ or error if any of their arguments are $\bot$ or error

of the function *value* : *Constant* $\to$ *Value* which perform conversions between expressions and values, and also the standard function domain and range operators *dom* and *rg*. The function *apply* : *Basefuncname* $\to$ *Value** $\to$ (*Value* $\cup$ {error}) returns the value resulting from the application of a base function to its arguments. Functions *value* and *apply* are assumed to terminate for any input.

In the ensuing text we will use the following notation:

Definition 5.1.1. For any set A we define A^* to be the set of all finite sequences over A; and A^ω to be the set of all infinite sequences over A; and $A^{*\omega} = A^* \cup A^\omega$. We use the same notation: $as = a_1a_2a_3\ldots$ for elements of either A^* or A^ω, and write $as = a_1a_2a_3\ldots a_n$ for elements of A^*.

Definition 5.1.2. (Call sequence) A *call sequence* is any finite or infinite $cs \in \mathit{CallSite}^{*\omega}$. Call sequence $c_1c_2\ldots$ is *well-formed* iff c_i labels a call to a function f whose body e^f contains the call labeled with c_{i+1}.

Definition 5.1.3. (Program States and Transitions)

—A *state* is a pair $(f, \vec{v}) \in \mathit{Funname} \times \mathit{Value}^*$.

—A *state transition* $(f, \vec{v}) \xrightarrow{c} (g, \vec{u})$ is a pair of states connected by a call $\boxed{c}\ g\ e_1 \ldots e_n$ in the body e^f of f, such that $\vec{u} = (u_1, \ldots, u_m)$ and $\mathcal{E}[\![e_i]\!]\vec{v} = u_i$.

—A *state transition sequence* is a sequence $\sigma = (f_0, \vec{v}_0) \xrightarrow{c_1} (f_1, \vec{v}_1) \to \cdots$ (finite or infinite), where $(f_i, \vec{v}_i) \xrightarrow{c_{i+1}} (f_{i+1}, \vec{v}_{i+1})$ is a state transition for each $i = 0, 1, \ldots$

—The *calls in a state transition sequence* $c_1c_2\ldots$ are given by $\mathit{calls}(\sigma) = c_1c_2\ldots$

—The set of *encountered values* for a state transition sequence σ is defined as

$$\mathcal{V}(\sigma) = \left\{ (x, V) \,\middle|\, \begin{array}{l} x \text{ is } f^{(i)}, \text{ i.e., parameter number } i \text{ of } f \\ \wedge\, V = \{v_i \mid \sigma = (f_0, \vec{v}_0) \to \cdots \to (f, \vec{v}) \to \cdots\} \end{array} \right\}$$

In the following we assume given a fixed program p.

5.2 Size-change descriptors and graphs

Recall Sections 3.4, 4.4 and 4.5. Our overall purpose is to detect sequences of infinitely descending parameter values (in order to establish termination), and to find parameters of bounded variation (if statically computable, they can be computed at program specialization time).

The “must-decrease” and “may-increase” graphs $G = (G^{\downarrow}, G^{\uparrow})$ are at the heart of size-change analysis. The edge labels in $G^{\downarrow}, G^{\uparrow}$ relate the size of an expression’s value to the sizes of its parameters, so it is essential to know what information they convey, and how it relates to our purposes. More precisely:

We let $D^{\downarrow} = \{\downarrow, \overline{\overline{\downarrow}}\}, D^{\uparrow} = \{\uparrow, \underline{\underline{\uparrow}}\}$ be two sets of *dependency labels* and impose an ordering $\sqsubseteq$ on the labels by $\overline{\overline{\downarrow}} \sqsubseteq \overline{\overline{\downarrow}}$, $\overline{\overline{\downarrow}} \sqsubseteq \downarrow$, $\downarrow \sqsubseteq \downarrow$, $\underline{\underline{\uparrow}} \sqsubseteq \underline{\underline{\uparrow}}$, $\underline{\underline{\uparrow}} \sqsubseteq \uparrow$, $\uparrow \sqsubseteq \uparrow$.

A *dependency* is such a label together with a parameter, with intuitive meanings:

$\downarrow(x)$	‘definitely depending on x and less than its value,’
$\overline{\overline{\downarrow}}(x)$	‘definitely depending on x and less than or equal to its value,’
$\underline{\underline{\uparrow}}(y)$	‘possibly depending on y, and unbounded if y is unbounded’
$\uparrow(y)$	‘possibly depending on y, and greater than y for unbounded y.’

We name the collections of such dependency sets[15]:

$$\mathit{Dep}^{\downarrow} \overset{\text{def}}{=} \mathcal{P}(D^{\downarrow} \times \mathit{Varname}).\mathit{Dep}^{\uparrow} \overset{\text{def}}{=} \mathcal{P}(D^{\uparrow} \times \mathit{Varname})$$

Each expression can be assigned a set of decreasing and a set of increasing dependencies. For instance, when evaluating the expression `cdr` x, the resulting value is *always* less than the value of x, written as $\{\downarrow(x)\}$, no matter the value of x. The value of expression x is always less than or equal to x, written as $\{\overline{\overline{\downarrow}}(x)\}$.

The set of increasing dependencies of `cons` x y is $\{\uparrow(x), \uparrow(y)\}$ since its value is greater than the values of x and y; and the set of decreasing dependencies of `cons` x y is empty. The set of increasing dependencies of x is $\{\underline{\underline{\uparrow}}(x)\}$ since its value is greater than or equal to x. The set of increasing dependencies of `cdr` x is *also* $\{\underline{\underline{\uparrow}}(x)\}$, since its value will be unbounded if x’s value is unbounded.

The analyses presented in detail in Appendix B center around two approximation operators

$$\begin{aligned} \mathcal{E}^{\downarrow} &: \mathit{Expression} \to (\mathit{Dep}^{\downarrow})^* \to \mathit{Dep}^{\downarrow} \\ \mathcal{E}^{\uparrow} &: \mathit{Expression} \to (\mathit{Dep}^{\uparrow})^* \to \mathit{Dep}^{\uparrow}. \end{aligned}$$

where $\mathcal{E}^{\downarrow}[\![e]\!]\vec{\Delta} = \Delta$ means: “Δ is a “must-decrease” description of the value of e, given size descriptions $\vec{\Delta}$ of its free variables,” and $\mathcal{E}^{\uparrow}[\![e]\!]\vec{\Delta} = \Delta$ means: “Δ is a “may-increase” description of the value of e, given size descriptions $\vec{\Delta}$ of its free variables,”

These operators must terminate on all expressions (so that the termination analysis can terminate), and return a set that describes correctly (but perhaps approximately) the size relation between the value of an expression (as computed by $\mathcal{E}$) and the values of its free parameters. For example we expect

$$\mathcal{E}^{\downarrow}[\![\texttt{cdr}\ x]\!](\{\overline{\overline{\downarrow}}(x)\}) = \{\downarrow(x)\}, \quad \mathcal{E}^{\uparrow}[\![\texttt{cons}\ x\ y]\!](\{\underline{\underline{\uparrow}}(x)\}, \{\underline{\underline{\uparrow}}(y)\}) = \{\uparrow(x), \uparrow(y)\}$$

As $\downarrow$ is a stronger statement than $\overline{\overline{\downarrow}}$ and $\uparrow$ a stronger statement than $\underline{\underline{\uparrow}}$ we remove $\overline{\overline{\downarrow}}(x)$ from sets containing $\downarrow(x)$ and $\underline{\underline{\uparrow}}(x)$ from sets containing $\uparrow(x)$.

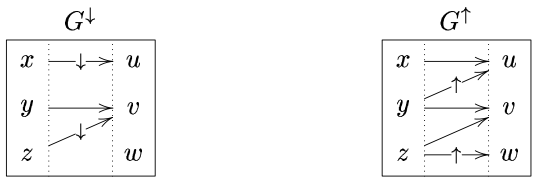
Now that we are able to relate the size of an expression value to the sizes of its free variables, we are able to construct the *size-change graphs* that represent the

[15] $\mathcal{P}$ is the power set operator: $\mathcal{P}(X) = \{Y \mid Y \subseteq X\}$.

size changes of the function parameters from one call to another. They are formally given by

Definition 5.2.1. (Size-change graphs)

—A *size-change graph* is a pair $G = (G^{\downarrow}, G^{\uparrow})$, usually written $G = \begin{bmatrix} G^{\uparrow} \\ G^{\downarrow} \end{bmatrix}$, where $G^{\mu} = (var(f), var(g), E)$ is a bipartite graph with edges $E \subseteq var(f) \times D^{\mu} \times var(g)$. Often, we will draw a graphical representation of size change graphs like this example:



where ↧ and ↥ labels have been omitted for readability. We will also refer to $G^{\downarrow}$ and $G^{\uparrow}$ as size-change graphs.

—Given a call site $c : g\ e_1 \ldots e_n$ in a function f, we define $G_c = \begin{bmatrix} G_c^{\uparrow} \\ G_c^{\downarrow} \end{bmatrix}$, where

$$\begin{aligned}
G_c^{\uparrow} &= (\{f^{(1)}, \ldots, f^{(m)}\}, \{g^{(1)}, \ldots, g^{(n)}\}, E^{\uparrow}) \\
G_c^{\downarrow} &= (\{f^{(1)}, \ldots, f^{(m)}\}, \{g^{(1)}, \ldots, g^{(n)}\}, E^{\downarrow}) \\
E^{\uparrow} &= \{ f^{(i)} \xrightarrow{\delta} g^{(j)} \mid \delta(f^{(i)}) \in \mathcal{E}^{\uparrow}[\![e_j]\!](\text{↥}(f^{(1)}), \ldots, \text{↥}(f^{(m)})) \} \\
E^{\downarrow} &= \{ f^{(i)} \xrightarrow{\delta} g^{(j)} \mid \delta(f^{(i)}) \in \mathcal{E}^{\downarrow}[\![e_j]\!](\text{↧}(f^{(1)}), \ldots, \text{↧}(f^{(m)})) \}
\end{aligned}$$

—In the ensuing text, we let $\mathcal{G}$ denote the set of size-change graphs for program p, defined by

$$\mathcal{G} = \{G_c \mid\ c : g\ e_1 \ldots e_n \text{ is a call in } p\},$$

5.3 Informal safety discussion

To illustrate the meanings of the size descriptions, here are some possible sets the size approximation operators could yield, where $\vec{\Delta}^{\downarrow}_{\mathbf{id}} = (\{\text{↧}(x_1)\}, \ldots, \{\text{↧}(x_n)\})$ is the identity "must-decrease" description, and $\vec{\Delta}^{\uparrow}_{\mathbf{id}} = (\{\text{↥}(x_1)\}, \ldots, \{\text{↥}(x_n)\})$ is the identity "may-increase" description[16].

Expression e	$\mathcal{E}^{\downarrow}[\![e]\!]\vec{\Delta}^{\downarrow}_{\mathbf{id}}$	$\mathcal{E}^{\uparrow}[\![e]\!]\vec{\Delta}^{\uparrow}_{\mathbf{id}}$
`cons` 42 17	$\{\}$	$\{\}$
`cons` x_1 x_2	$\{\}$	$\{\uparrow(x_1), \uparrow(x_2)\}$
`cdr` x_1	$\{\downarrow(x_1)\}$	$\{\underline{\underline{\uparrow}}(x_1)\}$
`minimum` x_1 x_2	$\{\overline{\downarrow}(x_1), \overline{\downarrow}(x_2)\}$	$\{\underline{\underline{\uparrow}}(x_1), \underline{\underline{\uparrow}}(x_2)\}$
if x_1 **then** `car` x_2 **else** `cdr` x_2	$\{\downarrow(x_2)\}$	$\{\underline{\underline{\uparrow}}(x_2)\}$
if x_1 **then** x_2 **else** `cons` 1 x_3	$\{\}$	$\{\underline{\underline{\uparrow}}(x_2), \uparrow(x_3)\}$
if `length` $x_1 < 42$ **then** x_1 **else** `cons` 17 19	$\{\}$	$\{\underline{\underline{\uparrow}}(x_1)\}$
if `length` $x_1 <$ `length` x_2 **then** x_1 **else** `cdr` x_2	$\{\}$	$\{\underline{\underline{\uparrow}}(x_1), \underline{\underline{\uparrow}}(x_2)\}$

The ',' in the sets generated by $\mathcal{E}^{\downarrow}$ is read as 'and,' while the ',' in the sets generated by $\mathcal{E}^{\uparrow}$ is read as 'or.' The informal interpretation of the approximation sets are

$\{\}$	no size-change information is given
$\{\downarrow(x_1)\}$	the value of e is definitely less than the value of x_1
$\{\underline{\underline{\uparrow}}(x_1)\}$	the values of e may be unbounded, if the values of x_1 are unbounded
$\{\overline{\downarrow}(x_1), \overline{\downarrow}(x_2)\}$	the value of e is no greater than that of x_1 and no greater than that of x_2
$\{\uparrow(x_1), \uparrow(x_2)\}$	the value of e may exceed that of x_1 or x_2 for infinitely many values of x_1 or x_2
$\{\underline{\underline{\uparrow}}(x_2), \uparrow(x_3)\}$	the values of e may be unbounded if the values of x_2 are unbounded; and may exceed x_3 for infinitely many x_3
$\{\underline{\underline{\uparrow}}(x_1), \underline{\underline{\uparrow}}(x_2)\}$	the values of e may be unbounded for unbounded values of x_1 or x_2.

Principles. Safety of the "must-decrease" analysis $\mathcal{E}^{\downarrow}$ is easier to define, and rather simpler to implement than $\mathcal{E}^{\uparrow}$. For any expression e, $\mathcal{E}^{\downarrow}[\![e]\!]$ will depend on the parameters seen in e, and the operators applied to them. For example, $\mathcal{E}^{\downarrow}[\![$`cons` e_1 $e_2]\!] = \{\}$, and

$$\mathcal{E}^{\downarrow}[\![\texttt{cdr}\ \ e]\!] = \{\downarrow(x) \mid \overline{\downarrow}(x) \in \mathcal{E}^{\downarrow}[\![e]\!] \text{ or } \downarrow(x) \in \mathcal{E}^{\downarrow}[\![e]\!]\}$$

For details, see Figure 15 in Appendix B.

The purpose of $\mathcal{E}^{\downarrow}$ is to detect expression behavior that is guaranteed always to decrease values, while $\mathcal{E}^{\uparrow}$ detects expression behavior that might increase values without bounds. Note that according to the concept of *safety* defined below, the sets of decreasing operators can safely be reduced and weakened: For instance, the expression $\{\downarrow(x_2)\}$ above could be approximated by $\{\overline{\downarrow}(x_2)\}$ or $\{\}$. Correspondingly, sets of increasing operators can safely be extended and exaggerated: the expression **if** x_1 **then** `car` x_2 **else** `cdr` x_2 could also be approximated by $\{\underline{\underline{\uparrow}}(x_1), \underline{\underline{\uparrow}}(x_2)\}$—or

[16]Subsequently we will very often use the identity descriptions, so we will sometimes just write $\mathcal{E}^{\downarrow}[\![e]\!]$ or $\mathcal{E}^{\uparrow}[\![e]\!]$, omitting $\vec{\Delta}^{\downarrow}_{\mathbf{id}}, \vec{\Delta}^{\uparrow}_{\mathbf{id}}$.

even $\{\uparrow(x_1), \uparrow(x_2)\}$. However, although these changes are safe, they will degrade the precision of the termination analysis.[17]

The approximation operators we have used for the implementation and experiments of Section 8 and proof of their safety are given in Appendix B. These operators yield rather crude approximations, but can easily be replaced by other and more precise safe approximation if desired, as long as they satisfy the safety conditions.

5.4 Safe "must-decrease" size approximations

Definition 5.4.1. (Safety of $\mathcal{E}^{\downarrow}$) A *decreasing size approximation operator* $\mathcal{E}^{\downarrow}$ is called *safe* iff both the following are true for all $e \in$ *Expression* and any infinite set of environments $\{\vec{v}_1, \vec{v}_2, \ldots\}$ Suppose $\mathcal{E}[\![e]\!]\vec{v}_1 = w_1, \mathcal{E}[\![e]\!]\vec{v}_2 = w_2, \ldots$ Then

(1) if $\bar{\downarrow}(x_i) \in \mathcal{E}^{\downarrow}[\![e]\!]$ then for all k, $w_k \notin \{\mathsf{error}, \bot\}$ implies $w_k \leq v_{ki}$

(2) if $\downarrow(x_i) \in \mathcal{E}^{\downarrow}[\![e]\!]$ then for all k, $w_k \notin \{\mathsf{error}, \bot\}$ implies $w_k < v_{ki}$

The safety of $\mathcal{E}^{\downarrow}$ is simple because we require that it give *definite* information on what the size of the value of an expression is, every time it is evaluated. Further, it is easily defined *compositionally*, so $\mathcal{E}^{\downarrow}[\![\mathtt{f}(\mathsf{e}_1, \ldots, \mathsf{e}_n)]\!]\vec{\Delta}$ is expressed only in terms of $\mathcal{E}^{\downarrow}[\![\mathsf{e}_1]\!]\vec{\Delta}, \ldots, \mathcal{E}^{\downarrow}[\![\mathsf{e}_n]\!]\vec{\Delta}$.

5.5 Safe "may-increase" size approximations

For "may-increase" dependencies $\underline{\uparrow}(x)$ or $\uparrow(x)$, one of two cases could apply:

(1) e's value is of unbounded variation, provided x is of unbounded variation. Example: $e = x$, or $e = \mathtt{cdr}\ x$ both have dependency $\underline{\uparrow}(x)$.

(2) the value of e exceeds the value of x infinitely often. Example: $e = \mathtt{cons}\ x\ y$ has dependency $\uparrow(x)$.

In this section we begin with plausible but inadequate definitions of safety of "may-increase" analysis, and then show examples to illustrate the subtlety of how to give program descriptions that are *safe* with respect to these informal readings. We examine the definitions' shortcomings, refine them, and end with a solid definition of safety of an analysis.

First try at "may-increase" size approximation. We adapt Definition 5.4.1:

Trial definition 1 (Safety of $\mathcal{E}^{\uparrow}$) An *increasing size approximation operator* $\mathcal{E}^{\uparrow}$ is called *safe* iff both the following are true for all $e \in$ *Expression* and any infinite set of environments $\{\vec{v}_1, \vec{v}_2, \ldots\}$: Suppose $\mathcal{E}[\![e]\!]\vec{v}_1 = w_1, \mathcal{E}[\![e]\!]\vec{v}_2 = w_2, \ldots$ Then

[17]The expression **if** $\mathtt{length}\ x_1 < 42$ **then** x_1 **else** $\mathtt{cons}$ 17 19 can in fact be given a more precise increasing size approximation, namely the empty set, because by inspecting the **if**-condition it can be seen that the length of the resulting value is bounded.

Similarly, the expression **if** $\mathtt{length}\ x_1 < \mathtt{length}\ x_2$ **then** x_1 **else** $\mathtt{cdr}\ x_2$ can be approximated more precisely by the decreasing dependency set $\{\bar{\downarrow}(x_1), \downarrow(x_2)\}$, which reads 'the value of e is definitely not greater than that of x_1, and definitely less than that of x_2.' Note also that either $\underline{\uparrow}(x_1)$ or $\underline{\uparrow}(x_2)$ (but not both!) could be omitted from the increasing dependency set of this expression, because unbounded increase of the omitted parameter is only possible if the remaining parameter is increased as well. This also implies that in general no "best" size approximation exists.

(1) if for all k, $w_k \notin \{\mathsf{error}, \bot\}$ implies $w_k \geq v_{ki}$, then $\ddagger(x_i) \in \mathcal{E}^{\uparrow}[\![e]\!]$
(2) if for all k, $w_k \notin \{\mathsf{error}, \bot\}$ implies $w_k > v_{ki}$ then $\uparrow(x_i) \in \mathcal{E}^{\uparrow}[\![e]\!]$

Two problems. The safety of $\mathcal{E}^{\uparrow}$ is complicated by two facts. First, a description $\ddagger(x)$ by part (1) above does not capture the requirement that unbounded x implies unbounded e, e.g., $e \equiv$ `cdr` x should have description $\ddagger(x)$.

Another problem is that an expression need not be classified as "increasing" if its value is greater than its arguments for only a finite set of argument values: Consider the function

```
lookupbody f p = if p = nil then
                    cons 'error: (cons 'undefined (cons 'function nil))
                 else if f = caar p then caddar p
                                    else lookupbody f (cdr p)
```

The value of `lookupbody` f p might be longer than the value of f or p, due to the first **if** branch. However, if we classify it as $\{\uparrow(f), \uparrow(p)\}$ we will not be able to detect that parameter `e` in the interpreter example of Figure 2 is of bounded variation, as `e` would seem to be increasingly dependent on itself at call site $\boxed{9}$. The point is though that even if this expression participates in a program loop where the length of f or p increases, the length of its value will always be bounded by p, and we need only classify it as $\{\ddagger(p)\}$.

We now try to remedy these problems:

Trial definition 2 (SAFETY OF $\mathcal{E}^{\uparrow}$) An *increasing size approximation operator* $\mathcal{E}^{\uparrow}$ is called *safe* iff both the following are true for all $e \in$ *Expression* and any infinite set of environments $\{\vec{v}_1, \vec{v}_2, \ldots\}$: Suppose $\mathcal{E}[\![e]\!]\vec{v}_1 = w_1, \mathcal{E}[\![e]\!]\vec{v}_2 = w_2, \ldots$ and every $w_i \notin \{\mathsf{error}, \bot\}$. Then

(1) if $|\{v_{ji}\}| = \infty$ and $|\{w_j\}| = \infty$, then $\ddagger(x_i) \in \mathcal{E}^{\uparrow}[\![e]\!]$.
(2) if $|\{v_{ji}\}| = \infty$ and $|\{j \mid w_j > v_{ji}\}| = \infty$, then $\uparrow(x_i) \in \mathcal{E}^{\uparrow}[\![e]\!]$.

Bounded variation. The familiar program:

```
double x      = dblplus x 0
dblplus u v   = if u ≤ 0 then v else dblplus (u − 1) (v + 2)
```

is a subtle "may-increase" example involving unboundedness. Safe size descriptions of $u - 1$, $v + 2$ are clearly $\ddagger(u)$ and $\uparrow(v)$. Finding a safe size description of a call `dblplus` u v involves finding the *net effect* of calling function `dblplus`. The appropriate safe description of `dblplus` u v is $\{\uparrow(u), \uparrow(v)\}$, which leads to the expected $\{\uparrow(x)\}$ as size description of `double` x (consistent with our *a priori* knowledge of the effect of `double`).

At first sight description $\{\uparrow(u), \uparrow(v)\}$ seems strange, since the value of `dblplus` u v is computed from v alone (by adding 2 repeatedly). However *the number of times* 2 is added depends on u. Consequence: if either u *or* v is unbounded, then the value of `dblplus` u v will also be unbounded.

We desire result description $\mathcal{E}^{\uparrow}[\![\texttt{dblplus}\ u\ v]\!](\vec{\Delta}^{\uparrow}_{\mathbf{id}}) = \{\uparrow(u), \uparrow(v)\}$. Unfortunately this cannot be obtained compositionally: The recursive call gives $\mathcal{E}^{\uparrow}[\![u - 1]\!](\vec{\Delta}^{\uparrow}_{\mathbf{id}}) = \{\ddagger(u)\}$, $\mathcal{E}^{\uparrow}[\![v + 2]\!](\vec{\Delta}^{\uparrow}_{\mathbf{id}}) = \{\uparrow(v)\}$ and the `dblplus` result value equals `v`'s final value—so the indirect dependency of the value of `dblplus` $u\ v$ on u is not registered at all.

A second idea is to account for indirect dependencies. Notation: for any tuple $\vec{v} = (v_1, \ldots, v_n)$ of values of e's free variables, let $\vec{v}\big|_{v_j=u}$ be $\vec{v}$ after replacing the j-th component by u. The following is formulated more simply, in terms of variations on a given tuple rather than infinite sequences of tuples.

Intuitively, $\mathcal{E}^{\uparrow}$ is safe if for all i we have that $\uparrow(x_i) \in \mathcal{E}^{\uparrow}[\![e]\!]\vec{v}$ if there is a tuple $\vec{v}$ such that the value of e is greater than v_i for infinitely many v_i, keeping the remaining parameters in $\vec{v}$ fixed. This could be stated formally by the following

Trial definition 3 (Safety of $\mathcal{E}^{\uparrow}$) An *increasing size approximation operator* $\mathcal{E}^{\uparrow}$ is called *safe* iff both the following are true for all $e \in \mathit{Expression}$:

(1) If there exists $\vec{v} = (v_1, \ldots, v_n) \in \mathit{Value}^n$ such that for all $K \in \mathit{Value}$

$$\mathcal{E}[\![e]\!]\vec{v}\big|_{v_j=v'} = v \neq \bot \wedge v > K \text{ for infinitely many } v' \in \mathit{Value},$$

then $\underline{\underline{\uparrow}}(x_j) \in \mathcal{E}^{\uparrow}[\![e]\!]$.

(2) If there exists $\vec{v} = (v_1, \ldots, v_n) \in \mathit{Value}^n$ such that

$$\mathcal{E}[\![e]\!]\vec{v}\big|_{v_j=v'} = v \neq \bot \wedge v > v' \text{ for infinitely many } v' \in \mathit{Value},$$

then $\uparrow(x_j) \in \mathcal{E}^{\uparrow}[\![e]\!]$.

This overcomes the two preceding problems (constants, and dependence of `dblplus` $u\ v$ on u.)

However, we wish the approximation to be safe for *all* cases, including cases like $e \equiv 1 + \mathit{min}\ x_1\ x_2$, where $\mathcal{E}[\![e]\!](v_1, v_2) > v_1$ for only finitely many v_1 when v_2 is fixed but $\mathcal{E}\ [\![e]\!]\ (v_1, v_2) > v_1$ for infinitely many v_1 if v_1 and v_2 are increased together. In this case $\uparrow(x_1)$ or $\uparrow(x_2)$ must be in $\mathcal{E}^{\uparrow}[\![e]\!]$; otherwise $\mathcal{E}^{\uparrow}[\![e]\!]$ would not reflect the fact that the value of e is greater than that of x_1 or x_2 for infinitely many (x_1, x_2). This is captured by considering all subsets of the parameters.

For given $\vec{v} \in \mathit{Value}$ and $X \subseteq \{x_1, \ldots, x_n\}$ we define the set $V(\vec{v}, X) = \{(u_1, \ldots, u_n) \mid \text{if } x_i \in X \text{ then } u_i \in \mathit{Value} \text{ else } u_i = v_i\}$, that is, fixing the values of all parameters except those mentioned in X.

Definition 5.5.1. (Safety of $\mathcal{E}^{\uparrow}$) An *increasing size approximation operator* $\mathcal{E}^{\uparrow}$ is called *safe* iff both the following conditions are true for all $e \in \mathit{Expression}$ and $X \subseteq \{x_1, \ldots, x_n\}$, where $\{x_1, \ldots, x_n\}$ are the parameters of the function in which e occurs:

(1) If for all $K \in \mathit{Value}$ there exists $\vec{v} = (v_1, \ldots, v_n) \in \mathit{Value}^n$ such that

$$\mathcal{E}[\![e]\!]\vec{v}\,' = v \neq \bot \wedge v > K \text{ for infinitely many } \vec{v}\,' \in V(\vec{v}, X),$$

then $\delta(x_i) \in \mathcal{E}^{\uparrow}[\![e]\!]$ for some $x_i \in X$ and $\delta \in \{\uparrow, \underline{\underline{\uparrow}}\}$.

(2) For all $x_j \in X$, if there exists $\vec{v} = (v_1, \ldots, v_n) \in \mathit{Value}^n$ such that

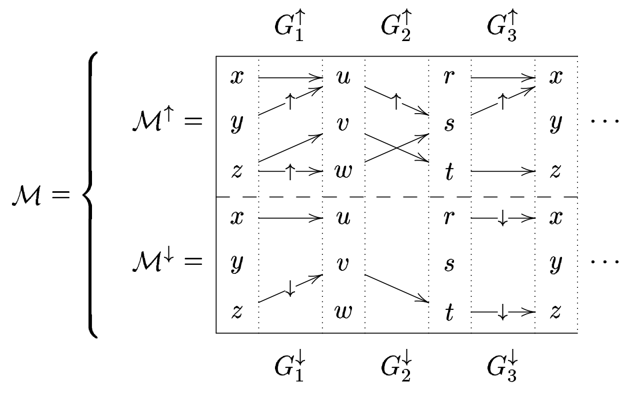
$$\mathcal{E}[\![e]\!]\vec{v}\,' = v \neq \bot \wedge v > v'_j \text{ for infinitely many } \vec{v}\,' \in V(\vec{v}, X),$$

then $\uparrow(x_i) \in \mathcal{E}^{\uparrow}[\![e]\!]$ for some $x_i \in X$.

5.6 Size-change graphs and multipaths

Definition 5.6.1. (MULTIPATH) Given a well-formed call sequence $cs = c_1c_2\ldots$, *multipath* $\mathcal{M}(cs)$ is the sequence $G_{c_1}, G_{c_2}, \ldots$ of size change graphs. Given a multipath $\mathcal{M} = G_1, G_2, \ldots$, we often consider only the increasing or decreasing part $\mathcal{M}^\delta = G_1^\delta, G_2^\delta, \ldots$ for $\delta \in \{\uparrow, \downarrow\}$.

A multipath can be viewed as one big graph obtained by concatenating the size-change graphs:



Thus, we make the following

Definition 5.6.2. (MULTIPATH SUBGRAPH)

—We define a *subgraph of a multipath* $\mathcal{M}$ (or $\mathcal{M}^\delta$) to be a connected subgraph of the graph created by the concatenation of the size-change graphs.

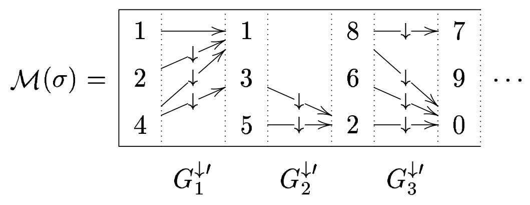
—A *maximal subgraph* is a subgraph $\widetilde{G}$ that is not a subgraph of any other subgraph $\widetilde{G}' \neq \widetilde{G}$.

—The *length* of a subgraph is the length of the longest directed path in the subgraph (possibly infinite).

Multipaths can be used not only to describe size relations between program parameters, but also to describe an actual computation. Without causing ambiguity, we can overload the $\mathcal{M}$ operator:

Definition 5.6.3. (STATE TRANSITION MULTIPATH) Given a state transition sequence $\sigma = (f_0, \vec{v}_0) \xrightarrow{c_1} (f_1, \vec{v}_1) \xrightarrow{c_2} (f_3, \vec{v}_3) \xrightarrow{c_3} \cdots$, we let $\mathcal{M}(\sigma)$ denote the multipath $G_1^{\downarrow\prime}, G_2^{\downarrow\prime}, \ldots$, where each $G_k^{\downarrow\prime}$ is obtained from $G_{c_k}^\downarrow$ by replacing the parameters by their actual values (given by $\vec{v}_{k-1}$ and $\vec{v}_k$), removing all the edges, and then adding new edges $v_{(k-1)i} \xrightarrow{\downarrow} v_{kj}$ whenever $v_{(k-1)i} > v_{kj}$ and $v_{(k-1)i} \xrightarrow{\overline{\downarrow}} v_{kj}$ whenever $v_{(k-1)i} = v_{kj}$.

One multipath $\mathcal{M}(\sigma)$ with $\mathit{calls}(\sigma) = c_1c_2\ldots$ from the preceding example in might look like this:



where any $\updownarrow$ labels have been omitted for readability.

Note that there is a bijective correspondence between the nodes of $\mathcal{M}(\sigma)$ and $\mathcal{M}^{\delta}(cs)$, where $cs = \mathit{calls}(\sigma)$. In the following, we will therefore consider them identical, and speak about *the value of a node n in $\mathcal{M}^{\delta}(cs)$*, meaning the value of the corresponding node in $\mathcal{M}(\sigma)$. Further, the safety definition of $\mathcal{E}^{\downarrow}$ (cf. Definition 5.4.1) implies the following lemma for the arcs:

LEMMA 5.6.4. (ARC CORRESPONDENCE FOR $\mathcal{M}^{\downarrow}$) *Given state transition sequence $\sigma = (f_0, \vec{v}_0) \xrightarrow{c_1} \cdots$, if there is an arc $n \xrightarrow{\delta} n'$ in $\mathcal{M}^{\downarrow}(cs)$, where $cs = \mathit{calls}(\sigma)$, there is a similar arc $v \xrightarrow{\delta'} v'$ in $\mathcal{M}(\sigma)$ between the values corresponding to n and n'. Furthermore, if $\delta = \downarrow$ then $\delta' = \downarrow$.*

Definition 5.6.5. (THREADS) Given a multipath $\mathcal{M}$, a *thread* is a subgraph in which each node has at most one ingoing and one outgoing arc.

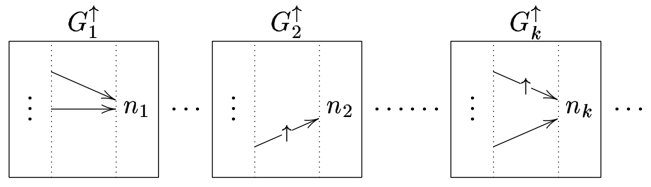
Combining this definition with Lemma 5.6.4, we obtain

COROLLARY 5.6.6. (THREAD CORRESPONDENCE) *If $\mathcal{M}^{\downarrow}(cs)$ has a thread τ with M $\downarrow$-labels and $cs = \mathit{calls}(\sigma)$, then $\mathcal{M}(\sigma)$ has a corresponding thread τ' with at least M $\downarrow$-labels.*

A multipath $\mathcal{M}^{\uparrow}(cs)$ conveys information about the boundedness of the corresponding values in $\mathcal{M}(\sigma)$:

LEMMA 5.6.7. (UNBOUNDED PREDECESSORS) *Given a state transition sequence σ with call sequence cs, there exists a K such that if a sequence of nodes $n_1, n_2, \ldots$ in $\mathcal{M}(\sigma)$ has an unbounded sequence of values $v_1, v_2, \ldots$ all greater than K, there exist infinitely many of these nodes $n_{k_1}, n_{k_2}, \ldots$ which have at least one immediate predecessor $p_{k_1}, p_{k_2}, \ldots$, and the sequence of values $w_{k_1}, w_{k_2}, \ldots$ of the nodes in $\mathcal{M}(\sigma)$ corresponding to these predecessors will be unbounded and greater than K. Furthermore, for all j, if there is no arc $p_{k_j} \xrightarrow{\uparrow} n_{k_j}$, then $w_{k_j} \geq v_{k_j}$.*

PROOF. Given this σ, assume the sequence $v_1, v_2, \ldots$ is unbounded, and consider the set of the size-change graphs in $\mathcal{M}^{\uparrow}(cs)$ that contain nodes $n_1, n_2, \ldots$ as right-side elements:



As $\mathcal{G}$ is finite, there must exist at least one $G^{\uparrow}$ with a right-side node n, and an infinite subsequence $k_1, k_2, \ldots$ for which $G^{\uparrow} = G^{\uparrow}_{k_1} = G^{\uparrow}_{k_2} = \cdots$, where $v_{k_1}, v_{k_2}, \ldots$ is unbounded.

For each such $G^{\uparrow}$, all of these values are computed by the same expression e, based on (a subset of) the values of the constants found in the program text, and the values of the immediately preceding nodes, i.e. the left-hand nodes in $G^{\uparrow}$. For any K, there are infinitely many $v_{k_j} > K$, and as the computation of the expression is deterministic, this is only possible if some parameter $x_{i'}$ in e takes on infinitely many different values $w'_1, w'_2, \ldots$, so by the second criterion of the safety of $\mathcal{E}^{\uparrow}$ (cf. Definition 5.5.1), $\delta(x_{i'}) \in \mathcal{E}^{\uparrow}[\![e]\!]$ for some $\delta \in \{\uparrow, \ddagger\}$, which implies that there exists an arc $x_{i'} \xrightarrow{\delta} n_{k_j}$ in $G^{\uparrow}$.

Finally, if there exist infinitely many w'_{k_j} for which $w'_{k_j} < v_{k_j}$ for some $x_{i'}$, the first criterion of the safety of $\mathcal{E}^{\uparrow}$ requires that the arc $x_i \xrightarrow{\delta} n_{k_j}$ be labeled $\delta = \uparrow$ for some x_i (possibly $i = i'$) whose sequence of values $w_1, w_2, \ldots$ is unbounded. So the desired predecessors are the (infinitely many) nodes $p_{k_1}, p_{k_2}, \ldots$ corresponding to x_i for which $w_{k_j} \geq v_{k_j}$. □

Definition 5.6.8. (ORIGIN NODE) A node in a multipath $\mathcal{M}^{\uparrow}$ which has no predecessors is called an *origin node.*

Given this definition, a consequence of the contraposition of Lemma 5.6.7 is the following

COROLLARY 5.6.9. (ORIGIN NODE BOUNDEDNESS) *Given a state transition sequence σ with call sequence cs, there exists a value K such that for any origin node in $\mathcal{M}^{\uparrow}(cs)$, the value of the corresponding node in $\mathcal{M}(\sigma)$ is bounded by K.*

We now use this corollary and the preceding lemma to show that obtaining a sequence of unbounded values in a state transition sequence σ requires that there exist paths with unboundedly many $\uparrow$-labeled arcs in $\mathcal{M}^{\uparrow}(cs)$:

LEMMA 5.6.10. *Let a state transition sequence σ be given with call sequence cs and assume for any K there exists a set of finite subgraphs $\widetilde{G}_1, \widetilde{G}_2, \ldots$ of $\mathcal{M}(\sigma)$ such that for each subgraph $\widetilde{G}_k$ there exists a right hand node n_k in the final (rightmost) size change graph, where the sequence of values of $n_1, n_2, \ldots$ is unbounded and greater than K.*

Then it follows for any $M \in \mathbb{N}$ that there exists $k \in \mathbb{N}$ such that there is a path $\xrightarrow{\delta} \cdots \xrightarrow{\delta} n_k$ in $\widetilde{G}^{\uparrow}_k$, the graph in $\mathcal{M}^{\uparrow}(cs)$ corresponding to $\widetilde{G}_k$, which has at least M $\uparrow$ labels.

PROOF. by induction on M: For $M = 0$, the lemma is trivially true.

For $M > 0$, choose K greater than the upper bound on the origin nodes such that it satisfies Lemma 5.6.7. That lemma then states that for infinitely many k_j there exist predecessors p_{k_j} of n_{k_j} where the sequence of values of $p_{k_1}, p_{k_2}, \ldots$ is unbounded. If infinitely many arcs $p_{k_j} \xrightarrow{\delta} n_{k_j}$ in $\mathcal{M}^{\uparrow}(cs)$ are labeled $\delta = \uparrow$, the induction hypothesis proves the desired property.

Otherwise, we proceed as follows: For each k_j where the arc is labeled $\ddagger$, we change $\widetilde{G}_{k_j}$ by removing the final size change graph. This way, we get a new sequence $\widetilde{G}_1, \widetilde{G}_2, \ldots$ where some of the subgraphs are shorter than the original ones. Further, Lemma 5.6.7 ensures that for each subgraph, either the final size change graph in $\mathcal{M}^{\uparrow}(cs)$ contains an arc labeled $\uparrow$, or there exists a right hand node n_k, where the sequence of values of $n_1, n_2, \ldots$ is unbounded and greater than K.

This procedure is applied repeatedly, and there are two possibilities: Either we stop after finitely many iterations when infinitely many of the subgraphs end with a size change graph that has a corresponding $\uparrow$-labeled arc in $\mathcal{M}^{\uparrow}(cs)$. Otherwise, the process continues indefinitely, and as each subgraph, as well as $\mathcal{G}$, is finite, this produces an infinite sequence of subgraphs where each final size change graph has a corresponding $\uparrow$-labeled arc in $\mathcal{M}^{\uparrow}(cs)$ (because the value of node n_{k_j} is greater than K, preventing it from being an origin node). An example of this process for subgraphs composed of three size change graphs G_1, G_2, G_3 whose corresponding graphs in $\mathcal{M}^{\uparrow}(cs)$ have no $\uparrow$-arcs and some which do have $\uparrow$-arcs, commonly denoted $\hat{G}$, is shown in Figure 6 (the size change graphs that are removed in each step are marked with boxes). Finally, the induction hypothesis proves the desired property. □

Lemma 5.6.11. *Given a state transition sequence σ, assume for any M' that there exists a subgraph in $\mathcal{M}^{\uparrow}(cs)$ that has a path $\longrightarrow \overset{\uparrow}{\cdots} \longrightarrow$ with at least M' $\uparrow$ labels. Then for any M there exists a parameter x and a subgraph in $\mathcal{M}^{\uparrow}(cs)$ with a path $x \longrightarrow \overset{\uparrow}{\cdots} \longrightarrow x \longrightarrow \overset{\uparrow}{\cdots} \longrightarrow x \longrightarrow \overset{\uparrow}{\cdots} \longrightarrow x$ that passes through x at least M times. Furthermore, each subpath $x \longrightarrow \overset{\uparrow}{\cdots} \longrightarrow x$ of this path has at least one $\uparrow$ label.*

Proof. Let N be the number of parameters in p. A path with at least N $\uparrow$ labels will contain at least one path $x \longrightarrow \overset{\uparrow}{\cdots} \longrightarrow x$ for some parameter x, and it follows that choosing a path with at least MN $\uparrow$ labels will contain the desired path. □

Definition 5.6.12. (Size-change graph composition) Given two size-change graphs $G_1^\delta = (var(f), var(g), E_1)$ and $G_2^\delta = (var(g), var(h), E_2)$, we define their *composition* by $G_1^\delta; G_2^\delta = (var(f), var(h), E)$, where $x_i \xrightarrow{\delta'} z_k \in E$ iff $(x_i \xrightarrow{\delta_1} y_j) \in E_1$ and $(y_j \xrightarrow{\delta_2} z_k) \in E_2$, and

$$\delta' = \begin{cases} \uparrow, \text{ if } \delta_1 = \uparrow \vee \delta_2 = \uparrow \\ \underset{=}{\uparrow}, \text{ if } \delta_1 = \underset{=}{\uparrow} \wedge \delta_2 = \underset{=}{\uparrow} \\ \downarrow, \text{ if } \delta_1 = \downarrow \vee \delta_2 = \downarrow \\ \overset{=}{\downarrow}, \text{ if } \delta_1 = \overset{=}{\downarrow} \wedge \delta_2 = \overset{=}{\downarrow} \end{cases}.$$

For $G_1 = \begin{bmatrix} G_1^{\uparrow} \\ G_1^{\downarrow} \end{bmatrix}$ and $G_2 = \begin{bmatrix} G_2^{\uparrow} \\ G_2^{\downarrow} \end{bmatrix}$ we define $G_1; G_2 = \begin{bmatrix} G_1^{\uparrow}; G_2^{\uparrow} \\ G_1^{\downarrow}; G_2^{\downarrow} \end{bmatrix}$.

Definition 5.6.13. For a well-formed nonempty call sequence $cs = c_1 \ldots c_n$, we define the size-change graph for cs, denoted by G_{cs}, as $G_{c_1}; \ldots; G_{c_n}$.

It is straightforward to show that composition is associative. Furthermore, the arcs in the composition are an approximation of the threads in the composed size-change graphs:

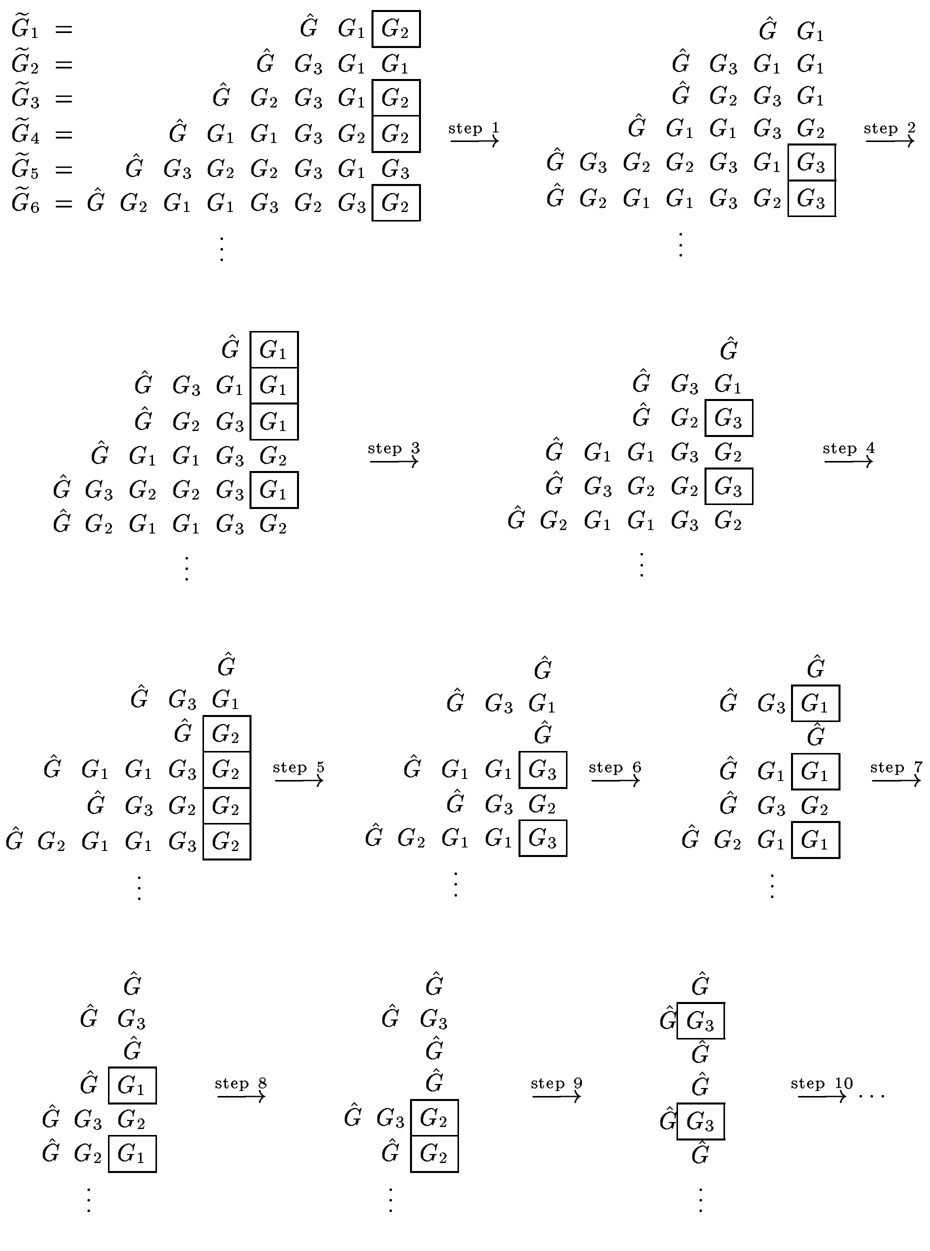


Fig. 6. Generating a sequence with infinitely many subgraphs ending in a $\hat{G}$ graph. In each step we remove the same last, non-$\hat{G}$ size-change graph from infinitely many subgraphs.

LEMMA 5.6.14. (COMPOSITION SOUNDNESS) *$G_1;\ldots;G_n$ has an arc $x \xrightarrow{\delta} u$ iff there is a thread $x \longrightarrow \cdots \longrightarrow u$ all the way along multipath $\mathcal{M} = G_1,\ldots,G_n$. Furthermore, $\delta = \uparrow$ or $\delta = \downarrow$ iff there is a thread with at least one $\uparrow$ or $\downarrow$ label, respectively.*

PROOF. by induction on n, using the definition of composition. □

6. QUASITERMINATION ANALYSIS

Now that we have defined the size-change graphs that express parameter changes along a call sequence, we can reason about the termination properties of all possible call sequences during evaluation. We start by defining some properties of the program parameters that describe *quasitermination,* originally proposed by Holst [1991], in which the program might not terminate, but the parameters will only take on a bounded set of values during any program run.

6.1 Quasitermination definitions

Definition 6.1.1. (BOUNDED VARIATION) For any state transition sequence σ the function *bounded*$(\sigma) = \{x \mid (x,V) \in \mathcal{V}(\sigma) \wedge |V| < \infty\}$ defines the set of parameters of bounded variation of that transition sequence. A parameter x is said to be of bounded variation, written $BV(x)$, iff $x \in$ *bounded*(σ) for all σ.

Considering Definition 3.4.1 we get the following

LEMMA 6.1.2. (QUASITERMINATION) *Program p is quasiterminating iff all program parameters are of bounded variation.*

6.2 Quasitermination detection

We now employ a division β : *Varname* $\rightarrow \{S, ?\}$ for dividing the parameters into definitely of bounded variation (S) or as yet undecided (?). A division that ensures termination is then constructed in the following way: First we classify all parameters as $\beta_0(x) = ?$. Then we iteratively apply the *bounded anchoring* step, which changes a number of ?-classifications into S-classifications. When no more ?-classifications can be changed, we are done.

Before considering bounded anchoring, we first give a definition of the set $\mathcal{S}$ of size-change graphs and their composition along well-formed call sequences, and a definition of an equivalence class for parameters that are mutually dependent on each other.

Definition 6.2.1. (SUMMARIZING PATH EFFECTS) $\mathcal{S} = \{G_{cs} \mid cs \text{ is well-formed}\}$

Note that we now summarize *effects,* lumping together *different* call sequences that have the *same* size-change effect!

Definition 6.2.2. (BINDING-TIME EQUIVALENCE CLASSES) We write $x \equiv y$ if there exists $G \in \mathcal{S}$ with an edge $x \longrightarrow y$ in $G^{\uparrow}$ and a $G' \in \mathcal{S}$ with an edge $y \longrightarrow x$ in $G'^{\uparrow}$, and define the equivalence class $[x] = \{y \mid x \equiv y\}$.

Definition 6.2.3. (BOUNDED ANCHORING) Let there be given a division β_q : *Varname* $\rightarrow \{S, ?\}$ and an equivalence class $[y']$ for which $\beta_q(y) = ?$ for all $y \in [y']$. We define β_{q+1} by checking the following two conditions for all $y \in [y']$:

(1) For all $G^{\uparrow} \in \mathcal{S}$ which has an arc $x \to y$, $\beta_q(x) = S$ or $[x] = [y']$.

(2) For all $G : f \to f \in \mathcal{S}$ where $G = G;G$ and $G^{\uparrow}$ has an arc $y \xrightarrow{\uparrow} y$, $G^{\downarrow}$ has an arc $z \xrightarrow{\downarrow} z$, and $\beta_q(z) = S$.

If both the conditions are true for all $y \in [y']$ then $\beta_{q+1}(x) = \begin{cases} S, & \text{if } x \in [y'] \\ \beta_q(x), & \text{otherwise} \end{cases}$; otherwise, $\beta_{q+1} = \beta_q$.

In the second condition above, we say that y *is anchored in* z *(for* G*).*

THEOREM 6.2.4. (CORRECTNESS OF BOUNDED ANCHORING) *If* p *quasiterminates for* β_q*, then* p *quasiterminates for* β_{q+1}.

PROOF. Assume p is quasiterminating for β_q, and suppose p is not quasiterminating for β_{q+1}, i.e. for some $f^{(i)}$ with $\beta_{q+1}(f^{(i)}) = S$ and some (infinite state transition sequence) σ we have $f^{(i)} \notin \textit{bounded}(\sigma)$. Thus we can find an ascending sequence of values $u_1, u_2, \ldots$ such that $u_k = (\vec{v}_{j_k})_i$ (the ith component of the kth element of a subsequence of $\vec{v}_0, \vec{v}_1, \ldots$) where $\sigma = (f_0, \vec{v}_0) \xrightarrow{c_1} (f_1, \vec{v}_1) \xrightarrow{c_2} \cdots$ and $f = f_{j_1} = f_{j_2} = \cdots = f_{j_k} = \cdots$. Obviously, as p is quasiterminating for β_q and $f^{(i)} \notin \textit{bounded}(\sigma)$, we have $\beta_q(f^{(i)}) \neq S$, i.e. $f^{(i)} \in [y']$ in the bounded anchoring step from β_q to β_{q+1}.

Choose a finite $M \in \mathbb{N}$ such that $M > |\{v \in V \mid (x, V) \in \mathcal{V}(\sigma) \wedge \beta_q(x) = S\}|$, that is, greater than any value ever assigned to a bounded parameter; this is possible because p is quasiterminating for β_q.

The rest of the proof is almost identical to that of the central theorem of the graph-based approach to termination detection presented by Lee et al. [2001, Theorem 4]:

Define a *2-set* to be a 2-element set $\{t, t'\}$ of positive integers. Without loss of generality, $t < t'$. Let $m > M$ and a sequence of size change graphs $G_1, \ldots, G_m$ be given, and define for each $G \in \mathcal{S}$

$$P_G = \{(t, t') \mid G = G_t; G_{t+1}; \ldots; G_{t'-1}\}$$

The set $\{P_G \mid G \in \mathcal{S}\}$ is a family of classes: they are mutually disjoint, every 2-set $\{t, t'\}, t, t' \leq m$ belongs to exactly one of them, and it is finite, since $\mathcal{S}$ is finite.

By the Finite Ramsey's Theorem [Ramsey 1930] given in Appendix A, we can compute an m such that there exists a set T of $M + 1$ indexes such that all 2-sets $\{t, t'\}$ with $t, t' \in T$ are in the same class P_{G°. Thus we have

$$G^\circ = G_{t_1}; \ldots; G_{t_3 - 1} = (G_{t_1}; \ldots; G_{t_2 - 1}); (G_{t_2}; \ldots; G_{t_3 - 1}) = G^\circ; G^\circ$$

and

$$G^\circ = G_{t_1}; \ldots; G_{t_{M+1} - 1} = (G_{t_1}; \ldots; G_{t_2 - 1}); \ldots; (G_{t_M}; \ldots; G_{t_{M+1} - 1}) = G^\circ; \ldots; G^\circ.$$

As the sequence $u_1, u_2, \ldots$ is unbounded, we can combine Lemmas 5.6.10 and 5.6.11 to find a path $x \longrightarrow^+ x$ with at least m $\uparrow$ labels. As $m > M$ and all $x \notin [y']$ with $x \longrightarrow y$ for some $y \in [y']$ are bounded (by the first criterion of bounded anchoring), we have $x \in [y']$. Dividing this path into m sections $x \longrightarrow^+ x \cdots x \longrightarrow^+ x$ such that each section has at least one $\uparrow$ label, and composing all the size change graphs in section k into G_k, we get the sequence of m size change graphs required above:

$$G'_1; \ldots; G'_{k_1-1}; \quad \cdots \quad ; G'_{k_{m-1}}; \ldots; G'_{k_m-1} = G_1; \ldots; G_m$$

As each section $G'^{\uparrow}_{k_i}; \ldots; G'^{\uparrow}_{k_{i+1}-1}$ has a thread with at least one $\uparrow$ label, by Lemma 5.6.14 $G^{\circ\uparrow}$ has an arc with an $\uparrow$ label, so by the second criterion of bounded anchoring, $G^{\circ\downarrow}$ has an arc $z \xrightarrow{\downarrow} z$, where $\beta_q(z) = S$. Thus by Lemma 5.6.14 each section $G^{\downarrow}_{k_i}; \ldots; G^{\downarrow}_{k_{i+1}-1}$ has an S-thread with at least one $\downarrow$ label $z \longrightarrow \overset{\downarrow}{\cdots} \longrightarrow z$. By Lemma 5.6.6, this implies that σ has a thread $z \longrightarrow \overset{\downarrow}{\cdots} \longrightarrow z$ with at least M $\downarrow$ labels. But this contradicts $\beta_q(z) = S$ because M was chosen to be larger than any static value, and we conclude that p *is* quasiterminating for β_{q+1}. $\square$

7. PARTIAL EVALUATION TERMINATION ANALYSIS

7.1 Syntax and semantics

Each expression of a program that is to be partially evaluated is annotated as static or dynamic, so we define the domains of *annotated expressions, definitions and programs:*

$$\begin{array}{rcl}
\mathit{AProgram} \ni p & ::= & d_1 \ldots d_n \\
\mathit{ADef} \ni d & ::= & f\ x_1^{\tau_1} \ldots x_n^{\tau_n} = e^f \\
\mathit{AExpression} \ni e & ::= & k^S \mid x_i \mid \textbf{lift}\ e_1^S \mid \textbf{if}^{\,\tau}\ e_1^\tau\ \textbf{then}\ e_2^{\tau'}\ \textbf{else}\ e_3^{\tau'} \\
 & \mid & b^\tau\ e_1^\tau \ldots e_n^\tau \mid c : f^\tau\ e_1^S \ldots e_m^S\ e_{m+1}^D \ldots e_n^D \\
\tau & \in & \{S, D\}
\end{array}$$

The **lift** construction is inserted around static expressions that are used in a dynamic context, so that the static value computed at specialization time is lifted into the residual program.

We now extend the meaning of β to represent a *binding-time division* β : *AExpression* $\cup$ *Varname* $\cup$ *CallSite* $\rightarrow \{S, D, ?\}$, which returns the binding time of its argument.

For simplicity, we consider only monovariant binding-time divisions, that is, each function parameter has the same binding time at all call sites[18], and wlog. we assume that all the static function parameters appear before the dynamic. The annotation f^τ in function calls indicates whether the call should be unfolded ($\tau = S$) at specialization time, or residualized ($\tau = D$), i.e. be a specialization point.

In the following we assume given the following domains:

$$\begin{array}{rcl}
v \in \mathit{PEValue} & = & \mathit{Value} \cup \mathit{Expression} \\
\varphi \in \mathit{Memo} & = & (\mathit{FunName} \times \mathit{Value}^*) \rightarrow \mathit{FunName} \\
\eta \in \mathit{ResProg} & = & \mathcal{P}(\mathit{Def})
\end{array}$$

and a size relation $\leq\ \in$ *Value* $\times$ *Value* which makes *Value* well-founded.

[18]Traditionally, partial evaluation has employed monovariant binding-time divisions to avoid the risk of code explosion in the residual program. Christensen et al. [2000] have shown that polyvariant binding-time divisions can be viable and desirable in some cases.

$$
\begin{array}{l}
\mathcal{PE} : \mathit{AExpression} \to \mathit{PEValue}^* \to \mathit{Memo} \to \mathit{ResProg} \\
\qquad\qquad\qquad \to (\mathit{PEValue} \times \mathit{Memo} \times \mathit{ResProg}) \cup \{\bot, \mathsf{error}\} \\
\mathcal{PE} = \mathcal{PE}_{\mathbf{e}}(\mathit{fix}(\lambda\phi.\{\mathit{f1} \mapsto \lambda v_1 \ldots v_m . \mathcal{PE}_{\mathbf{e}}\, \phi[\![e^{f1}]\!](v_1, \ldots, v_m), \ldots, \\
\qquad\qquad\qquad \mathit{fn} \mapsto \lambda v_1 \ldots v_k \,.\, \mathcal{PE}_{\mathbf{e}}\, \phi[\![e^{fn}]\!](v_1, \ldots, v_k)\}))
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{PE}_{\mathbf{e}}\, \phi[\![k^S]\!]\vec{v}\, \varphi_0\eta_0 & = & (\mathit{value}\ k, \varphi_0, \eta_0) \\
\mathcal{PE}_{\mathbf{e}}\, \phi[\![x_i]\!]\vec{v}\, \varphi_0\eta_0 & = & (v_i, \varphi_0, \eta_0) \\
\mathcal{PE}_{\mathbf{e}}\, \phi[\![\mathbf{lift}\ e_1^S]\!]\vec{v}\, \varphi_0\eta_0 & = & (\mathit{lift}\ v_1', \varphi_1, \eta_1) \\
 & & \text{where } (v_i', \varphi_i, \eta_i) = \mathcal{PE}_{\mathbf{e}}\, \phi[\![e_i]\!]\vec{v}\, \varphi_{i-1}\eta_{i-1} \\
\mathcal{PE}_{\mathbf{e}}\, \phi[\![\mathbf{if}^{\,S}\ e_1^S\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3]\!]\vec{v}\, \varphi_0\eta_0 & = & (\text{if } v_1' \text{ then } v_2' \text{ else } v_3', \varphi_3, \eta_3) \\
 & & \text{where } (v_i', \varphi_i, \eta_i) = \mathcal{PE}_{\mathbf{e}}\, \phi[\![e_i]\!]\vec{v}\, \varphi_{i-1}\eta_{i-1} \\
\mathcal{PE}_{\mathbf{e}}\, \phi[\![\mathbf{if}^{\,D}\ e_1^D\ \mathbf{then}\ e_2^D\ \mathbf{else}\ e_3^D]\!]\vec{v}\, \varphi_0\eta_0 & = & (\underline{\mathbf{if}\ v_1'\ \mathbf{then}\ v_2'\ \mathbf{else}\ v_3'}, \varphi_3, \eta_3) \\
 & & \text{where } (v_i', \varphi_i, \eta_i) = \mathcal{PE}_{\mathbf{e}}\, \phi[\![e_i]\!]\vec{v}\, \varphi_{i-1}\eta_{i-1} \\
\mathcal{PE}_{\mathbf{e}}\, \phi[\![b^S\ e_1^S \ldots e_n^S]\!]\vec{v}\, \varphi_0\eta_0 & = & (\mathit{apply}\ b\ v_1' \ldots v_n', \varphi_n, \eta_n) \\
 & & \text{where } (v_i', \varphi_i, \eta_i) = \mathcal{PE}_{\mathbf{e}}\, \phi[\![e_i]\!]\vec{v}\, \varphi_{i-1}\eta_{i-1} \\
\mathcal{PE}_{\mathbf{e}}\, \phi[\![b^D\ e_1^D \ldots e_n^D]\!]\vec{v}\, \varphi_0\eta_0 & = & (\underline{b\ v_1' \ldots v_n'}, \varphi_n, \eta_n) \\
 & & \text{where } (v_i', \varphi_i, \eta_i) = \mathcal{PE}_{\mathbf{e}}\, \phi[\![e_i]\!]\vec{v}\, \varphi_{i-1}\eta_{i-1} \\
\mathcal{PE}_{\mathbf{e}}\, \phi[\![f^S\ e_1 \ldots e_n]\!]\vec{v}\, \varphi_0\eta_0 & = & \phi\ f\ v_1' \ldots v_n'\ \varphi_n\ \eta_n \\
 & & \text{where } (v_i', \varphi_i, \eta_i) = \mathcal{PE}_{\mathbf{e}}\, \phi[\![e_i]\!]\vec{v}\, \varphi_{i-1}\eta_{i-1} \\
\mathcal{PE}_{\mathbf{e}}\, \phi[\![f^D\ e_1^S \ldots e_m^S\ e_{m+1}^D \ldots e_n^D]\!]\vec{v}\, \varphi_0\eta_0 & = & (\underline{f'\ v_{m+1}' \ldots v_n'}, \varphi', \eta') \\
\text{where } (v_i', \varphi_i, \eta_i) & = & \mathcal{PE}_{\mathbf{e}}\, \phi[\![e_i]\!]\vec{v}\, \varphi_{i-1}\eta_{i-1} \\
(f', \varphi'') & = & \text{if } (f, v_1' \ldots v_m') \in \mathit{dom}\ \ \varphi_n \\
 & & \text{then } (\varphi_n\ (f, v_1' \ldots v_m'), \varphi_n) \\
 & & \text{else } (g, \varphi_n + \{(f, v_1' \ldots v_m') \mapsto g\}) \\
g & \notin & \mathit{rg}\ \ \varphi_n \\
(e, \varphi', \eta'') & = & \text{if } (f, v_1' \ldots v_m') \in \mathit{dom}\ \ \varphi_n \text{ then } (\_, \varphi'', \eta_n) \\
 & & \text{else } \phi\ f\ v_1' \ldots v_m'\ \underline{x_1} \ldots \underline{x_{n-m}}\ \varphi''\ \eta_n \\
\eta' & = & \text{if } (f, v_1' \ldots v_m') \in \mathit{dom}\ \ \varphi_n \text{ then } \eta'' \\
 & & \text{else } \eta'' \cup \{\underline{g\ x_1 \ldots x_{n-m} = e}\}
\end{array}
$$

Fig. 7. Partial evaluation semantics. Use of 'if' are strict in the first argument, and all other functions on the right-hand side ($\mathit{lift}, (\cdot,\cdot,\cdot), \mathit{apply}$, etc.) are strict, returning $\bot$ or error if any of their arguments are $\bot$ or error

In addition to the functions used in Figure 5, we now also make use of function $\mathit{lift} : \mathit{Value} \to \mathit{Constant}$ which converts a constant value into the corresponding expression.

Partial evaluation of program p to static input $\vec{v}$ is defined by $\mathcal{PE}[\![\mathit{f1}^S\ \ \vec{x}]\!]\ \vec{v}\{\}\{\}$, where operator $\mathcal{PE}$ is defined in Figure 7; pieces of residual program syntax are underlined. Note that if all parts of an expression e are annotated as static, and all function calls in e are calls to totally static functions, $\mathcal{PE}$ performs normal, strict evaluation, so the lemmas from the preceding chapters apply to the static part of partial evaluation. The following lemma, easily proved by complete induction on the recursion depth, states that the $\mathcal{PE}$ operator only ever *adds* entries to the memoization function φ:

LEMMA 7.1.1. *If* $\{\bot, \mathsf{error}\} \not\ni \mathcal{PE}[\![e]\!]\vec{v}\varphi\eta = (v', \varphi', \eta')$ *then* $\mathit{dom}\ \ \varphi \subseteq \mathit{dom}\ \ \varphi'$.

Definition 7.1.2. (PARTIAL EVALUATION STATE TRANSITIONS) Given a binding-time division β,

(1) A *state transition* $(f, \vec{v}) \xrightarrow{c} (g, \vec{u})$ is a pair of states connected by a call $c : g\ e_1 \ldots e_n$ in the body e^f of f, such that $\vec{u} = (u_1, \ldots, u_m)$ and $\mathcal{PE}[\![e_i]\!]\vec{v}\varphi_{i-1}\eta_{i-1} = (u_i, \varphi_i, \eta_i)$ for some (φ_0, η_0).

(2) A *state transition sequence* is a (finite or infinite) sequence $\sigma = (f_0, \vec{v}_0) \xrightarrow{c_1} (f_1, \vec{v}_1) \to \cdots$, where $(f_i, \vec{v}_i) \xrightarrow{c_{i+1}} (f_{i+1}, \vec{v}_{i+1})$ is a state transition for each $i = 0, 1, \ldots$, and for any $i \neq j$ with $(f_i, \vec{v}_i) = (f_j, \vec{v}_j)$, $\beta(c_i) = S$ or $\beta(c_j) = S$.

Note that memoization is modeled by disallowing transition sequences where the same specialization point is encountered twice for the same tuple of values, cf. Point 2.

Definition 7.1.3. (S-THREADS) Given a binding-time division β, an S-thread is a thread where $\beta(x) = S$ for all nodes x in the thread.

7.2 Termination definitions

Definition 7.2.1. (TERMINATION OF PARTIAL EVALUATION) Given a binding-time division β,

—We say that partial evaluation terminates for a function f wrt. static input $\vec{v} = v_1 \ldots v_m$ iff

$$\mathcal{PE}[\![f\ x_1 \ldots x_m\ x_{m+1} \ldots x_n]\!](v_1, \ldots, v_m, \underline{x_1}, \ldots, \underline{x_{n-m}})\{\}\{\} \neq \bot.$$

—We say that partial evaluation terminates for program $p = d_1 \ldots d_n$ iff partial evaluation terminates for *f1* for all $\vec{v} \in \mathit{Value}^*$, where $d_1 \equiv (\mathit{f1}\ x_1 \ldots x_n = e^{\mathit{f1}})$.

As there are no iteration constructs in the language, and all base functions terminate, nontermination can only arise through recursive function calls. Furthermore, every dynamic call which is unfolded causes a new entry to be made in the memoization table:

LEMMA 7.2.2. (NONTERMINATION IS CAUSED BY FUNCTION CALLS) *If $\mathcal{PE}[\![e]\!]\vec{v}\varphi\eta = \bot$, then there exists a call $c : f\ e_1 \ldots e_n$ in e and $\varphi_0, \ldots, \varphi_n, \eta_0, \ldots, \eta_n, \vec{v}', \varphi'', \eta''$ with dom $\varphi \subseteq$ dom φ'' such that $\mathcal{PE}[\![f\ e_1 \ldots e_n]\!]\vec{v}\varphi_0\eta_0 = \mathcal{PE}[\![e^f]\!]\vec{v}'\varphi''\eta'' = \bot$, and $\bot \neq \mathcal{PE}[\![e_i]\!]\vec{v}\varphi_{i-1}\eta_{i-1} = (v'_i, \varphi_i, \eta_i)$ for all $i = 1, \ldots, n$. Furthermore, if $\beta(c) = D$ then $(f, v'_1 \ldots v'_m) \notin$ dom φ and $(f, v'_1 \ldots v'_m) \in$ dom φ''.*

PROOF. by structural induction on e, considering Figure 7; we note that $e \not\equiv k$ and $e \not\equiv x_i$.

Case $e \equiv$ **lift** e_1^S. Directly by the induction hypothesis.

Case $e \equiv$ **if** e_1 **then** e_2 **else** e_3. By Lemma 7.1.1 we have *dom* $\varphi =$ *dom* $\varphi_0 \subseteq$ *dom* $\varphi_1 \subseteq$ *dom* $\varphi_2 \subseteq$ *dom* φ_3, so if $\mathcal{PE}[\![e_1]\!]\vec{v}\varphi\eta = \bot$ we simply apply the induction hypothesis to e_1; else if $\mathcal{PE}[\![e_2]\!]\vec{v}\varphi_1\eta_1 = \bot$ we apply the induction hypothesis to e_2, else $\mathcal{PE}[\![e_3]\!]\vec{v}\varphi_2\eta_2 = \bot$ and we apply the induction hypothesis to e_3.

Case $e \equiv b\ e_1 \ldots e_n$. Analogous to the **if** case.

Case $e \equiv f^S\ e_1 \ldots e_n$. The first part is analogous to the **if** case. Then, if $\mathcal{PE}[\![e_i]\!]\vec{v}\varphi_{i-1}\eta_{i-1} \neq \bot$ for all $i = 1, \ldots, n$, this is the required call and $\varphi'' = \varphi_n$.

Case $e \equiv f^D\ e_1 \dots e_n$. The first part is analogous to the **if** case. Then, if $\mathcal{PE}[\![e_i]\!]\vec{v}\varphi_{i-1}\eta_{i-1} \neq \perp$ for all $i = 1, \dots, n$, this is the required call. Further, we find that $(f, v'_1 \dots v'_m) \notin dom\ \varphi_n \supseteq dom\ \varphi$, because otherwise $\mathcal{PE}[\![e]\!]\vec{v}\varphi\eta \neq \perp$. This in turn implies that $\mathcal{PE}[\![f\ e_1 \dots e_n]\!]\vec{v}\varphi\eta = \phi\ f\ v'_1 \dots v'_m\ \underline{x_1} \dots \underline{x_{n-m}}\ \varphi''\ \eta_n = \mathcal{PE}[\![e^f]\!](v'_1, \dots, v'_m, \underline{x_1}, \dots, \underline{x_{n-m}})\varphi''\eta_n$ with $(f, v'_1 \dots v'_m) \in \varphi''$. □

LEMMA 7.2.3. *If* $\mathcal{PE}[\![f_0\ x_1 \dots x_m\ x_{m+1} \dots x_n]\!](\vec{v}_0, \underline{x_1}, \dots, \underline{x_{n-m}})\varphi^0\eta^0 = \perp$ *then there exists an infinite state transition sequence* $\sigma = (f_0, \vec{v}_0) \xrightarrow{c_1} (f_1, \vec{v}_1) \xrightarrow{c_2} \cdots$.

PROOF. Using Lemma 7.2.2 we find $\varphi^1 = \varphi''$ and a call $c_1 : f_1\ e_1 \dots e_n$ in the body of f_0 for which

$$\begin{aligned} &\mathcal{PE}[\![f_1\ e_1 \dots e_n]\!](\vec{v}_0, \underline{x_1}, \dots, \underline{x_{n-m}})\varphi_0\eta_0 \\ &= \mathcal{PE}[\![e^{f_1}]\!](v'_1, \dots, v'_n, \underline{x_1}, \dots, \underline{x_{n-m}})\varphi^1\eta^1 = \perp \end{aligned}$$

and $\perp \neq \mathcal{PE}[\![e_i]\!](\vec{v}_0, \underline{x_1}, \dots, \underline{x_{n-m}})\varphi_{i-1}\eta_{i-1} = (v'_i, \varphi_i, \eta_i)$. Now letting $\vec{v}_1 = (v'_1, \dots, v'_n)$, we find that $(f_0, \vec{v}_0) \xrightarrow{c_1} (f_1, \vec{v}_1)$ is a state transition, and $dom\ \varphi^0 \subseteq dom\ \varphi^1$.

Continuing in this fashion with e^{f_1} we find infinite sequences

$$\begin{array}{c} (f_0, \vec{v}_0) \xrightarrow{c_1} (f_1, \vec{v}_1) \xrightarrow{c_2} \cdots \xrightarrow{c_i} (f_i, \vec{v}_i) \xrightarrow{c_{i+1}} \cdots \\ dom\ \varphi^0 \subseteq dom\ \varphi^1 \subseteq \cdots \subseteq dom\ \varphi^i \subseteq \cdots \end{array}$$

Now suppose for some $i < j$ that $(f_i, \vec{v}_i) = (f_j, \vec{v}_j)$ and $\beta(c_i) = D$. As $(f_j, \vec{v}_j) = (f_i, \vec{v}_i) \in dom\ \varphi_i \subseteq dom\ \varphi_{j-1}$, we cannot have $\beta(c_j) = D$, due to the way the transition $(f_{j-1}, \vec{v}_{j-1}) \xrightarrow{c_j} (f_j, \vec{v}_j)$ was constructed using Lemma 7.2.2. Thus the sequence is a state transition sequence, cf. Definition 7.1.2. □

Definition 7.2.4. (PEQ-TERMINATION) Program p is said to be PEQ-terminating (partial evaluation quasi-terminating) for a binding-time division β iff for all parameters x we have $\beta(x) = S \Rightarrow BV(x)$.

LEMMA 7.2.5. (PEQ-TERMINATION IMPLIES TERMINATION OF PARTIAL EVALUATION) *Given a binding-time division β, if all function calls of program p are marked D, and p PEQ-terminates for β, partial evaluation of p also terminates.*

PROOF. Assume p is PEQ-terminating for a given binding-time division β. Let a function f and static input $\vec{s} = s_1 \dots s_m$ be given, and let N be the upper bound on the size of *poly*, i.e. $N = \sum_{(f\ x_1 \dots x_m = e^f) \in p} \prod_{i=1}^{m} |V_i|$, where $(x_i, V_i) \in \mathcal{V}(\sigma)$; we note that N is finite because p is PEQ-terminating. Defining hypothesis

$$\begin{aligned} H(n) = {} & \text{for all } e, \eta \text{ and } \varphi \text{ for which } |dom\ \varphi| = N - n, \\ & \perp \neq \mathcal{PE}[\![e]\!](s_1, \dots, s_m, \underline{x_{m+1}}, \dots, \underline{x_n})\varphi\eta = (e', \varphi', \eta') \\ & \text{and } |dom\ \varphi'| \geq |dom\ \varphi|, \end{aligned}$$

we prove the lemma by complete induction on n, noting that as all function calls are marked D, the first branch of function call interpretation is never used.

Case $n = 0$. This is proved by structural induction on e. The only case which is not immediately obvious is the recursive call to $\mathcal{PE}[\![e^f]\!](v'_1 \dots v'_m)\varphi''\eta_n$ in the function call interpretation, but this call will never be reached. This is due to the fact that when $n = 0$, $(f, v'_1 \dots v'_n) \in dom\ \varphi_n$

Case $n > 0$. Again, this is proved by structural induction on e. In this case we find that $|dom\ \varphi''| > |dom\ \varphi|$ in the recursive call to $\mathcal{PE}[\![e^f]\!](v'_1 \dots v'_m)\varphi''\eta_n$, so the induction hypothesis can be applied. □

7.3 A termination-ensuring binding-time analysis

Recall that the binding-time division $\beta : \mathit{Varname} \to \{S, D, ?\}$ classifies parameters as definitely static (S), definitely dynamic (D) or undecided (?). A binding-time division that ensures termination is then constructed in the following way: First we classify as $\beta_0(x) = D$ any parameter x that requires the values of the dynamic input to be known. Any remaining parameter y is classified $\beta_0(y) = ?$. Then we iteratively apply the *bounded anchoring* step (cf. Definition 6.2.3), which changes a number of ?-classifications into S-classifications. When no more ?-classifications can be changed by bounded anchoring, we change any remaining classification into D.

7.4 Dynamic parameters and congruency

It is well-known that for partial evaluation defined in Figure 7 to work, the binding-time division β must be *congruent,* that is, e_1 of $\mathbf{if}^S$-expressions and arguments $e_1, \dots, e_n$ of calls to basic functions b^S must be S-classified expressions, etc. The details of how to achieve congruency can be found elsewhere [Jones et al. 1993; Glenstrup 1999], here the important point to note is that whenever a parameter is classified D, congruency may cause other parameters to be classified D too.

It is obvious from Definition 7.2.4 that if the final change of the binding-time division mentioned at the start of Section 7.3 only changes ? to D, it will make no difference to the termination of partial evaluation of p, and it is also obvious that partial evaluation of p terminates for β_0. In this case we conclude from Theorem 6.2.4 that partial evaluation will terminate for the final binding-time division.

If, however, the final change results in the need to change S-classified parameters into D due to congruency, resulting in a binding-time division β, we must restart the entire process with a new binding time division

$$\beta_0(x) = \begin{cases} D, & \text{if } \beta(x) = D \\ ?, & \text{otherwise} \end{cases} .$$

As this new β_0 has strictly more parameters classified D than the original one, this process restart cannot go on forever, and we will end up with a congruent binding time division β for which partial evaluation terminates.

Note also that although the bounded anchoring process is restarted, we do not need to re-perform the costly computation of $\mathcal{S}$.

7.5 Specialization point insertion

Lemma 7.2.5 ensures termination of partial evaluation, but only if all function calls are residualized. In many cases it is desirable to perform some function calls at specialization time, e.g. the lk*xxx* functions in Figure 2.

To analyze the need for specialization points, conceptually we augment each function of the subject program p by adding a *call depth parameter*, called *dp*. At every function call this parameter is incremented by 1 if the call is to another function in the same strongly-connected component of the program call graph, and reset to 0 otherwise. Thus *dp* reflects the call depth within the strongly-connected component. It is intuitively clear that if *dp* of all functions f are analyzed to be of bounded static variation, i.e. $\beta(dp) = S$, specialization will terminate.

However, we can go further and compute minimized sets of specialization points which will guarantee that specialization will terminate. To this end, we extend the size-change graphs to include a set Γ of call sites, so that $G'_c = (G^{\uparrow}_c, G^{\downarrow}_c, \{c\})$ is the new size-change graph for call site c. Composition of two size-change graphs $(G^{\uparrow}_1, G^{\downarrow}_1, \Gamma_1)$ and $(G^{\uparrow}_2, G^{\downarrow}_2, \Gamma_2)$ is as before, with the resulting call site set Γ being $\Gamma_1 \cup \Gamma_2$. We denote the set of call-site extended size-change graphs for program p by $\mathcal{S}' = \{G'_{cs} \mid cs \text{ is well-formed}\}$, and define function $\mathit{calls}(G^{\uparrow}, G^{\downarrow}, \Gamma) = \Gamma$.

To ensure termination we must now require for any idempotent $G' \in \mathcal{S}'$ in which *dp* is unanchored that there exists at least one specialization point in the set of calls Γ corresponding to G':

Theorem 7.5.1. (Specialization point insertion correctness) *Given an augmented program p and a binding-time division β, if p PEQ-terminates for β, and for all $G \in \mathcal{S}'$ where $G = G;G$ and $\beta(dp) \neq S$ there exists a $c \in \mathit{calls}(G)$ for which $\beta(c) = D$, then partial evaluation of p terminates.*

Proof. Assume p PEQ-terminates for β, but partial evaluation of p does *not* terminate, i.e. there exist function f and $\vec{v} \in \mathit{Value}^*$ such that

$$\mathcal{PE}[\![f\ x_1 \ldots x_m\ x_{m+1} \ldots x_n]\!](v_1, \ldots, v_m, \underline{x_1}, \ldots, \underline{x_{n-m}})\{\}\{\} = \bot$$

By Lemma 7.2.3 there exists an infinite state transition sequence $\sigma = (f, \vec{v}) \xrightarrow{c_1} (f_1, \vec{v}_1) \xrightarrow{c_2} \cdots \xrightarrow{c_i} \cdots$. We let $\mathcal{S}'' = \{G''_i \mid i = 1, \ldots\}$ where $G''_i = \big(G^{\uparrow}_{c_i}, G^{\downarrow}_{c_i}, \{(c_i, \vec{v}_i)\}\big)$ with the same definition of composition as before. Note that $\mathcal{S}''$ is finite because p PEQ-terminates, yielding only a finite number of different values $\vec{v}_i$.

By a construction similar to that of the proof of Theorem 6.2.4, we can find a G''° with $G''^{\circ} = G''^{\circ}; G''^{\circ}$ for which a part of $\mathcal{M}^{\uparrow}(cs)$ can be expressed as $(G^{\uparrow}_{t_1}, \ldots, G^{\uparrow}_{t_2-1}), (G^{\uparrow}_{t_2}, \ldots, G^{\uparrow}_{t_3-1})$, where $G''_{t_i}; \ldots; G''_{t_i-1} = G''^{\circ}$. Note that as σ is infinite, the set of values of *dp* is unbounded, so $\beta(dp) \neq S$.

We now find that $\beta(c) = S$ for all $(c, \vec{v}) \in \mathit{calls}(G''^{\circ})$; otherwise, as $\xrightarrow{c}(f, \vec{v})$ occurs twice, σ would violate the last requirement of a partial evaluation state transition sequence (cf. Definition 7.1.2). □

8. IMPLEMENTATION AND EXPERIMENTS

A prototype of the bounded anchoring and specialization point analysis has been implemented in Scheme as an extension of the partial evaluator PGG [Thiemann 1999]. The implementation handles **let** expressions in the obvious way, and all higher-order and unknown functions (e.g., `eval` and `apply`) are conservatively size-approximated as returning values greater than all of their inputs.

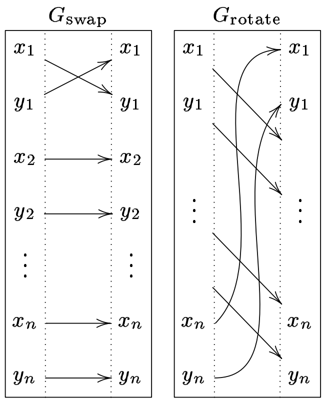


Fig. 8. Size change graphs which yield a transitive closure containing $n2^n$ graphs

8.1 Complexity

The analysis has been shown by Lee et al. [2001] to be intrinsically PSPACE complete, and the worst-case running time for PEQ-termination detection is exponential in the maximum arity of the functions of p. For instance, the size-change graphs of the program

```
f x₁ y₁ ... xₙ yₙ = if ... then
                        f y₁ x₁ x₂ y₂ x₃ y₃ ... xₙ yₙ
                    else
                        f xₙ yₙ x₁ y₁ x₂ y₂ ... xₙ₋₁ yₙ₋₁,
```

shown in Figure 8, force the analyzer to compute a set $\mathcal{S}$ with $n2^n$ size change graphs: The dependency edges of each pair (x_i, y_i) can be "rotated" up to n times by composing a number of copies of G_{rotate}, and the edges of each of the n pairs can be either "straight" or "swapped" by appropriate compositions of G_{swap}. This is confirmed by the implementation.

However, if the arity is assumed bounded, which is reasonable for typical programs, it can be shown that the implementation runs in worst-case cubic time in the size of p [Glenstrup 1999].

Due to the additional gathering of sets of call sites, potentially yielding $O(2^n)$ different call site sets, insertion of specialization points requires worst-case exponential time in the size n of p. We have reduced this problem by applying the following heuristic: We say that $G'_1 \preceq G'_2$ when for every edge $x \xrightarrow{\delta} x \in G_1^{\downarrow}$ there is an edge $x \xrightarrow{\delta'} x \in G_2^{\downarrow}$ with $\delta \sqsubseteq \delta'$, and for every edge $y \xrightarrow{\delta} y \in G_2^{\uparrow}$ there is an edge $y \xrightarrow{\delta'} y \in G_1^{\uparrow}$ with $\delta \sqsubseteq \delta'$, and $\Gamma_1 \subseteq \Gamma_2$. In other words, G'_1 has "weaker" (or identical) anchors than G'_2. During the computing of $\mathcal{S}'$ we then discard any $G'_2 \in \mathcal{S}'$ if there exists a $G'_1 \in \mathcal{S}'$ for which $G'_1 \prec G'_2$. Applying this heuristical optimization, the running times are reduced for typical programs.

8.2 Experiments

We have collected a suite of more than 50 small example programs, including several from the work of Lindenstrauss and Sagiv [1997]. 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 sorting functions. For instance, in mergesort `merge (split xs)` is changed into `splitmerge xs [] []`, where `splitmerge` splits `xs` into two lists using accumulating parameters and calls `merge` on them.

The programs and their analysis results are given in detail in Appendix C. In the majority of the examples no parameters are overconservatively generalized, the main exception being the list sorting functions. For these functions, the analyzes are not sufficiently accurate for detecting that the length of the list does not change when it is sorted. In the following we will discuss just a couple of the example programs: An interpreter and a pattern matcher.

8.2.1 *Simple first-order Scheme interpreter.* This program, a Scheme implementation of the interpreter in Figure 2, is shown in Figure 9. As it stands it employs dynamic scoping, but removing the four lines labeled '`DYNAMIC SCOPING`' changes it to employ static, lexical scoping. Function calls in the `evalexp` function are numbered for easy reference. The results of the termination analysis and specialization-point insertion analysis are shown in Figure 10 for the interpreter using static scoping, and Figure 11 for the interpreter using dynamic scoping. The results confirm that the analysis correctly generalizes the `names` list when using dynamic scoping, but keeps it static under static, lexical scoping. Furthermore, the minimal number specialization points necessary for termination of specialization are computed. Note that as `names` can be unbounded for dynamic scoping, a specialization point is needed in the `variable?` function, which causes its return value to become dynamic. In this program it has no further effect, but in general such binding-time changes of return values must be propagated and the analyses reiterated.

8.2.2 *Simple pattern matcher.* The simple pattern matcher shown in Figure 12 takes a pattern and a string and returns a list of positions in the string where the pattern matches. For static pattern and dynamic string, the analyzer correctly determines that `n` should be generalized, and that we must insert specialization points into the recursive calls to `domatch`, cf. Figure 13. However, if it is the string which is static, no parameters need to be generalized, and no specialization points are necessary.

9. RELATED WORK

The standard strategy for approaching termination in current implementations of partial evaluators is to insert memoization points at all dynamic conditionals [Bondorf and Danvy 1991]. This is based on the fact that if specialization then fails to terminate, there would also be a combination of inputs that would cause nontermination for normal evaluation. However, this strategy does not prevent nontermination when specializing an interpreter with dynamic scoping (cf. Section 8.2.1) where infinitely many residual functions would be generated. Neither

```
(define (run data program)
  (evalexp (lookup-body 'main program)
        (lookup-paramnames 'main program)
        data program))
(define (function? funname program)
  (and (pair? program) (or (eq? funname (caadar program))
                           (function? funname (cdr program)))))
(define (lookup-paramnames funname program)
  (if (eq? funname (caadar program)) (cdadar program)
      (lookup-paramnames funname (cdr program))))
(define (lookup-body funname program)
  (if (eq? funname (caadar program)) (caddar program)
      (lookup-body funname (cdr program))))
(define (variable? varname names)
  (and (pair? names)
       (or (eq? varname (car names)) (variable? varname (cdr names)))))
(define (lookup-value varname names values)
  (if (eq? varname (car names)) (car values)
      (lookup-value varname (cdr names) (cdr values))))
(define (evalexp exp names values program)
  (cond
   ((list? exp)
    (case (car exp)
      ((QUOTE) (cadr exp))
      ((LET LET*)
       (let* ((value
               (evalexp (car (cdaadr exp)) names values program))) ; 1
         (evalexp (caddr exp)                                      ; 2
                  (cons (caaadr exp) names)
                  (cons value values)
                  program)))
      ((IF)
       (if (evalexp (cadr exp) names values program)               ; 3
           (evalexp (caddr exp) names values program)              ; 4
           (evalexp (cadddr exp) names values program)))           ; 5
      (else
       (if (function? (car exp) program)                           ; 6
           (evalexp (lookup-body (car exp) program)                ; 7, 8
                    (append ; DYNAMIC SCOPING
                     (lookup-paramnames (car exp) program)         ; 9
                     names)  ; DYNAMIC SCOPING
                    (append ; DYNAMIC SCOPING
                     (argvals (cdr exp) names values program)      ; 10
                     values) ; DYNAMIC SCOPING
                    program)
           ;; else it must be a base function
           (apply (eval (car exp) (scheme-report-environment 5))
                  (argvals (cdr exp) names values program))))))    ; 11
   ((variable? exp names)                                          ; 12
    (lookup-value exp names values))                               ; 13
   (else exp))) ;; it must be a constant
(define (argvals exps names values program)
  (if (null? exps) '()
      (cons (evalexp (car exps) names values program)
            (argvals (cdr exps) names values program))))
```

Fig. 9. Interpreter for first order Scheme programs

```
Parameter binding times:
 (D = dynamic,  B = static & bounded,  S = static & possibly unbounded)
  argvals:  exps : B  names : B  values : D  program : B
  evalexp:  exp : B  names : B  values : D  program : B
  lookup-value:  varname : B  names : B  values : D
  variable?:  varname : B  names : B
  lookup-body:  funname : B  program : B
  lookup-paramnames:  funname : B  program : B
  function?:  funname : B  program : B
  run:  data : D  program : B
Specialization points:
Call 7 in evalexp to evalexp
```

Fig. 10. Analysis results for the interpreter using static, lexical scoping

```
Parameter binding times:
 (D = dynamic,  B = static & bounded,  S = static & possibly unbounded)
  argvals:  exps : B  names : S  values : D  program : B
  evalexp:  exp : B  names : S  values : D  program : B
  lookup-value:  varname : B  names : S  values : D
  variable?:  varname : B  names : S
  lookup-body:  funname : B  program : B
  lookup-paramnames:  funname : B  program : B
  function?:  funname : B  program : B
  run:  data : D  program : B
Specialization points:
Call 1 in variable? to variable?
Call 1 in lookup-value to lookup-value
Call 7 in evalexp to evalexp
```

Fig. 11. Analysis results for the interpreter using dynamic scoping

```
(define (strmatch patstr str)
  (domatch (string->list patstr) (string->list str) 0))

(define (domatch patcs cs n)
  (if (pair? cs)
      (if (prefix patcs cs)                          ; 1
          (cons n (domatch patcs (cdr cs) (+ n 1))) ; 2
          (domatch patcs (cdr cs) (+ n 1)))         ; 3
      (if (equal? patcs cs) (cons n '()) '())))

(define (prefix precs cs)
  (or (null? precs) (and (pair? cs) (and (equal? (car precs) (car cs))
                                         (prefix (cdr precs) (cdr cs))))))
```

Fig. 12. A simple pattern matcher that returns a list of match positions

For static pattern and dynamic string:

```
Parameter binding times:
 (D = dynamic,  B = static & bounded,  S = static & possibly unbounded)
  prefix:  precs : B  cs : D
  domatch:  patcs : B  cs : D  n : S
  strmatch:  patstr : B  str : D
Specialization points:
Call 3 in domatch to domatch
Call 2 in domatch to domatch
```

For dynamic pattern and static string:

```
Parameter binding times:
 (D = dynamic,  B = static & bounded,  S = static & possibly unbounded)
  prefix:  precs : D  cs : B
  domatch:  patcs : D  cs : B  n : B
  strmatch:  patstr : D  str : B
Specialization points:
```

Fig. 13. Analysis results for the pattern matcher

does the strategy prevent nontermination when specializing an interpreter to a *subject* program containing an infinite loop with no conditionals.

Another approach is *poor man's generalization* [Holst 1988; Bondorf and Jørgensen 1993], in which all parameters that are not controlling the specialization (i.e., are tested in an **if** construct) are generalized. This would lead to overly conservative results for the interpreter, where both the list of names, `ns`, and the program, `pg`, would be generalized, resulting in no specialization at all!

The concept of quasitermination was first considered by [Holst] which also included a technique for generalizing parameters involved in an in-situ (i.e., from a parameter to itself) increasing loop not anchored in an in-situ decreasing loop for a static parameter of bounded variation. Andersen and Holst [1996] presented an 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 to our computation of $\mathcal{S}$.

The main differences are that we clearly separate the modalities by conceptually using both $G^{\downarrow}$ and $G^{\uparrow}$, and state their safety conditions, which are substantially different in nature. Our distinction between $\underline{\underline{\uparrow}}$ and $\overline{\overline{\downarrow}}$ enables 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 $\mathtt{x} \xrightarrow{\uparrow} \mathtt{x}$ and thus a generalization of `x`, whereas we just record an equality transition, leading to no generalization.

Furthermore we include an algorithm for inserting a set of necessary specialization points, an essential ingredient in making specialization of quasi-terminating programs terminate. This in turn requires that we handle cases where an 'S' parameter used as anchor becomes dynamic due to an inserted specialization point.

The present work is an extension of previous work [Glenstrup and Jones 1996; Glenstrup 1999], the main difference being the correctness proofs, and the size-change graph framework which allows less conservative annotations due to the condition that only increasing loops in *idempotent* size-change graphs need to be anchored.

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 those techniques could be transferred to binding-time analyses 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 parameters. In fact, it turns out that to avoid too conservative binding-time divisions due to dynamic conditionals, he adds analyses similar to anchoring the call depth parameter 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 a program with static parameter s and dynamic parameter d which is known to have a value between 1 and 10:

$$s = d \qquad \text{into} \qquad \begin{array}{l} ss = 10; \\ \textbf{while}\ (ss\ != d)\ ss = ss - 1; \\ s = ss; \end{array}$$

one has explicitly introduced a dynamic control dependency (from $ss\ != d$) to an 'S' parameter (ss). This will cause Das' control dependency approach to conservatively generalize the parameter, canceling 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 parameters that are truly bounded, and supply a set of specialization points to avoid static-infinite computations. As partial evaluation is over-strict, possibly performing computations not intended by the programmer, we believe that a full termination guarantee should be given.

—Additionally, the part of Das' work handling functional programs makes use of a weaker form of bounded anchoring, and will thus handle lexicographic ordering conservatively. In fact, his condition for detecting an unsafe, i.e. possibly unbounded, parameter v is [Das 1998, Algorithm 3, p. 162]

```
  If    v gets a value from an unbounded parameter
  or    (there exists an increasing loop $v \rightarrow \cdots \rightarrow v$    and
         there exists no sibling parameter w which decreases along all loops
         $w \rightarrow \cdots \rightarrow w$)
then    v must be generalized 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 graph is used for both $G^{\uparrow}$ and $G^{\downarrow}$ because only tail-recursive programs are handled. Thus, both edges labeled $\uparrow$ and $\downarrow$ 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 canceling effect to our analyses.

9.1 Pessimistic vs. optimistic binding-time analysis

One aspect that makes our approach stand out is that we start by assuming all parameters to be "unsafe," i.e. marked '?', and we only promote to 'S' when we have a guarantee (an anchor) that $x \in \textit{bounded}(\sigma)$. This can be termed a 'safe' or 'pessimistic' approach. Other approaches [Andersen and Holst 1996; Das and Reps 1996; Das 1998] are 'optimistic' in the sense that they start by marking all parameters 'S', and then re-classify as dynamic parameters which seem to be unbounded. We do not expect there to be any difference in power, i.e. that the safe approach generalizes parameters too conservatively[19], 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.

9.2 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 and 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, and this relies on the use of different anchors in different loops to allow less 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 9.2.1. (No lexicographic ordering)

```
f dp (a₁, b₁) (a₂, b₂) (a₃, b₃) = ...
  if ... then fᵃ (dp+1) (b₁-1, a₁-1) (a₂-1, b₂-1) (b₃-1, a₃-1)
         else fᵇ (dp+1) (b₁-1, a₁-1) (b₂-1, a₂-1) (a₃-1, b₃-1)
```

In this example, we have tupled the parameters to highlight that parameter interactions only occur between $\texttt{a}_i$ and $\texttt{b}_i$.

[19]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.

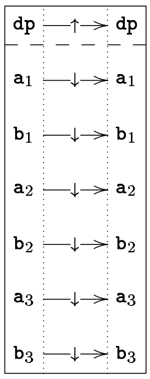


Fig. 14. Idempotent size change graph for Example 9.2.1

It can be shown by bounded anchoring that specialization of f with all parameters static terminates (i.e. that dp is marked 'S'):

(1) All a's and b's are bounded (there are no increasing operations).

(2) The only $G \in \mathcal{S}$ for which $G; G = G$ is shown in Figure 14 (where all as and bs have been left out of $G^{\uparrow}$) and it can clearly be seen that dp is anchored and thus bounded.

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 discover under what ordering termination should be proved.

9.3 Other work on size-change termination

The size-change termination principle was first used for detecting termination of normal evaluation [Lee et al. 2001] which only requires size-change graphs involving decreasing dependencies. The contribution of the present work is the addition of increasing dependencies necessary for termination analysis in partial evaluation, including the corresponding safety condition; expressing and proving correct the bounded anchoring principle in the size-change graph framework; and the use of analysis results for specialization-point insertion. In work made available to the authors recently [Lee 2002a; Lee 2002b], similar formulations and proofs are given. While Lee handles unfolding of static function calls ("local termination" of a reduction system) and dynamic function calls ("global termination" of a specializer state transition system) separately, the present work includes memoization directly in an operational definition of partial evaluation (cf. Figure 7). In Lee's PhD Thesis [Lee 2002b], higher-order constructs are also handled.

10. CONCLUSION

The problem of termination of generated programs and program generators must be solved before fully automatic and reliable program generation for a broad range of applications becomes a reality. Achieving this goal is also necessary, if we hope ever to elevate software engineering from its current state (a highly-developed handiwork) into a successful branch of engineering, capable of solving a wide range of new problems by systematic, well-automated and well-founded methods.

In this paper we have presented a binding-time and specialization-point insertion analysis for programs written in a first-order functional language. The analysis computes a binding-time division and a set of specialization points which we have proved ensure termination of the specialization phase.

Algorithms implementing the binding-time analysis obtain a worst-case complexity of $O(p^3)$ for programs of bounded arity, where p is the program size. Computing a set of specialization points has exponential worst-case time, but can be reduced for typical programs using heuristical optimizations.

Experiments with a prototype implementation have shown that the analyses work well on several small example 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 deconstructing and constructing operations.

This paper contributes towards making off-line partial evaluators automatic and useful tools, even for users who know little about specializer termination problems. But this is not all—the techniques presented here can also improve both the degree and speed of on-line specialization by annotating parameters that are guaranteed to be of bounded variation and thus do not need to be checked for homeomorphic embeddings, and do not need to be generalized. Another important contribution is, we hope, a better understanding of the problems occurring in the attempt to ensure termination of partial evaluation.

10.1 Directions for future work

10.1.1 *Specialization-point insertion.* The standard way of selecting specialization points is to choose all dynamic conditionals, unfolding all the original functions. This does not prevent static-infinite computations, nor does it prevent unbounded parameters during specialization. While our strategy guarantees termination, it would be interesting to compare it with the standard strategy for real-world programs.

As the algorithm for finding a small set of specialization points necessary for guaranteeing termination has exponential worst-case complexity, this problem should be addressed in depth in future research.

10.1.2 *More precise size dependencies.* The present algorithm is based on a rather crude approximation of the program values, using only $\uparrow$ and $\downarrow$ arrows, and is overly conservative in many cases, e.g. `cdr (cons x y)` which could be considered as $\{\ddagger(\mathtt{y})\}$, but is approximated to $\{\uparrow(\mathtt{x}), \uparrow(\mathtt{y})\}$. Das and Reps have addressed this problem, using context-free language reachability [Das 1998; Das and Reps 1996], 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] and Pareto's [2000] sized types.

10.1.3 *Other data domains.* An obvious extension necessary for the analyses to be useful in practice is to handle integers and other data domains. This could be realized in a fairly straightforward manner by adding a domain analysis to find some bounds on the values that each parameter 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 and Le Métayer 1997; Ghiya and Hendren 1996] to infer the "shapes" of the data structures (e.g. a list constructed using pointers), before they could be used as anchors.

10.1.4 *General topics.* We have described recent progress towards taming rather intricate problems in the context of partial evaluation. While the large lines are becoming much clearer, there is still much to do to bring these ideas to the level of practical, day-to-day usability. Therefore we conclude with a long list of goals and challenges.

Termination analysis.

—Develop a termination analysis for a realistic programming language, e.g., C, Java or C#.
—Develop termination-guaranteeing BTAs for
  —a functional language (e.g., Scheme).
  —an imperative language (e.g., C).
  —more widely used programming languages (e.g., Java, C#).
—Develop a still stricter BTA to identify programs whose specialized versions will always terminate.
—Find ways to combine termination analysis with
  —*abstract interpretation*
  —*other static analyses.* For example, the size-change analysis depends critically on the fact that data is well-founded (e.g., natural numbers or lists), but the more common integer type is *not* well-founded.

One approach is to apply abstract interpretation to the type of integers to recognize non-negative parameters, and somehow combine this information with size-change analysis.

Partial evaluation.

—Perform efficient, reliable, predictable specialization of realistic programming languages, e.g., C, Java and C#.

Static program analyses.

—Devise an analysis for *overlap detection*, to discover when a function can be called repeatedly with the same arguments. Memoization can make such a program run faster, often yielding superlinear speedups, but should not be used indiscriminately due to time and space overhead.
—Devise a way automatically to estimate or bound a program's running time, as a function of its input size or value.

$$
\begin{aligned}
\mathcal{E}^{\downarrow} = \mathcal{E}_{\mathbf{e}}^{\downarrow}\ \Big(\mathit{fix}\ (\lambda\phi.\{ \mathit{f1} \mapsto \lambda\Delta_1 \ldots \Delta_m . \mathcal{E}_{\mathbf{e}}^{\downarrow}\ \phi\ [\![e^{\mathit{f1}}]\!]\ (\Delta_1, \ldots, \Delta_m), \ldots, \\
\mathit{fn} \mapsto \lambda\Delta_1 \ldots \Delta_k\ . \mathcal{E}_{\mathbf{e}}^{\downarrow}\ \phi\ [\![e^{\mathit{fn}}]\!]\ (\Delta_1, \ldots, \Delta_k)\})\Big)\ \vec{\Delta}_{\mathbf{id}}^{\downarrow}
\end{aligned}
$$

$$\vec{\Delta}_{\mathbf{id}}^{\downarrow} = (\{\overline{\overline{\downarrow}}(x_1)\}, \ldots, \{\overline{\overline{\downarrow}}(x_n)\})$$

$$
\begin{aligned}
\mathcal{E}_{\mathbf{e}}^{\downarrow}\ \phi\ [\![k]\!]\ \vec{\Delta} &= \{\} \\
\mathcal{E}_{\mathbf{e}}^{\downarrow}\ \phi\ [\![x_i]\!]\ \vec{\Delta} &= \Delta_i \\
\mathcal{E}_{\mathbf{e}}^{\downarrow}\ \phi\ [\![\textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3]\!]\ \vec{\Delta} &= \mathcal{E}_{\mathbf{e}}^{\downarrow}\ \phi\ [\![e_2]\!]\ \vec{\Delta}\ \sqcap^{\downarrow}\ \mathcal{E}_{\mathbf{e}}^{\downarrow}\ \phi\ [\![e_3]\!]\ \vec{\Delta} \\
\mathcal{E}_{\mathbf{e}}^{\downarrow}\ \phi\ [\![b\ e_1 \ldots e_n]\!]\ \vec{\Delta} &= \mathcal{E}_{\mathbf{b}}^{\downarrow}\ b\ (\mathcal{E}_{\mathbf{e}}^{\downarrow}\ \phi\ [\![e_1]\!]\ \vec{\Delta}) \ldots (\mathcal{E}_{\mathbf{e}}^{\downarrow}\ \phi\ [\![e_n]\!]\ \vec{\Delta}) \\
\mathcal{E}_{\mathbf{e}}^{\downarrow}\ \phi\ [\![f\ e_1 \ldots e_n]\!]\ \vec{\Delta} &= \phi\ f\ (\mathcal{E}_{\mathbf{e}}^{\downarrow} \phi\ [\![e_1]\!]\ \vec{\Delta}) \ldots (\mathcal{E}_{\mathbf{e}}^{\downarrow}\ \phi\ [\![e_n]\!]\ \vec{\Delta})
\end{aligned}
$$

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

Fig. 15. Operator approximating decreasing size information for the value of an expression

APPENDIX

A. FINITE RAMSEY'S THEOREM

THEOREM A.1. (FINITE RAMSEY'S THEOREM) *There exists a computable function $R(e, r, k)$ such that for all positive integers e, r, k and each finite set M the following holds: if $|M| \geq R(e, r, k)$, and each e-member subset of M is marked by one of r colors, then there is a subset H of M such that $|H| = k$, and all e-member subsets of H are marked by the same color.*

B. APPROXIMATION OPERATORS

Figures 15 and 16 show the approximation operators used in the implementation for the experiments. The $\mathcal{E}^{\downarrow}$ operator is a straightforward abstract interpretation of the program, while the $\mathcal{E}^{\uparrow}$ is more subtle because it must distinguish between bounded and unbounded size increases for infinitely many environments $\vec{v}$.

A call to the helper operator $\mathcal{E}_{\mathbf{e}}^{\uparrow}\ \phi\ [\![e]\!]\ g\ F\ X\ \delta\ \vec{\Delta}$ returns a triple (Δ', X', δ') such that

- Δ' is the size dependency set for e
- X' is the set of free variables encountered in **if**-tests during recursive descent and unfolding of e to any call to g
- δ' is '$\uparrow$' if there is an increasing operation (i.e., a `cons`) during recursive descent and unfolding of e to any call to g, else it is '$\underline{\underline{\uparrow}}$.'

The arguments F, X and δ are accumulating parameters, where

- F is the set of names of functions "on the call stack"
- X is the set of free variables encountered in **if**-tests
- δ is '$\uparrow$' if an increasing operation (i.e., a `cons`) has been encountered, else it is '$\underline{\underline{\uparrow}}$'

If an increasing operation is encountered between one call to a function f and a (possibly indirect) recursive call to itself, we say it has "recursive increase." This is

$$\begin{array}{l}
\mathcal{E}^{\uparrow}[\![e]\!] = \Delta, \\
\text{where } (\Delta, X, \delta) \;=\; \mathcal{E}_{\mathbf{e}}^{\uparrow}\ \big(\mathit{fix}\ (\lambda\phi.\{\mathit{f1} \mapsto \lambda gFX\Delta_1 \ldots \Delta_m . \mathcal{E}_{\mathbf{e}}^{\uparrow}\ \phi\ [\![e^{\mathit{f1}}]\!]\ g\ F\ X\ (\Delta_1, \ldots, \Delta_m), \ldots, \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{fn} \mapsto \lambda gFX\Delta_1 \ldots \Delta_k\ . \mathcal{E}_{\mathbf{e}}^{\uparrow}\ \phi\ [\![e^{\mathit{fn}}]\!]\ g\ F\ X\ (\Delta_1, \ldots, \Delta_k)\})) \\
\qquad\qquad\qquad\quad \bullet\ \{\}\ \{\}\ \ddagger\ \vec{\Delta}^{\uparrow}_{\mathbf{id}} \\
\qquad\qquad \vec{\Delta}^{\uparrow}_{\mathbf{id}} \;=\; (\{\ddagger(x_1)\}, \ldots, \{\ddagger(x_n)\})
\end{array}$$

$$\begin{array}{lcl}
\mathcal{E}_{\mathbf{e}}^{\uparrow}\ \phi\ [\![k]\!]\ g\ F\ X\ \delta\ \vec{\Delta} & = & (\{\}, \{\}, \ddagger) \\
\mathcal{E}_{\mathbf{e}}^{\uparrow}\ \phi\ [\![x_i]\!]\ g\ F\ X\ \delta\ \vec{\Delta} & = & (\Delta_i, \{\}, \ddagger) \\
\mathcal{E}_{\mathbf{e}}^{\uparrow}\ \phi\ [\![\textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3]\!] & & \\
\qquad\qquad\qquad g\ F\ X\ \delta\ \vec{\Delta} & = & (\Delta'_2 \sqcup^{\uparrow} \Delta'_3, X_1 \cup X_2 \cup X_3, \delta_1 \sqcup^{\uparrow} \delta_2 \sqcup^{\uparrow} \delta_3), \\
 & & \text{where } (\Delta'_1, X_1, \delta_1) \;=\; \mathcal{E}_{\mathbf{e}}^{\uparrow}\ \phi\ [\![e_1]\!]\ g\ F\ X\ \delta\ \vec{\Delta} \\
 & & \qquad\quad (\Delta'_i, X_i, \delta_i) \;=\; \mathcal{E}_{\mathbf{e}}^{\uparrow}\ \phi\ [\![e_i]\!]\ g\ F\ (\mathit{fv}\ e_1 \cup X)\ \delta\ \vec{\Delta}, i > 1 \\
\mathcal{E}_{\mathbf{e}}^{\uparrow}\ \phi\ [\![\texttt{cons}\ e_1\ e_2]\!]\ g\ F\ X\ \delta\ \vec{\Delta} & = & (\{\uparrow(f_x) \mid \delta(f_x) \in \Delta'_1 \cup \Delta'_2\}, X_1 \cup X_2, \delta_1 \sqcup^{\uparrow} \delta_2) \\
 & & \text{where } (\Delta'_i, X_i, \delta_i) \;=\; \mathcal{E}_{\mathbf{e}}^{\uparrow}\ \phi\ [\![e_i]\!]\ g\ F\ X \uparrow \vec{\Delta} \\
\mathcal{E}_{\mathbf{e}}^{\uparrow}\ \phi\ [\![\texttt{car}\ e_1]\!]\ g\ F\ X\ \delta\ \vec{\Delta} & = & \mathcal{E}_{\mathbf{e}}^{\uparrow}\ \phi\ [\![e_1]\!]\ g\ F\ X\ \delta\ \vec{\Delta} \\
\mathcal{E}_{\mathbf{e}}^{\uparrow}\ \phi\ [\![f\ e_1 \ldots e_n]\!]\ g\ F\ X\ \delta\ \vec{\Delta} & = & (\Delta', X' \cup X_1 \cup \cdots \cup X_n, \delta' \sqcup^{\uparrow} \delta_1 \sqcup^{\uparrow} \cdots \sqcup^{\uparrow} \delta_n)
\end{array}$$

$$\begin{array}{rcll}
\text{where } (\Delta'_i, X_i, \delta_i) & = & \mathcal{E}_{\mathbf{e}}^{\uparrow}\ \phi\ [\![e_i]\!]\ g\ F\ X\ \delta\ \vec{\Delta} & \text{(analyze arguments)} \\
(\Delta'', X'', \delta'') & = & \phi\ f\ g\ (F \cup \{f\})\ X\ \delta\ (\Delta'_1, \ldots, \Delta'_n) & \text{(analyze function call)} \\
(X', \delta') & = & \text{if } f = g \text{ then } (X, \delta) \text{ else } (X'', \delta'') & \text{(return recursive increase info)} \\
(\Delta^f, X^f, \delta^f) & = & \phi\ f\ f\ \{\}\ \{\}\ \ddagger\ (\{\ddagger(x_1)\}, \ldots, \{\ddagger(x_n)\}) & \text{(compute recursive increase infor-} \\
 & & & \text{mation for } f\text{)} \\
\Delta' & = & \text{if } f \in F \wedge (\delta^f = \uparrow \vee \exists x : \uparrow(x) \in \Delta^f) & \text{(if } f \text{ is recursive and has} \\
 & & \text{then } \Delta'' \cup \{\uparrow(x) \mid x \in X^f\} & \text{recursive increase, then add} \\
 & & \text{else } \Delta'' & \text{free variables from } \textbf{if}\text{-tests)}
\end{array}$$

$$\begin{array}{rcl}
\text{where } \Delta_1 \sqcup^{\uparrow} \Delta_2 & = & \{\uparrow(f_x) \mid \uparrow(f_x) \in \Delta_1 \cup \Delta_2\} \cup \\
 & & \{\ddagger(f_x) \mid \ddagger(f_x) \in \Delta_1 \cup \Delta_2 \wedge \uparrow(f_x) \notin \Delta_1 \cup \Delta_2\} \\
\delta_1 \sqcup^{\uparrow} \delta_2 & = & \text{if } \delta_1 = \uparrow \vee \delta_2 = \uparrow \text{ then } \uparrow \text{ else } \ddagger
\end{array}$$

Fig. 16. Operator for approximating increasing size information for the value of an expression. '•' is a symbol distinct from all function names.

true of the following functions which return the list length in a unary represention:

$$\begin{array}{lcl}
\texttt{length1}\ xs & = & \textbf{if}\ xs = \texttt{nil}\ \textbf{then}\ \texttt{nil}\ \textbf{else}\ \texttt{cons 1 (length1 (cdr}\ xs\texttt{))} \\
\texttt{length2}\ xs\ l & = & \textbf{if}\ xs = \texttt{nil}\ \textbf{then}\ l\ \textbf{else}\ \texttt{length2 (cdr}\ xs\texttt{) (cons 1}\ l\texttt{)}
\end{array}$$

A safe approximation must include $\uparrow(xs)$ for these functions, but a naïve abstract interpretation would not detect this, because xs is only used in the **if**-tests.

$\mathcal{E}_{\mathbf{e}}^{\uparrow}$ of Figure 16 is safe because in the case of a function call $f\ e_1 \ldots e_n$, all free variables of **if**-tests in recursive f-loops are conservatively added to the returned dependency set if f has recursive increase.

The remaining cases of $\mathcal{E}_{\mathbf{e}}^{\uparrow}$ are straightforward; note that the free variables of the **if**-test are added to the accumulating parameter X in the recursive calls for e_2 and e_3, and that the accumulating parameter δ is changed to $\uparrow$ in the recursive call for e_1 and e_2 for the `cons` case.

$$\frac{}{\vec{v}, \vec{\Delta} \vdash k \hookrightarrow k : \{\}} \qquad \frac{}{\vec{v}, \vec{\Delta} \vdash x_i \hookrightarrow v_i : \Delta_i} \qquad \frac{\vec{v}, \vec{\Delta} \vdash e_1 \hookrightarrow \mathsf{true} : \Delta'_1 \quad \vec{v}, \vec{\Delta} \vdash e_2 \hookrightarrow v'_2 : \Delta'_2}{\vec{v}, \vec{\Delta} \vdash \textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3 \hookrightarrow v'_2 : \Delta'_2}$$

$$\frac{\vec{v}, \vec{\Delta} \vdash e_1 \hookrightarrow \mathsf{false} : \Delta'_1 \quad \vec{v}, \vec{\Delta} \vdash e_3 \hookrightarrow v'_3 : \Delta'_3}{\vec{v}, \vec{\Delta} \vdash \textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3 \hookrightarrow v'_3 : \Delta'_3} \qquad \frac{\vec{v}, \vec{\Delta} \vdash e_1 \hookrightarrow v'_1 : \Delta'_1 \quad \vec{v}, \vec{\Delta} \vdash e_2 \hookrightarrow v'_2 : \Delta'_2}{\vec{v}, \vec{\Delta} \vdash \texttt{cons}\ e_1\ e_2 \hookrightarrow (v'_1.v'_2) : \{\}}$$

$$\frac{\vec{v}, \vec{\Delta} \vdash e_1 \hookrightarrow (v'_1.v'_2) : \Delta' \quad \downarrow\Delta' = \{\downarrow(x) \mid \delta(x) \in \Delta'\}}{\vec{v}, \vec{\Delta} \vdash \texttt{car}\ e_1 \hookrightarrow v'_1 : \downarrow\Delta'}$$

$$\frac{\vec{v}, \vec{\Delta} \vdash e_1 \hookrightarrow (v'_1.v'_2) : \Delta' \quad \downarrow\Delta' = \{\downarrow(x) \mid \delta(x) \in \Delta'\}}{\vec{v}, \vec{\Delta} \vdash \texttt{cdr}\ e_1 \hookrightarrow v'_2 : \downarrow\Delta'} \qquad \frac{\begin{array}{c}\vec{v}, \vec{\Delta} \vdash e_i \hookrightarrow v'_i : \Delta'_i, i = 1, \ldots, n \\ (v'_1, \ldots, v'_n), (\Delta'_1, \ldots, \Delta'_n) \vdash e^f \hookrightarrow v^f : \Delta^f\end{array}}{\vec{v}, \vec{\Delta} \vdash f\ e_1 \ldots e_n \hookrightarrow v^f : \Delta^f}$$

Fig. 17. Inference system connecting evaluation and decreasing size approximation

B.1 Safety proofs

B.1.1 *Safety of decreasing approximation.* The inference system shown in Figure 17 is used to derive a judgment $\vec{v}, \vec{\Delta} \vdash e \hookrightarrow v : \Delta$ which informally states that if evaluating e using $\vec{v}$ for the free variables terminates, it yields v, and also that under dependency environment $\vec{\Delta}$, Δ decribes what e is decreasingly dependent on.

The following lemma states formally the connection to evaluation:

Lemma B.1.1. *For all $e, \vec{v}, \vec{\Delta}$, if $\mathcal{E}$ $[\![e]\!]$ $\vec{v} = v' \notin \{\mathsf{error}, \bot\}$, then $\vec{v}, \vec{\Delta} \vdash e \hookrightarrow v : \Delta$ is well-defined and $v' = v$*

Proof. By fixpoint induction: let

$$P(\phi) = \text{for all } e, \vec{v}, \vec{\Delta}, \text{if } \mathcal{E}_{\mathsf{e}}\ \phi\ [\![e]\!]\ \vec{v} = v' \notin \{\mathsf{error}, \bot\},$$
$$\text{then } \vec{v}, \vec{\Delta} \vdash e \hookrightarrow v : \Delta \text{ is well-defined and } v' = v$$

be the property we want to prove, and let

$$F\ \phi = \{ f1 \mapsto \lambda v_1 \ldots v_m . \mathcal{E}_{\mathsf{e}}\ \phi[\![e^{f1}]\!](v_1, \ldots, v_m), \ldots,$$
$$fn \mapsto \lambda v_1 \ldots v_k\ . \mathcal{E}_{\mathsf{e}}\ \phi[\![e^{fn}]\!](v_1, \ldots, v_k)\}$$

We now show

1: $P(\bot)$. This is shown by structural induction over e.

2: $P(\phi) \Rightarrow P(F\ \phi)$. Assume $P(\phi)$ and let $e, \vec{v}$ be given such that $\mathcal{E}_{\mathsf{e}}\,(F\ \phi)\ [\![e]\!]\ \vec{v} = v \notin \{\mathsf{error}, \bot\}$. If $\mathcal{E}_{\mathsf{e}}\ \phi\ [\![e]\!]\ \vec{v} = v' \notin \{\mathsf{error}, \bot\}$ then $v = v'$ because $\phi \sqsubseteq F\ \phi$, and we are done.

Otherwise, $\mathcal{E}_{\mathsf{e}}\ \phi\ [\![e]\!]\ \vec{v} = \bot$ and we perform structural induction over e using the hypothesis

$$H(e) = \text{if } \mathcal{E}_{\mathsf{e}}\ \phi[\![e]\!]\vec{v} = \bot \text{ and } \mathcal{E}_{\mathsf{e}}\,(F\ \phi)[\![e]\!]\vec{v} = v' \notin \{\mathsf{error}, \bot\},$$
$$\text{then } \vec{v}, \vec{\Delta} \vdash e \hookrightarrow v : \Delta \text{ is well-defined and } v' = v$$

Case $e \equiv k$ and $e \equiv x_i$. Not applicable, because $\mathcal{E}_{\mathsf{e}}\ \phi\ [\![e]\!]\ \vec{v} = \bot$

Case $e \equiv$ **if** e_1 **then** e_2 **else** e_3. Assume wlog. $\mathcal{E}_{\mathsf{e}}\,(F\ \phi)\ [\![e_1]\!]\ \vec{v} = \mathsf{true}$ and $\mathcal{E}_{\mathsf{e}}\,(F\ \phi)\ [\![e_2]\!]\ \vec{v} = v'$. If $\mathcal{E}_{\mathsf{e}}\ \phi\ [\![e_2]\!]\ \vec{v} = v'' \notin \{\mathsf{error}, \bot\}$ then $v' = v''$ because

$\phi \sqsubseteq F\ \phi$, and we apply $P(\phi)$. If, on the other hand, $\mathcal{E}_{\mathsf{e}}\ \phi\ [\![e_2]\!]\ \vec{v} = v'' = \bot$, we apply the induction hypothesis $H(e_2)$. Either way, we find that $\vec{v}, \vec{\Delta} \vdash e_2 \hookrightarrow v'_2 : \Delta_2$ is well-defined and $v' = v'_2$. Applying the same reasoning to e_1 and using the inference rule for **if** we find that $\vec{v}, \vec{\Delta} \vdash$ **if** e_1 **then** e_2 **else** $e_3 \hookrightarrow v'_2 : \Delta_2$ is well-defined and $v' = v'_2$.

Case $e \equiv$ `car` e_1 *and* $e \equiv$ `cdr` e_1. Applying the same reasoning as in the **if** case we find that $\vec{v}, \vec{\Delta} \vdash e_1 \hookrightarrow (v'_1.v'_2) : \Delta$ is well-defined and $\mathcal{E}_{\mathsf{e}}\,(F\ \phi)\ [\![e_1]\!]\ \vec{v} = v' = (v'_1.v'_2)$ (v' must be a cons cell because $\mathcal{E}_{\mathsf{e}}\,(F\ \phi)\ [\![e]\!]\ \vec{v} \neq \bot$). The inference rule for `car`/`cdr` then shows this case.

Case $e \equiv$ `cons` $e_1\ e_2$. Analogous to the preceding case.

Case $e \equiv f\ e_1 \ldots e_n$. By the same reasoning as the **if** case we find that $\vec{v}, \vec{\Delta} \vdash e_i \hookrightarrow v'_i : \Delta_i$ is well-defined and $\mathcal{E}_{\mathsf{e}}\,(F\ \phi)\ [\![e_i]\!]\ \vec{v} = v'_i$ for $i = 1, \ldots, n$. We have $(F\ \phi)\ f\ v'_1 \ldots v'_n = \mathcal{E}_{\mathsf{e}}\ \phi\ [\![e^f]\!](v'_1, \ldots, v'_n) = v' \notin \{\mathsf{error}, \bot\}$ which by applying $P(\phi)$ implies that $(v'_1, \ldots, v'_n), \vec{\Delta}^f \vdash e^f \hookrightarrow v^f : \Delta^f$ is well-defined and $v' = v^f$.

3: $(\forall i : P(\phi_i)) \Rightarrow P\left(\bigsqcap_i \phi_i\right)$ *for ascending chains* $\phi_1 \sqsubseteq \phi_2 \sqsubseteq \cdots$. Given $\vec{v}$ and e, assume $\mathcal{E}_{\mathsf{e}}\ \left(\bigsqcap_i \phi_i\right)\ [\![e_i]\!]\ \vec{v} = v' \notin \{\mathsf{error}, \bot\}$. Then there must exist a ϕ_i such that $\mathcal{E}_{\mathsf{e}}\ \phi_i\ [\![e_i]\!]\ \vec{v} = v' \notin \{\mathsf{error}, \bot\}$, which by $P(\phi_i)$ proves the desired property. □

We extend the partial order $\sqsubseteq$ to labels, dependencies, dependency sets and dependency environments by

$$\begin{array}{rcl}
\mapsdown \sqsubseteq \downarrow & & \\
\mapsup \sqsubseteq \uparrow & & \\
\delta_1(x) \sqsubseteq \delta_2(x) & \Leftrightarrow & \delta_1 \sqsubseteq \delta_2 \\
\Delta_1 \sqsubseteq \Delta_2 & \Leftrightarrow & \forall(\delta_1(x)) \in \Delta_1 \exists(\delta_2(x)) \in \Delta_2 : \delta_1(x) \sqsubseteq \delta_2(x) \\
\vec{\Delta}_1 \sqsubseteq \vec{\Delta}_2 & \Leftrightarrow & \forall i : \Delta_{1i} \sqsubseteq \Delta_{2i}
\end{array}$$

Definition B.1.2. Given $\vec{u} \in \mathit{Value}^*$, we say that

—Δ $\vec{u}$-describes v iff $(\downarrow(x_j) \in \Delta \Rightarrow v < u_j) \wedge (\mapsdown(x_j) \in \Delta \Rightarrow v \leq u_j)$

—$\vec{\Delta}$ $\vec{u}$-describes $\vec{v}$ iff Δ_i $\vec{u}$-describes v_i for all i.

Corollary B.1.3. *If* $\Delta' \sqsubseteq \Delta$ *and* Δ *$\vec{u}$-describes* v *then* Δ' *also $\vec{u}$-describes* v. *Further, if* Δ *$\vec{u}$-describes* v, *then* $\Delta \sqcap^{\downarrow} \Delta''$ *(cf. Figure 15) also $\vec{u}$-describes* v *for any* Δ''.

Now that we have made the connection to normal evaluation, the following key lemma shows the required safety relation:

Lemma B.1.4. *Let* $\vec{u}$ *be given. Now for all* $e, \vec{v}, \vec{\Delta}$, *if* $\mathcal{E}\ [\![e]\!]\ \vec{v}$ *terminates without errors and* $\vec{\Delta}$ *$\vec{u}$-describes* $\vec{v}$, *then* $\vec{v}, \vec{\Delta} \vdash e \hookrightarrow v : \Delta$ *is well-defined,* $\forall \vec{\Delta}' \sqsubseteq \vec{\Delta} :$ $\mathcal{E}^{\downarrow}_{\mathsf{e}}\ \phi\ [\![e]\!]\ \vec{\Delta}' \sqsubseteq \Delta$, *and* Δ *$\vec{u}$-describes* v.

Proof. Let $\vec{u}, e, \vec{v}, \vec{\Delta}, \vec{\Delta}'$ be given such that $\vec{\Delta}$ $\vec{u}$-describes $\vec{v}$, $\vec{\Delta}' \sqsubseteq \vec{\Delta}$ and assume $\mathcal{E}\ [\![e]\!]\ \vec{v} = v \notin \{\mathsf{error}, \bot\}$. By Lemma B.1.1 we know that $\vec{v}, \vec{\Delta} \vdash e \hookrightarrow v : \Delta$ is well-defined, so we proceed by induction over the height of the inference tree, considering the syntax of e:

Case $e \equiv k$. Trivial.

Case $e \equiv x_i$. Immediate by Corollary B.1.3.

$$\frac{}{\vec{v},\vec{\Delta} \vdash k \hookrightarrow k : \{\}} \qquad \frac{}{\vec{v},\vec{\Delta} \vdash x_i \hookrightarrow v_i : \Delta_i} \qquad \frac{\vec{v},\vec{\Delta} \vdash e_1 \hookrightarrow \mathsf{true} : \Delta_1' \quad \vec{v},\vec{\Delta} \vdash e_2 \hookrightarrow v_2' : \Delta_2'}{\vec{v},\vec{\Delta} \vdash \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 \hookrightarrow v_2' : \Delta_2'}$$

$$\frac{\vec{v},\vec{\Delta} \vdash e_1 \hookrightarrow \mathsf{false} : \Delta_1' \quad \vec{v},\vec{\Delta} \vdash e_3 \hookrightarrow v_3' : \Delta_3'}{\vec{v},\vec{\Delta} \vdash \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 \hookrightarrow v_3' : \Delta_3'}$$

$$\frac{\vec{v},\vec{\Delta} \vdash e_1 \hookrightarrow v_1' : \Delta_1' \quad \vec{v},\vec{\Delta} \vdash e_2 \hookrightarrow v_2' : \Delta_2'}{\vec{v},\vec{\Delta} \vdash \texttt{cons } e_1\ e_2 \hookrightarrow (v_1'.v_2') : \{\uparrow(x) \mid \delta(x) \in \Delta_1' \cup \Delta_2'\}} \qquad \frac{\vec{v},\vec{\Delta} \vdash e_1 \hookrightarrow (v_1'.v_2') : \Delta'}{\vec{v},\vec{\Delta} \vdash \texttt{car } e_1 \hookrightarrow v_1' : \Delta'}$$

$$\frac{\vec{v},\vec{\Delta} \vdash e_1 \hookrightarrow (v_1'.v_2') : \Delta'}{\vec{v},\vec{\Delta} \vdash \texttt{cdr } e_1 \hookrightarrow v_2' : \Delta'} \qquad \frac{\begin{array}{c}\vec{v},\vec{\Delta} \vdash e_i \hookrightarrow v_i' : \Delta_i', i = 1,\ldots,n \\ (v_1',\ldots,v_n'),(\Delta_1',\ldots,\Delta_n') \vdash e^f \hookrightarrow v^f : \Delta^f\end{array}}{\vec{v},\vec{\Delta} \vdash f\ e_1 \ldots e_n \hookrightarrow v^f : \Delta^f}$$

Fig. 18. Inference system connecting evaluation and increasing size approximation

Case $e \equiv \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3$. Straightforward by induction and Corollary B.1.3.

Case $e \equiv \texttt{cons } e_1\ e_2$. Trivial, because $\mathcal{E}^{\downarrow}_{\mathsf{b}}$ $\texttt{cons}$ $\Delta_1\Delta_2 = \{\}$.

Case $e \equiv \texttt{car } e_1$ *and* $e \equiv \texttt{cdr } e_1$. By induction we know that $\Delta_1 = \mathcal{E}^{\downarrow}_{\mathsf{e}}\ \phi\ [\![e_1]\!]\vec{\Delta}' \sqsubseteq \Delta'$ and that Δ' $\vec{u}$-describes $(v_1'.v_2')$. Now if $\downarrow(x) \in \mathcal{E}^{\downarrow}_{\mathsf{e}}\ \phi\ [\![e]\!]\vec{\Delta}' = \{\downarrow(x) \mid \delta(x) \in \Delta_1\}$ then $\delta(x) \in \mathcal{E}^{\downarrow}_{\mathsf{e}}\ \phi\ [\![e_1]\!]\vec{\Delta} \sqsubseteq \Delta'$, so $\delta'(x) \in \Delta'$, which by the inference rule implies that $\downarrow(x) \in \downarrow\Delta' = \Delta$, proving that $\mathcal{E}^{\downarrow}_{\mathsf{e}}\ \phi\ [\![e]\!]\ \vec{\Delta}' \sqsubseteq \Delta$. Further, if $\downarrow(x_i) \in \Delta = \downarrow\Delta$ then $\delta(x_i) \in \Delta'$, that is, $(v_1'.v_2') \leq u_i$, which implies that $v_1' < u_i$ and $v_2' < u_i$, so Δ does indeed $\vec{u}$-describe v.

Case $e \equiv f\ e_1 \ldots e_n$. By induction we know that $\Delta_i'' = \mathcal{E}^{\downarrow}_{\mathsf{e}}\ \phi\ [\![e_i]\!]\vec{\Delta}' \sqsubseteq \Delta_i'$ and $\mathcal{E}^{\downarrow}_{\mathsf{e}}\ \phi\ [\![e^f]\!]\vec{\Delta}'' \sqsubseteq \Delta^f$ for $\vec{\Delta}'' = (\Delta_1'',\ldots,\Delta_n'')$, so $\mathcal{E}^{\downarrow}_{\mathsf{e}}\ \phi\ [\![e]\!]\vec{\Delta}' = \mathcal{E}^{\downarrow}_{\mathsf{e}}\ \phi\ [\![e^f]\!]\vec{\Delta}'' \sqsubseteq \Delta^f = \Delta$. Further, by induction we know that $\Delta = \Delta^f$ $\vec{u}$-describes $v^f = v$. □

By letting $\vec{u} = \vec{v}$ and $\vec{\Delta}' = \vec{\Delta} = \vec{\Delta}^{\downarrow}_{\mathsf{id}}$ in Lemma B.1.4 we find that $\mathcal{E}^{\downarrow}[\![e]\!] = \mathcal{E}^{\downarrow}_{\mathsf{e}}\ \phi\ [\![e]\!]\ \vec{\Delta} = \Delta'$ where Δ' $\vec{v}$-describes v because $\Delta' \sqsubseteq \Delta$ which implies

Corollary B.1.5. *$\mathcal{E}^{\downarrow}$ is safe.*

B.1.2 *Safety of increasing approximation.* The inference system shown in Figure 18 is used to derive a judgment $\vec{v},\vec{\Delta} \vdash e \hookrightarrow v : \Delta$ which informally states that if evaluating e using $\vec{v}$ for the free variables terminates, it yields v, and also that under dependency environment $\vec{\Delta}$, Δ decribes what e is increasingly dependent on. The inference system is similar to that of Figure 17, and as only the approximations are different, Lemma B.1.1 applies.

Definition B.1.6. Given $\bar{U} = \vec{u}_1,\vec{u}_2,\ldots \in (\mathit{Value}^n)^\omega$, $\bar{V} = \vec{v}_1,\vec{v}_2,\ldots \in (\mathit{Value}^m)^\omega$ and $W = w_1,w_2,\ldots \in \mathit{Value}^\omega$, we say that

—Δ $\bar{U}$-describes W iff for all $X \subseteq \{1,\ldots,n\}$

$$\begin{array}{lll} \sum_{i\in X} |\{u_{ji}\}| = \sum_{i\in X} |\{j \mid w_j > u_{ji}\}| = \infty & \Rightarrow \exists i \in X : & \uparrow(x_i) \in \Delta \quad \text{and} \\ \sum_{i\in X} |\{u_{ji}\}| = |\{w_j\}| = \infty & \Rightarrow \exists i \in X : \exists\delta : & \delta(x_i) \in \Delta. \end{array}$$

—$\vec{\Delta}$ $\bar{U}$-describes $\bar{V}$ iff Δ_i $\bar{U}$-describes $V_i = v_{1i}, v_{2i}, \ldots$ for all $i \in \{1,\ldots,m\}$.

Corollary B.1.7. *If $\Delta' \sqsupseteq \Delta$ and Δ $\bar{U}$-describes W then Δ' also $\bar{U}$-describes W. Further, if Δ $\bar{U}$-describes W, then $\Delta \sqcup^{\uparrow} \Delta''$ (cf. Figure 16) also $\bar{U}$-describes W for any Δ''.*

LEMMA B.1.8. *Let $\bar{U}$ be given. Now for all $e, \bar{V}, \vec{\Delta}$, if for all j $\mathcal{E}$ $[\![e]\!]$ $\vec{v}_j$ terminates without errors and $\vec{\Delta}$ $\bar{U}$-describes $\bar{V}$, then $\vec{v}_j, \vec{\Delta} \vdash e \hookrightarrow w_j : \Delta$ is well-defined, $\forall \vec{\Delta}' \sqsupseteq \vec{\Delta} : \mathcal{E}^{\uparrow}_{\mathsf{e}}\ \phi\ [\![e]\!]\ \vec{\Delta}' \sqsupseteq \Delta$, and Δ $\bar{U}$-describes W.*

PROOF. We let h denote the maximum height of the inference trees and consider two cases:

Case $h < \infty$. We proceed by induction over the maximum height of the inference trees for $j = 1, 2, \ldots$. In each case $\mathcal{E}^{\uparrow}_{\mathsf{e}}\ \phi\ [\![e]\!]\ \vec{\Delta}' \sqsupseteq \Delta$ is obvious, so we show just that Δ $\bar{U}$-describes W.

Base case $e \equiv k$. We find $w_j = k$, so $|\{j \mid w_j > u_{ji}\}| < \infty$ and $|\{w_j\}| < \infty$, which makes the implications vacuously true.

Base case $e \equiv x_i$. We find $w_j = v_{ji}$, and by the premise we see that $\mathcal{E}^{\uparrow}_{\mathsf{e}}\ \phi\ [\![e]\!]\ \vec{\Delta}' = \Delta'_i \sqsupseteq \Delta_i$ $\bar{U}$-describes $V_i = W$.

*Inductive case $e \equiv$ **if** e_1 **then** e_2 **else** e_3.* For given X, suppose wlog. that $\vec{v}_{j_\iota}, \vec{\Delta} \vdash e_1 \hookrightarrow \texttt{true}$ for an infinite subsequence $j_1, j_2, \ldots$ and $\sum_{i \in X} |\{u_{j_\iota i}\}| = \sum_{i \in X} |\{j_\iota \mid w_{j_\iota} > u_{j_\iota i}\}| = \infty$. Then the induction hypothesis can be applied to e_2. Similar reasoning applies if $\sum_{i \in X} |\{u_{j_\iota i}\}| = |\{w_{j_\iota}\}|$

Inductive case $e \equiv \texttt{cons}\ e_1\ e_2$. For given X, if $\sum_{i \in X} |\{u_{ji}\}| = \infty$ and $(\sum_{i \in X} |\{j \mid w_j > u_{ji}\}| = \infty$ or $|\{w_j\}| = \infty)$, then we can assume wlog. that $|\{w'_{1j}\}| = \infty$, where $(w'_{1j}.w'_{2j}) = w_j$, and apply the induction hypothesis to e_1. We then find $\exists i \in X : \uparrow(x_i) \in \Delta$ as required.

Inductive cases $e \equiv \texttt{car}\ e_1$ and $e \equiv \texttt{cdr}\ e_1$. For given X, if $\sum_{i \in X} |\{u_{ji}\}| = \infty$ and $(\sum_{i \in X} |\{j \mid w_j > u_{ji}\}| = \infty$ or $|\{w_j\}| = \infty)$, then the same is the case for $w_j.w'_2$ and $w'_1.w_j$, so the induction hypothesis can be applied.

Inductive case $e \equiv f\ e_1 \ldots e_n$. Applying the induction hypothesis to $e_1, \ldots, e_n$ we find that $(\Delta'_1, \ldots, \Delta'_n) = \vec{\Delta}'$ $\bar{U}$-describes $\bar{V}' = (v'_{11}, \ldots, v'_{1n}), (v'_{21}, \ldots, v'_{2n}), \ldots$ and $\Delta''_i = \mathcal{E}^{\uparrow}_{\mathsf{e}}\ \phi\ [\![e_i]\!]\ \vec{\Delta} \sqsupseteq \Delta'_i$, so we can apply the induction hypothesis to e^f, finding that $\mathcal{E}^{\uparrow}_{\mathsf{e}}\ \phi\ [\![f\ e_1 \ldots e_n]\!]\ \vec{\Delta}' = \mathcal{E}^{\uparrow}_{\mathsf{e}}\ \phi\ [\![e^f]\!]\ \vec{\Delta}'' \sqsupseteq \Delta^f$ and Δ^f $\bar{U}$-describes W.

Case $h = \infty$. For given X, if we can find an infinite subsequence $j_1, j_2, \ldots$ for which $\sum_{i \in X} |\{u_{j_\iota i}\}| = \sum_{i \in X} |\{j_\iota \mid w_{j_\iota} > u_{j_\iota i}\}| = \infty$ or $\sum_{i \in X} |\{u_{j_\iota i}\}| = |\{w_{j_\iota}\}| = \infty$ and the maximum height of the inference trees for this subsequence is $h_\iota < \infty$, we proceed as for $h < \infty$.

Otherwise, the number of function call inference steps occurring in an inference tree must be unbounded, and thus there must be a function f which loops an unbounded number of times. Further, as $|\{w_j\}| = \infty$, this loop must involve a recursive increase, and as each inference tree is of finite height, the loop must include at least one **if**-expression in which a free variable of the condition depends on x_i for some $i \in X$, so $\uparrow(x_i) \in \mathcal{E}^{\uparrow}_{\mathsf{e}}\ \phi\ [\![e]\!] \vec{\Delta}$. □

C. DETAILED ANALYSIS RESULTS

Appendix C available only online. You should be able to get the online-only appendix from the citation page for this article:

http://www.acm.org/toplas

Alternative instructions on how to obtain online-only appendices are given on the back inside cover of current issues of ACM TOPLAS or on the ACM TOPLAS web page:

http://www.acm.org/toplas

ACKNOWLEDGMENTS

We would like to thank Niels H. Christensen, Olivier Danvy, Julia L. Lawall, Jens Peter Secher, Walid Taha and anonymous reviewers for valuable comments and suggestions for improvements of this paper.

REFERENCES

Abel, A. and Altenkirch, T. 1999. A semantical analysis of structural recursion. In *Abstracts of the Fourth International Workshop on Termination WST'99.* unpublished, Dagstuhl, Germany, 24–25.

Aho, A. V., Sethi, R., and Ullman, J. D. 1986. *Compilers: Principles, Techniques, and Tools.* Addison Wesley, Reading, Mass.

Andersen, P. H. and 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)*, R. Cousot and D. A. Schmidt, Eds. Lecture Notes in Computer Science, vol. 1145. Springer-Verlag, Berlin, 67–82.

Arts, T. and Giesl, J. 1997. Proving innermost termination automatically. In *Proceedings Rewriting Techniques and Applications RTA'97.* Lecture Notes in Computer Science, vol. 1232. Springer-Verlag, Berlin, 157–171.

Bawden, A. 1988. Reification without evaluation. In *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming.* ACM Press, ACM Press, New York, 342–351.

Ben-Amram, A. M. 2002. General size-change termination and lexicographic descent. In *The Essence of Computation: Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones*, T. Mogensen, D. Schmidt, and I. H. Sudborough, Eds. Lecture Notes in Computer Science, vol. 2566. Springer-Verlag, Berlin, 3–17.

Berlin, A. and Weise, D. 1990. Compiling scientific code using partial evaluation. *IEEE Computer 23,* 12, 25–37.

Birkedal, L. and Welinder, M. 1994. Hand-writing program generator generators. In *Proceedings of the 6th International Symposium on Programming Language Implementation and Logic Programming (PLILP '94)*, M. Hermenegildo and J. Penjam, Eds. Springer-Verlag, Berlin, 198–214.

Bondorf, A. 1991. Automatic autoprojection of higher order recursive equations. *Sci. Comput. Program. 17,* 3–34.

Bondorf, A. and Danvy, O. 1991. Automatic autoprojection of recursive equations with global variables and abstract data types. *Sci. Comput. Program. 16,* 2, 151–195.

Bondorf, A. and Jørgensen, J. 1993. Efficient analysis for realistic off-line partial evaluation: Extended version. Tech. Rep. 93/4, DIKU, University of Copenhagen, Denmark, Copenhagen, Denmark. Mar.

Brauburger, J. 1997. Automatic termination analysis for partial functions using polynomial orderings. In *Static Analysis Symposium.* Lecture Notes in Computer Science, vol. 1302. Springer-Verlag, Berlin, 330–344.

Burstall, R. M. and Darlington, J. 1977. A transformation system for developing recursive programs. *Journal of the ACM 24,* 1 (Jan.), 44–67.

Cai, J., Facon, P., Henglein, F., Paige, R., and Schonberg, E. 1991. Type analysis and data structure selection. In *Constructing Programs From Specifications*, B. Möller, Ed. North-Holland, Amsterdam, 126–164.

Chin, W.-N. and Khoo, S.-C. 2002. Calculating sized types. *Journal of Higher-Order and Symbolic Computation 14,* 2/3, 261–300.

Chin, W.-N., Khoo, S.-C., and Lee, T.-W. 1998. Synchronisation analysis to stop tupling. In *Programming Languages and Systems (ESOP'98)*. Lecture Notes in Computer Science, vol. 1381. Springer-Verlag, Lisbon, 75–89.

Christensen, N. H., Glück, R., and Laursen, S. 2000. Binding-time analysis in partial evaluation: One size does *Not* fit all. In *PSI'99*, D. Bjørner, M. Broy, and A. Zamulin, Eds. Lecture Notes in Computer Science, vol. 1755. Springer-Verlag, Berlin Heidelberg, 80–92.

Colón, M. A. and Sipma, H. B. 2002. Practical methods for proving program termination. In *Conference on Computer-Aided Verification (CAV)*, E. Brinksma and K. G. Larsen, Eds. Lecture Notes in Computer Science, vol. 2404. Springer-Verlag, Berlin, 442–454.

Consel, C. 1993. A tour of Schism: A partial evaluation system for higher-order applicative languages. In *ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation*. ACM Press, Copenhagen, Denmark, 66–77.

Consel, C. and Danvy, O. 1993. Tutorial notes on partial evaluation. In *ACM Symposium on Principles of Programming Languages*. ACM Press, Charleston, South Carolina, 493–501.

Consel, C. and Noël, F. 1996. A general approach for run-time specialization and its application to C. In *ACM Symposium on Principles of Programming Languages*. ACM Press, New York, NY, USA, 145–156.

Coquand, C. 2001. The interactive theorem prover Agda. http://www.cs.chalmers.se/~catarina/agda/.

Corbett, J., Dwyer, M., Hatcliff, J., Pasareanu, C., Robby, Laubach, S., and Zheng, H. 2000. Bandera: Extracting finite-state models from Java source code. In *22nd International Conference on Software Engineering*. IEEE Press, Limerick, Ireland, 439–448.

Cousot, P. and Cousot, R. 1977. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th POPL, Los Angeles, CA*. ACM Press, New York, NY, 238–252.

Danvy, O., Glück, R., and Thiemann, P., Eds. 1996. *Partial Evaluation*. Lecture Notes in Computer Science, vol. 1110. Springer-Verlag.

Danvy, O. and Malmkjær, K. 1988. Intensions and extensions in a reflective tower. In *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming*. ACM Press, ACM Press, New York, 327–341.

Das, M. 1998. Partial evaluation using dependence graphs. Ph.D. thesis, University of Wisconsin-Madison.

Das, M. and Reps, T. 1996. BTA termination using CFL-reachability. Tech. Rep. 1329, Computer Science Department, University of Wisconsin-Madison.

De Schreye, D., Glück, R., Jørgensen, J., Leuschel, M., Martens, B., and Sørensen, M. H. B. 1999. Conjunctive partial deduction: Foundations, control, algorithms, and experiments. *Journal of Logic Programming 41,* 2&3, 231–277.

des Rivières, J. and Smith, B. C. 1984. The implementation of procedurally reflective languages. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*. ACM Press, ACM Press, New York, 331–347.

Fradet, P. and Le Métayer, D. 1997. Shape types. In *ACM Symposium on Principles of Programming Languages*. ACM Press, Paris, France, 27–39.

Friedman, D. P. and Wand, M. 1984. Reification: Reflection without metaphysics. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*. ACM Press, ACM Press, New York, 348–355.

Futamura, Y. 1999a. Partial evaluation of computation process – an approach to a compiler-compiler. *Higher-Order and Symbolic Computation 12,* 4, 381–391. Reprinted from Systems · Computers · Controls 2(5), 1971.

Futamura, Y. 1999b. Partial evaluation of computation process, revisited. *Higher-Order and Symbolic Computation 12,* 4, 377–380.

Gallagher, J. and Bruynooghe, M. 1990. Some low-level source transformations for logic programs. In *Proceedings of the Second Workshop on Meta-Programming in Logic, April 1990, Leuven, Belgium*, M. Bruynooghe, Ed. Department of Computer Science, KU Leuven, Belgium, 229–246.

GALLAGHER, J. P. 1993. Tutorial on specialisation of logic programs. In *Proceedings of PEPM'93, the ACM Sigplan Symposium on Partial Evaluation and Semantics-Based Program Manipulation*. ACM Press, Copenhagen, Denmark, 88–98.

GANZ, S., SABRY, A., AND TAHA, W. 2001. Macros as multi-stage computations: Type-safe, generative, binding macros in MacroML. In *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming (ICFP-01)*, C. Norris and J. J. B. Fenwick, Eds. ACM SIGPLAN notices, vol. 36, 10. ACM Press, New York, 74–85.

GHIYA, R. AND 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*. ACM Press, Florida, 1–15.

GIESL, J. 1995. Termination analysis for functional programs using term orderings. In *Proceedings of the Second International Static Analysis Symposium (SAS'95)*. Lecture Notes in Computer Science, vol. 983. Springer-Verlag, Glasgow, Scotland.

GLENSTRUP, A., MAKHOLM, H., AND SECHER, J. P. 1999. C-Mix — specialization of C programs. See Hatcliff et al. [1999], 108–154.

GLENSTRUP, A. J. 1999. Terminator II: Stopping partial evaluation of fully recursive programs. M.S. thesis, DIKU, University of Copenhagen, DK-2100 Copenhagen Ø.

GLENSTRUP, A. J. AND 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*. Lecture Notes in Computer Science. Springer-Verlag, Berlin.

GLÜCK, R., NAKASHIGE, R., AND ZÖCHLING, R. 1995. Binding-time analysis applied to mathematical algorithms. In *System Modelling and Optimization*, J. Doležal and J. Fidler, Eds. Chapman & Hall, London, 137–146.

GLÜCK, R. AND SØRENSEN, M. H. 1996. A roadmap to metacomputation by supercompilation. In *Partial Evaluation*, O. Danvy, R. Glück, and P. Thiemann, Eds. Lecture Notes in Computer Science, vol. 1110. Springer-Verlag, Berlin, 137–160.

GRANT, B., MOCK, M., PHILIPOSE, M., CHAMBERS, C., AND EGGERS, S. J. 2000. DyC: An expressive annotation-directed dynamic compiler for C. *Theoretical Computer Science 248,* 1–2, 147–199.

GROBAUER, B. 2001. Topics in semantics-based program manipulation. Ph.D. thesis, BRICS, Department of Computer Science, University of Aarhus, Aarhus, Denmark. DS–01–6.

HATCLIFF, J., MOGENSEN, T., AND THIEMANN, P., Eds. 1999. *Partial Evaluation: Practice and Theory. Proceedings of the 1998 DIKU International Summerschool*. Lecture Notes in Computer Science, vol. 1706. Springer-Verlag.

HOLST, C. K. 1988. Poor man's generalization. Tech. rep., DIKU, University of Copenhagen.

HOLST, C. K. 1991. Finiteness analysis. In *Functional Programming Languages and Computer Architecture*, J. Hughes, Ed. Lecture Notes in Computer Science, vol. 523. Springer-Verlag, Berlin, DE, 473–495.

HOLST, C. K. AND LAUNCHBURY, J. 1991. Handwriting cogen to avoid problems with static typing. In *Draft Proceedings, Fourth Annual Glasgow Workshop on Functional Programming*. Glasgow University, Skye, Scotland, 210–218.

HUDAK, P. 1996. Building domain specific embedded languages. *ACM Computing Surveys 28A*, (electronic).

HUGHES, J. 1996. Type specialisation for the λ-calculus; or a new paradigm for partial evaluation based on type inference. See Danvy et al. [1996], 183–215.

HUGHES, J., PARETO, L., AND SABRY, A. 1996. Proving the correctness of reactive systems using sized types. In *ACM Symposium on Principles of Programming Languages*. ACM Press, St. PetersBurg FLA USA, 410–423.

JEFFERSON, S. AND FRIEDMAN, D. P. 1996. A simple reflective interpreter. *Lisp and Symbolic Computation 9,* 2/3 (May/June), 181–202.

JONES, N. D. 1996. What *Not* to do when writing an interpreter for specialisation. In *Partial Evaluation*, O. Danvy, R. Glück, and P. Thiemann, Eds. Lecture Notes in Computer Science, vol. 1110. Springer-Verlag, Berlin, 216–237. International Seminar at Dagstuhl Castle, Germany.

Jones, N. D., Gomard, C. K., and Sestoft, P. 1993. *Partial Evaluation and Automatic Program Generation*. International Series in Computer Science. Prentice Hall International, New York. ISBN number 0-13-020249-5 (pbk).

Jones, N. D. and Nielson, F. 1994. Abstract interpretation: a semantics-based tool for program analysis. In *Handbook of Logic in Computer Science*. Oxford University Press, New York. 527–629.

Kieburtz, R. B., McKinney, L., Bell, J., Hook, J., Kotov, A., Lewis, J., Oliva, D., Sheard, T., Smith, I., and Walton, L. 1996. A software engineering experiment in software component generation. In *18th International Conference in Software Engineering*. IEEE Computer Society, Los Alamitos, USA, 542–553.

Launchbury, J. 1991. *Projection Factorisations in Partial Evaluation*. Distinguished Dissertations in Computer Science. Cambridge University Press, Cambridge.

Lawall, J. L. and Thiemann, P. 1997. Sound specialization in the presence of computational effects. In *Proceedings of the 3rd International Symposium on Theoretical Aspects of Computer Software (TACS'97)*, M. Abadi and T. Ito, Eds. Number 1281 in Lecture Notes in Computer Science. Springer-Verlag, Berlin, 165–190.

Lee, C. S. 2002a. Finiteness analysis in polynomial time. In *Static Analysis: 9th International Symposium, SAS 2002*, M. Hermenegildo and G. Puebla, Eds. Lecture Notes in Computer Science, vol. 2477. Springer-Verlag, Berlin, 493–508.

Lee, C. S. 2002b. Program termination analysis and termination of offline partial evaluation. Ph.D. thesis, University of Western Australia. Aug.

Lee, C. S., Jones, N. D., and Ben-Amram, A. M. 2001. The size-change principle for program termination. In *ACM Symposium on Principles of Programming Languages*. Vol. 28. ACM Press, New York, 81–92.

Leuschel, M. 1998. On the power of homeomorphic embedding for online termination. In *Static Analysis. Proceedings (Pisa, Italy)*, G. Levi, Ed. Lecture Notes in Computer Science, vol. 1503. Springer-Verlag, Berlin, 230–245.

Leuschel, M. and Bruynooghe, M. 2002. Logic program specialisation through partial deduction: Control issues. *Theory and Practice of Logic Programming 2,* 4–5 (July–Sept.), 461–515.

Lindenstrauss, N. and Sagiv, Y. 1997. Automatic termination analysis of logic programs (with detailed experimental results). Unpublished.

Lindenstrauss, N., Sagiv, Y., and Serebrenik, A. 1997. Termilog: A system for checking termination of queries to logic programs. In *Computer Aided Verification, 9th International Conference, CAV '97, Haifa, Israel, June 22–25, 1997*, O. Grumberg, Ed. Lecture Notes in Computer Science, vol. 1254. Springer-Verlag, Berlin, 444–447.

Liu, Y. A. 2000. Efficiency by incrementalization: An introduction. *Journal of Higher-Order and Symbolic Computation 13,* 4, 289–313.

Lloyd, J. W. and Shepherdson, J. C. 1991. Partial evaluation in logic programming. *Journal of Logic Programming 11*, 217–242.

McNamee, D., Walpole, J., Pu, C., Cowan, C., Krasic, C., Goel, A., Wagle, P., Consel, C., Muller, G., and Marlet, R. 2001. Specialization tools and techniques for systematic optimization of system software. *ACM Transactions on Computer Systems 19,* 2, 217–251.

Mogensen, T. 1988. Partially static structures in a self-applicable partial evaluator. In *Partial Evaluation and Mixed Computation*, D. Bjørner, A. Ershov, and N. Jones, Eds. Elsevier Science Publishers, North-Holland, 325–347.

Mogensen, T. Æ. 2000. Glossary for partial evaluation and related topics. *Journal of Higher-Order and Symbolic Computation 13,* 4 (Dec.), 355–368.

Moggi, E., Taha, W., Benaissa, Z.-E.-A., and Sheard, T. 1999. An idealized MetaML: Simpler, and more expressive. In European Symposium on Programming. *Lecture Notes in Computer Science 1576*, 193–207.

Nielson, F. and Nielson, H. R. 1996. Operational semantics of termination types. *Nordic Journal of Computing 3*, 144–187.

Pareto, L. 2000. Types for crash prevention. Ph.D. thesis, Chalmers University of Technology and Göteborg University, Göteborg, Sweden.

PÉTER, R. 1951 (1976). *Rekursive Funktionen (Recursive Functions).* Académiai Kiadó, Budapest. (Academic Press, New York).

POLETTO, M., HSIEH, W. C., ENGLER, D. R., AND KAASHOEK, M. F. 1999. ‘C and tcc: A language and compiler for dynamic code generation. *ACM Transactions on Programming Languages and Systems 21,* 2 (Mar.), 324–369.

PRASAD SISTLA, A., VARDI, M. Y., AND WOLPER, P. 1987. The complementation problem for Büchi automata with applications to temporal logic. *Theoretical Computer Science 49*, 217–237.

RAMSEY, F. P. 1930. On a problem of formal logic. In *Proceedings of the London Mathematical Society*. Vol. 30. Cambridge University Press, Cambridge, 264–285.

SAGONAS, K. F., SWIFT, T., AND WARREN, D. S. 1994. XSB as an efficient deductive database engine. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data, Minneapolis, Minnesota, May 24–27, 1994*, R. T. Snodgrass and M. Winslett, Eds. ACM Press, New York, 442–453.

SCHULTZ, U. P. 2001. Partial evaluation for class-based object-oriented languages. In *Programs as Data Objects (PADO-II).* Lecture Notes in Computer Science, vol. 2053. Springer-Verlag, Berlin, 173–197.

SESTOFT, P. 2001. Bibliography on partial evaluation and mixed computation. Tech. rep., DIKU, University of Copenhagen, Denmark.

SHEARD, T. 1999. Using MetaML: A staged programming language. In Third International School in Advanced Functional Programming, Braga, Portugal, September 12-19, 1998, Revised Lectures, S. D. Swierstra, P. R. Henriques, and J. N. Oliveira, Eds. *Lecture Notes in Computer Science 1608*, 207–239.

SONG, L. AND FUTAMURA, Y. 2000. A new termination approach for specialization. In *Semantics, Applications, and Implementation of Program Generation*, W. Taha, Ed. Lecture Notes in Computer Science, vol. 1924. Springer-Verlag, Berlin, 72–91.

SØRENSEN, M. H. AND GLŪCK, R. 1995. An algorithm of generalization in positive supercompilation. In *Logic Programming: Proceedings of the 1995 International Symposium*, J. Lloyd, Ed. MIT Press, Cambridge, MA, 465–479.

SPEIRS, C., SOMOGYI, Z., AND SØNDERGAARD, H. 1997. Termination analysis for Mercury. In *Static Analysis, Proceedings of the 4th International Symposium, SAS '97, Paris, France, Sep 8–19, 1997*, P. V. Hentenryck, Ed. Lecture Notes in Computer Science, vol. 1302. Springer-Verlag, Berlin, 160–171.

SPERBER, M. AND THIEMANN, P. 2000. Generation of LR parsers by partial evaluation. *ACM Transactions on Programming Languages and Systems 22,* 2 (Mar.), 224–264.

TAHA, W. 1999a. Multi-stage programming: Its theory and applications. Ph.D. thesis, Oregon Graduate Institute of Science and Technology.

TAHA, W. 1999b. A sound reduction semantics for untyped CBN multi-stage computation. or, the theory of MetaML is non-trival. *ACM SIGPLAN Notices 34,* 11 (Nov.), 34–43. Extended abstract.

TAHA, W., Ed. 2000. *Semantics, Applications, and Implementation of Program Generation.* Lecture Notes in Computer Science, vol. 1924. Springer-Verlag, Montréal.

TAHA, W., MAKHOLM, H., AND HUGHES, J. 2001. Tag elimination and Jones-optimality. In *Programs as Data Objects (PADO-II).* Lecture Notes in Computer Science, vol. 2053. Springer-Verlag, Berlin, 257–275. http://cs-www.cs.yale.edu/homes/taha/publications/preprints/pado00.dvi.

TAHA, W. AND SHEARD, T. 2000. MetaML and multi-stage programming with explicit annotations. *Theoretical Computer Science 248,* 1–2 (Oct.), 211–242.

THIEMANN, P. 1997. A unified framework for binding-time analysis. In *TAPSOFT '97: Theory and Practice of Software Development, Lille, France*, M. Bidoit and M. Dauchet, Eds. Lecture Notes in Computer Science, vol. 1214. Springer-Verlag, Berlin, 742–756.

THIEMANN, P. 1999. Aspects of the PGG system: Specialization for standard scheme. See Hatcliff et al. [1999], 412–432.

Turchin, V. F. 1979. A supercompiler system based on the language Refal. *SIGPLAN Notices 14,* 2 (Feb.), 46–54.

Wadler, P. 1988. Deforestation: Transforming programs to eliminate trees. In *ESOP'88. 2nd European Symposium on Programming, Nancy, France*, H. Ganzinger, Ed. Lecture Notes in Computer Science, vol. 300. Springer-Verlag, Berlin, 344–358.

Wand, M. and Friedman, D. P. 1988. The mystery of the tower revealed: A non-reflective description of the reflective tower. *Lisp and Symbolic Computation 1,* 1 (June), 11–37. Reprinted in *Meta-Level Architectures and Reflection* (P. Maes and D. Nardi, eds.) North-Holland, Amsterdam, 1988, pp. 111–134. Preliminary version appeared in *Proc. 1986 ACM Conf. on Lisp and Functional Programming,* 298–307.

Xi, H. 2002. Dependent types for program termination verification. *Journal of Higher-Order and Symbolic Computation 15,* 1, 91–132.

Received ???-Month ????; ???-Month ????; accepted ???-Month ????

THIS DOCUMENT IS THE ONLINE-ONLY APPENDIX TO:

Termination Analysis and Specialization-Point Insertion in Off-line Partial Evaluation

ARNE JOHN GLENSTRUP and NEIL D. JONES
DIKU, University of Copenhagen

ACM Transactions on Programming Languages and Systems, Vol. ?, No. ?, January 2003.

C. DETAILED ANALYSIS RESULTS

An overview of the results of the binding-time analysis, given binding-time patterns for each goal function, are shown in Tables I–II. By manually 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. This is because during the reordering of the list, our rather crude size approximations lose track of the list sizes. One could patch on this problem by passing around a measure 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, *assocrw*, 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.

The example program *nestimeql* shows one shortcoming of the size approximation function $\mathcal{E}^{\uparrow}$: for the function

```
immatcopy x = if x = []  then []
                         else cons (car x) (immatcopy (cdr x))
```

our size approximation 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 a `revapp` call in *permute*, and a `reverse` call in *shuffle*.

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

The programs and results are given in detail in the following sections. The binding times are given by

Permission to make digital/hard copy of all or part of this material without fee for personal or classroom use provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.

Program	*Goal BT*	*Result*	*Optimal result*	*Conservative*	*Program description*
int-loop	s s d			no	interpret a LOOP program: int($prog$, $bound$, $input$)
	s d d	SP	SP	no	
int-while-d	s d	SP, G	SP, G	no	interpret a WHILE program with dynamic scoping
int-while-s	s d	SP	SP	no	interpret a WHILE program, cf. Figure 2
lambdaint	s	SP, G	SP, G	no	interpret a λ expression
parsexp	s			no	parse a numerical expression
turing	s d	SP	SP	no	run a program on a Turing machine: turing($prog$, $tape$)
ack	s d	SP	SP	no	compute Ackermann's function: ack(m, n)
binom	s d			no	compute the binomial function using unary numbers
	d s	SP	SP	no	
gcd-1	s d	SP	SP	no	compute greatest common divisor
gcd-2	s d	SP	SP	no	compute greatest common divisor, swapping arguments
graphcol-1	d s	SP	SP	no	colour a graph with colours from a set (using The Trick)
	s d	SP	SP	no	
graphcol-2	d s	SP	SP	no	colour a graph with colours from a set (tail recursive, using The Trick)
	s d	SP	SP	no	
graphcol-3	d s	SP	SP	no	colour a graph with colours from a set (tail recursive, without The Trick)
	s d	SP	SP	no	
match	s d	SP	SP	no	match naïvely sublist pattern p in a symbol list s
	d s			no	
power	d s	SP	SP	no	compute the power function: x^a
	s d	SP	SP	no	
reach	d d s	SP	SP	no	compute a path from node x to node y in graph G
	d s s	SP	SP	no	
	s d s	SP	SP	no	
rematch	s d	SP, G	SP, G	no	match regular expression r in a string s
	d s	SP	SP	no	
strmatch	s d	SP, G	SP, G	no	match naïvely string pattern p in a string s
	d s			no	
typeinf	s s			no	infer a type in the typed lambda calculus
add	s d	SP, G	SP, G	no	add two unary numbers
	d s			no	
addlists	s d			no	add two lists elementwise
anchored	s d			no	function where y is anchored in x
	d s	SP, G	SP, G	no	
append	s d			no	append two lists: append($list1$, $list2$)
assocrw	s	SP, G		yes	make an associative expression right-recursive
badd	s d	SP	SP	no	add numbers using a quasiterminating addition function
contrived-1	s d			no	exercise the analyser with a small contrived program
contrived-2	s d	SP	SP	no	insert a specialisation point in a static function
decrease	s			no	recursive calls with a decreasing argument
deeprev	s			no	recursively reverse all list elements in a data structure
disjconj	s			no	check for disjunctive and conjunctive terms
duplicate	s			no	build a copy of a list
equal	s	SP	SP	no	recursive calls with a constant argument
evenodd	s			no	check whether a unary number is even or odd
exponential	s s	SP	SP	no	functions generating exponentially many SCGs

s = static, d = dynamic,
SP = insert specialisation point(s), G = generalise variable(s) to ensure termination

Table I. Results of binding-time analysis, part I.

Program	*Goal BT*	*Result*	*Optimal result*	*Conservative*	*Program description*
fold	d s			no	fold a fixed operator over a list: fold(*elt*, *list*)
	s d	SP	SP	no	
game	s s s			no	play a small game with two players
increase	s	SP, G	SP, G	no	recursive calls with increasing argument
intlookup	d s	SP	SP	no	look up variable values like in an interpreter
letexp	s s	SP, G	SP, G	no	example using the let construction
list	s			no	check for a list data structure
lte	s d			no	check whether x is a substructure of y
	d s	SP	SP	no	
map	s			no	map a fixed function along a list
member	d s			no	check for list membership: member(*element*, *list*)
	s d	SP	SP	no	
mergelists	s d	SP	SP	no	merge two sorted lists
mul	s d	SP	SP	no	multiply two unary numbers
naiverev	s			no	naïve reverse: append reversed tail to head element
nestdec	s			no	nested call returns cdr of its argument
nesteql	s	SP	SP	no	nested call returns a copy of its argument
nestimeql	s	SP, G	SP	yes	nested call to a function that constructs a copy of its argument only from constants
nestinc	s	SP, G	SP, G	no	nested call returns a cons cell containing its argument
nolexicord	s s s s s s			no	a terminating function with no lexicographical ordering, cf. Example 9.2.1
	s s s s s d			no	
ordered	s			no	check whether a list is ordered
overlap	s d	SP	SP	no	check for non-empty set intersection
permute	s	SP, G	SP	yes	compute all the permutations of a list
revapp	s d			no	reverse *list1* and append to *list2*: revapp(*list1*, *list2*)
select	s			no	pick out an element and cons it onto the remaining list
shuffle	s	SP, G		yes	shuffle a list
sp1	s d	SP	SP	no	mutual recursion requiring specialisation points
subsets	s			no	compute all subsets of a set
thetrick	s d	SP	SP	no	example using the trick for dynamic if conditionals
	d s	SP, G	SP, G	no	
vangelder	s d	SP	SP	no	quasiterminating example invented by Van Gelder
mergesort	s	SP, G		yes	sort list by splitting, recursive sorting, and merging
minsort	s	SP, G		yes	sort list: extract min elt, cons it onto the sorted rest
quicksort	s	SP, G		yes	sort list by splitting by size, sorting and appending

s = static, d = dynamic,
SP = insert specialisation point(s), G = generalise variable(s) to ensure termination

Table II. Results of binding-time analysis, part II.

B: Static and of bounded variation

S: Static but possibly of unbounded variation

D: Dynamic

and are shown before propagating the effects of specialization points. In some cases this would change more variables from B or S to D.

C.1 Interpreters

C.1.1 *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)                                   ; 1
  (if (equal? (car e) 3) ; basefcn
      (let* ((v1 (eeval (car (cdr (cdr e))) ns vs l p))         ; 2
             (v2 (eeval (car (cdr (cdr (cdr e)))) ns vs l p)))  ; 3
        (apply (car (cdr e)) v1 v2))                            ; 4
  (if (equal? (car e) 4)  ; if
      (if (equal? (eeval (car (cdr e)) ns vs l p) 'T)           ; 5
          (eeval (car (cdr (cdr e))) ns vs l p)                 ; 6
          (eeval (car (cdr (cdr (cdr e)))) ns vs l p))          ; 7
  (if (equal? (car e) 5)  ; ==
      (if (equal? (eeval (car (cdr e)) ns vs l p)               ; 8
                  (eeval (car (cdr (cdr e))) ns vs l p))        ; 9
          'T
          'F)
                         ; call
  (let* ((ef (lookbody (car (cdr e)) p))                        ; 10
         (nf (lookname (car (cdr e)) p))                        ; 11
         (v  (eeval (car (cdr (cdr e))) ns vs l p)))            ; 12
    (if (equal? l '()) '()
        (eeval ef (cons nf '()) (cons v '()) (cdr l) p)))))))))  ; 13

(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)))
```

```
Parameter binding times:
  apply:     op : B  v1 : D  v2 : D
  lookname:  f : B  p : B
  lookbody:  f : B  p : B
  lookvar:   x : B  ns : B  vs : D
  eeval:     e : B  ns : B  vs : D  l : B  p : B
  run:       p : B  l : B  input : D
Specialisation points:
None.
----------------------------

Parameter binding times:
  apply:     op : B  v1 : D  v2 : D
  lookname:  f : B  p : B
  lookbody:  f : B  p : B
  lookvar:   x : B  ns : B  vs : D
  eeval:     e : B  ns : B  vs : D  l : D  p : B
  run:       p : B  l : D  input : D
Specialisation points:
  Call 13 in eeval to eeval
```

C.1.2 *int-while-dynscope*

```
;;; Simple Scheme interpreter with dynamic scoping

(define (run data program)
  (evalexp (lookup-body 'main program) ; 1, 2
        (lookup-paramnames 'main program) ; 3
        data program))

(define (function? funname program)
  (and (pair? program) (or (eq? funname (caadar program))
                           (function? funname (cdr program)))))

(define (lookup-paramnames funname program)
  (if (eq? funname (caadar program)) (cdadar program)
      (lookup-paramnames funname (cdr program))))

(define (lookup-body funname program)
  (if (eq? funname (caadar program)) (caddar program)
      (lookup-body funname (cdr program))))

(define (variable? varname names)
  (and (pair? names)
       (or (eq? varname (car names)) (variable? varname (cdr names)))))

(define (lookup-value varname names values)
  (if (eq? varname (car names)) (car values)
      (lookup-value varname (cdr names) (cdr values))))

(define (evalexp exp names values program)
  (cond
   ((list? exp)
    (case (car exp)
      ((QUOTE) (cadr exp))
      ((LET LET*)
       (let* ((value
               (evalexp (car (cdaadr exp)) names values program))) ; 1
         (evalexp (caddr exp)                                     ; 2
                  (cons (caaadr exp) names)
                  (cons value values)
                  program)))
      ((IF)
       (if (evalexp (cadr exp) names values program)              ; 3
           (evalexp (caddr exp) names values program)             ; 4
           (evalexp (cadddr exp) names values program)))          ; 5
      (else
       (if (function? (car exp) program)                          ; 6
           (evalexp (lookup-body (car exp) program)               ; 7, 8
                    (append ;; DYNAMIC SCOPING
                     (lookup-paramnames (car exp) program)        ; 9
                     names) ;; DYNAMIC SCOPING
                    (append ;; DYNAMIC SCOPING
                     (argvals (cdr exp) names values program)     ; 10
                     values) ;; DYNAMIC SCOPING
                    program)
           ;; else it must be a base function
           (apply (eval (car exp) (scheme-report-environment 5))
                  (argvals (cdr exp) names values program))))))   ; 11
   ((variable? exp names)                                         ; 12
    (lookup-value exp names values))                              ; 13
   (else ;; it must be a constant
    exp)))

(define (argvals exps names values program)
  (if (null? exps) '()
      (cons (evalexp (car exps) names values program)
            (argvals (cdr exps) names values program))))

Parameter binding times:
  argvals:            exps : B  names : S  values : D  program : B
  evalexp:            exp : B  names : S  values : D  program : B
  lookup-value:       varname : B  names : S  values : D
  variable?:          varname : B  names : S
  lookup-body:        funname : B  program : B
  lookup-paramnames:  funname : B  program : B
  function?:          funname : B  program : B
  run:                data : D  program : B
Specialisation points:
  Call 1 in variable? to variable?
  Call 1 in lookup-value to lookup-value
  Call 7 in evalexp to evalexp
```

C.1.3 *int-while-statscope*

```
;;; Simple Scheme interpreter with static lexical scoping

(define (run data program)
  (evalexp (lookup-body 'main program)    ; 1, 2
        (lookup-paramnames 'main program) ; 3
        data program))

(define (function? funname program)
  (and (pair? program) (or (eq? funname (caadar program))
                           (function? funname (cdr program)))))

(define (lookup-paramnames funname program)
  (if (eq? funname (caadar program)) (cdadar program)
      (lookup-paramnames funname (cdr program))))

(define (lookup-body funname program)
  (if (eq? funname (caadar program)) (caddar program)
      (lookup-body funname (cdr program))))

(define (variable? varname names)
  (and (pair? names)
       (or (eq? varname (car names)) (variable? varname (cdr names)))))

(define (lookup-value varname names values)
  (if (eq? varname (car names)) (car values)
      (lookup-value varname (cdr names) (cdr values))))

(define (evalexp exp names values program)
  (cond
   ((list? exp)
    (case (car exp)
      ((QUOTE) (cadr exp))
      ((LET LET*)
       (let* ((value
               (evalexp (car (cdaadr exp)) names values program))) ; 1
         (evalexp (caddr exp)                                       ; 2
                  (cons (caaadr exp) names)
                  (cons value values)
                  program)))
      ((IF)
       (if (evalexp (cadr exp) names values program)               ; 3
           (evalexp (caddr exp) names values program)              ; 4
           (evalexp (cadddr exp) names values program)))           ; 5
      (else
       (if (function? (car exp) program)                           ; 6
           (evalexp (lookup-body (car exp) program)                ; 7, 8
                    (lookup-paramnames (car exp) program)          ; 9
                    (argvals (cdr exp) names values program)       ; 10
                    program)
           ;; else it must be a base function
           (apply (eval (car exp) (scheme-report-environment 5))
                  (argvals (cdr exp) names values program)))))))   ; 11
   ((variable? exp names)                                          ; 12
    (lookup-value exp names values))                               ; 13
   (else ;; it must be a constant
    exp)))

(define (argvals exps names values program)
  (if (null? exps) '()
      (cons (evalexp (car exps) names values program)
            (argvals (cdr exps) names values program))))

Parameter binding times:
  argvals:            exps : B  names : B  values : D  program : B
  evalexp:            exp : B  names : B  values : D  program : B
  lookup-value:       varname : B  names : B  values : D
  variable?:          varname : B  names : B
  lookup-body:        funname : B  program : B
  lookup-paramnames:  funname : B  program : B
  function?:          funname : B  program : B
  run:                data : D  program : B
Specialisation points:
  Call 7 in evalexp to evalexp
```

C.1.4 *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))


Parameter binding times:
  app->e2:    e : S
  app->e1:    e : S
  mkapp:      e1 : S  e2 : S
  lam->body:  e : S
  lam->var:   e : S
  mklam:      n : S  e : S
  islam?:     e : S
  isvar?:     e : S
  subst:      x : S  a : S  e : S
  red:        e : S
  lambdaint:  e : B
Specialisation points:
  Call 6 in subst to subst
  Call 8 in red to red
  Call 9 in subst to subst
  Call 5 in red to red
  Call 11 in subst to subst
  Call 3 in red to red
```

C.1.5 *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)
        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 '())
        rs1
        (if (member? (car rs1) '("*" "/"))
            (let* ((rs2 (term (cdr rs1))))
              (if (equal? rs2 (cdr rs1)) rs1 rs2))
            rs1))))

(define (factor xs)
  (if (equal? "(" (car xs))
      (let* ((rs1 (expr (cdr xs))))
        (if (and (not (equal? rs1 '()))
                 (not (equal? rs1 (cdr xs)))
                 (equal? ")" (car rs1)))
            (cdr rs1)
            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) xs))


Parameter binding times:
  atom:     xs : B
  member?:  x : B  xs : B
  factor:   xs : B
  term:     xs : B
  expr:     xs : B
  parsexp:  xs : B
Specialisation points:
None.
```

C.1.6 *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)                                 ; 1
                  revltape (cons (cdar instrs) (cdr rtape)) prog)
      (if (equal? 'Left (caar instrs))
          (if (pair? revltape)
              (turing (cdr instrs)                             ; 2
                      (cdr revltape)
                      (cons (car revltape) rtape) prog)
              (turing (cdr instrs)                             ; 3
                      '()
                      (cons 'Blank rtape) prog))
      (if (equal? 'Right (caar instrs))
          (if (pair? rtape)
              (turing (cdr instrs)                             ; 4
                      (cons (car rtape) revltape)
                      (cdr rtape) prog)
              (turing (cdr instrs)                             ; 5
                      (cons 'Blank revltape)
                      '() prog))
      (if (equal? 'Goto (caar instrs))
          (turing (lookup (cdar instrs) prog) revltape rtape prog) ; 6, 7
      (if (equal? 'IfGoto (caar instrs))
          (if (equal? (car rtape) (cadar instrs))
              (turing                                          ; 8
               (lookup (cddar instrs) prog) revltape rtape prog)   ; 9
              (turing (cdr instrs) revltape rtape prog))         ; 10
          rtape
          ))))))
      rtape
      ))

(define (lookup i instrs)
  (if (= i 1) instrs (lookup (- i 1) (cdr instrs))))


Parameter binding times:
  lookup:  i : B  instrs : B
  turing:  instrs : B  revltape : D  rtape : D  prog : B
  run:     prog : B  tapeinput : B
Specialisation points:
  Call 8 in turing to turing
  Call 6 in turing to turing
```

C.2 Algorithms

C.2.1 *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))))))


Parameter binding times:
  ack:    m : B  n : D
  goal:   m : B  n : B
Specialisation points:
  Call 3 in ack to ack
```

C.2.2 *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)))))


Parameter binding times:
  binom:  n : B  k : D
  goal:   n : B  k : B
Specialisation points:
None.
----------------------------

Parameter binding times:
  binom:  n : D  k : B
  goal:   n : B  k : B
Specialisation points:
  Call 2 in binom to binom
```

C.2.3 *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)))))

Parameter binding times:
  lgth:   x : D
  monus:  x : D  y : D
  gt:     x : D  y : D
  gcd:    x : D  y : D
  goal:   x : B  y : B
Specialisation points:
  Call 1 in gt to gt
  Call 2 in monus to monus
  Call 4 in gcd to gcd
  Call 1 in lgth to lgth
  Call 2 in gcd to gcd
```

C.2.4 *gcd-2*

```
;;; Greatest common divisor, numbers represented by list length
;;; x and y are swapped in the recursive call
(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)))))


Parameter binding times:
  lgth:   x : D
  monus:  x : D  y : D
  gt:     x : D  y : D
  gcd:    x : D  y : D
  goal:   x : B  y : B
Specialisation points:
  Call 2 in monus to monus
  Call 1 in lgth to lgth
  Call 1 in gt to gt
  Call 4 in gcd to gcd
  Call 2 in gcd to gcd
```

C.2.5 *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))


Parameter binding times:
  revapp:             xs : D  rest : D
  reverse:            xs : D
  colorrest-thetrick: cs1 : D  cs : D  ncs : D  colorednodes : D  rest : B
  colorrest:          cs : D  ncs : D  colorednodes : D  rest : B
  colorof:            node : B  colorednodes : D
  possible:           color : D  adjs : B  colorednodes : D
  colornode:          cs : D  node : B  colorednodes : D
  graphcolour:        g : B  cs : D
Specialisation points:
  Call 1 in colorof to colorof
  Call 2 in colorrest-thetrick to colorrest-thetrick
  Call 2 in colornode to colornode
  Call 1 in revapp to revapp
  Call 3 in colorrest to colorrest-thetrick
----------------------------

Parameter binding times:
  revapp:             xs : D  rest : D
  reverse:            xs : D
  colorrest-thetrick: cs1 : B  cs : B  ncs : D  colorednodes : D  rest : D
  colorrest:          cs : B  ncs : B  colorednodes : D  rest : D
  colorof:            node : D  colorednodes : D
  possible:           color : B  adjs : D  colorednodes : D
  colornode:          cs : B  node : D  colorednodes : D
  graphcolour:        g : D  cs : B
Specialisation points:
  Call 1 in colorof to colorof
  Call 2 in colorrest to colorrest
  Call 2 in possible to possible
  Call 1 in revapp to revapp
  Call 3 in colorrest to colorrest-thetrick
```

C.2.6 *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 some 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))


Parameter binding times:
  revapp:              xs : D  rest : D
  reverse:             xs : D
  colorrest-thetrick:  cs1 : D  cs : D  ncs : D  colorednodes : D  rest : B
  colornoderest:       cs : D  ncs : D  node : B  colorednodes : D  rest : B
  colorrest:           cs : D  ncs : D  colorednodes : D  rest : B
  colorof:             node : B  colorednodes : D
  possible:            color : D  adjs : B  colorednodes : D
  colornode:           cs : D  node : B  colorednodes : B
  graphcolour:         g : B  cs : D
Specialisation points:
  Call 4 in colornoderest to colornoderest
  Call 1 in colorof to colorof
  Call 2 in colorrest-thetrick to colorrest-thetrick
  Call 2 in colornode to colornode
  Call 1 in revapp to revapp
  Call 1 in colorrest to colornoderest
----------------------------
Parameter binding times:
  revapp:              xs : D  rest : D
  reverse:             xs : D
  colorrest-thetrick:  cs1 : B  cs : B  ncs : B  colorednodes : D  rest : D
  colornoderest:       cs : B  ncs : B  node : D  colorednodes : D  rest : D
  colorrest:           cs : B  ncs : B  colorednodes : D  rest : D
  colorof:             node : D  colorednodes : D
  possible:            color : B  adjs : D  colorednodes : D
  colornode:           cs : B  node : D  colorednodes : B
  graphcolour:         g : D  cs : B
Specialisation points:
  Call 1 in colorof to colorof
  Call 2 in possible to possible
  Call 1 in revapp to revapp
  Call 1 in colorrest to colornoderest
```

C.2.7 *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 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 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))


Parameter binding times:
  revapp:         xs : D  rest : D
  reverse:        xs : D
  colornoderest:  cs : D  ncs : D  node : B  colorednodes : D  rest : B
  colorrest:      cs : D  ncs : D  colorednodes : D  rest : B
  colorof:        node : B  colorednodes : D
  possible:       color : D  adjs : B  colorednodes : D
  colornode:      cs : D  node : B  colorednodes : B
  graphcolour:    g : B  cs : D
Specialisation points:
  Call 1 in colorof to colorof
  Call 4 in colornoderest to colornoderest
  Call 2 in colornode to colornode
  Call 1 in revapp to revapp
  Call 1 in colorrest to colornoderest
----------------------------

Parameter binding times:
  revapp:         xs : D  rest : D
  reverse:        xs : D
  colornoderest:  cs : B  ncs : B  node : D  colorednodes : D  rest : D
  colorrest:      cs : B  ncs : B  colorednodes : D  rest : D
  colorof:        node : D  colorednodes : D
  possible:       color : B  adjs : D  colorednodes : D
  colornode:      cs : B  node : D  colorednodes : B
  graphcolour:    g : D  cs : B
Specialisation points:
  Call 1 in colorof to colorof
  Call 2 in possible to possible
  Call 1 in revapp to revapp
  Call 1 in colorrest to colornoderest
```

C.2.8 *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))))))
```

```
Parameter binding times:
  loop:   p : B  s : D  pp : B  ss : D
  match:  p : B  s : D
Specialisation points:
  Call 2 in loop to loop
----------------------------

Parameter binding times:
  loop:   p : D  s : B  pp : D  ss : B
  match:  p : D  s : B
Specialisation points:
None.
```

C.2.9 *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)))))
```

```
Parameter binding times:
  add:    x : B  y : D
  mult:   x : B  y : D
  power:  x : B  n : D
  goal:   x : B  n : B
Specialisation points:
  Call 2 in mult to mult
  Call 1 in add to add
  Call 2 in power to power
----------------------------

Parameter binding times:
  add:    x : D  y : D
  mult:   x : D  y : D
  power:  x : D  n : B
  goal:   x : B  n : B
Specialisation points:
  Call 1 in add to add
  Call 2 in mult to mult
```

C.2.10 *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)))))


Parameter binding times:
  member?:  x : D  xs : B
  via:      u : B  v : D  rest : B  edges : B
  reach:    u : B  v : D  edges : B
  goal:     u : B  v : B  edges : B
Specialisation points:
  Call 2 in reach to via
----------------------------

Parameter binding times:
  member?:  x : D  xs : B
  via:      u : D  v : B  rest : B  edges : B
  reach:    u : D  v : B  edges : B
  goal:     u : B  v : B  edges : B
Specialisation points:
  Call 2 in reach to via
----------------------------

Parameter binding times:
  member?:  x : D  xs : B
  via:      u : D  v : D  rest : B  edges : B
  reach:    u : D  v : D  edges : B
  goal:     u : B  v : B  edges : B
Specialisation points:
  Call 2 in reach to via
```

C.2.11 *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!
```

```
Parameter binding times:
  domatch-seq:    cs : D  rest : D  pats : B
  domatch-star:   cs : D  pat : B  init : D
  domatch-char:   cs : D  c : B
  domatch-dot:    cs : D
  domatch:        pat : B  cs : D
  parsep-char:    patchars : B  seq : S  stack : S
  parsep-closeb:  patchars : B  seq : S  stack : S
  parsep-openb:   patchars : B  seq : S  stack : S
  parsep-star:    patchars : B  seq : S  stack : S
  parsep-dot:     patchars : B  seq : S  stack : S
  parsep:         patchars : B  seq : S  stack : S
  parsepat:       patstr : B
  rematch:        patstr : B  str : D
Specialisation points:
  Call 2 in domatch-star to domatch-star
  Call 3 in domatch-seq to domatch-seq
  Call 6 in parsep to parsep-char
----------------------------

Parameter binding times:
  domatch-seq:    cs : D  rest : D  pats : D
  domatch-star:   cs : D  pat : D  init : D
  domatch-char:   cs : D  c : D
  domatch-dot:    cs : D
  domatch:        pat : D  cs : D
  parsep-char:    patchars : D  seq : D  stack : D
  parsep-closeb:  patchars : D  seq : D  stack : D
  parsep-openb:   patchars : D  seq : D  stack : D
  parsep-star:    patchars : D  seq : D  stack : D
  parsep-dot:     patchars : D  seq : D  stack : D
  parsep:         patchars : D  seq : D  stack : D
  parsepat:       patstr : D
  rematch:        patstr : D  str : B
Specialisation points:
  Call 2 in domatch-star to domatch-star
  Call 2 in domatch-seq to domatch-seq
  Call 1 in parsep to parsep-dot
  Call 3 in domatch-seq to domatch-seq
  Call 3 in domatch to domatch-star
  Call 2 in parsep to parsep-star
  Call 2 in parsep-char to parsep
  Call 1 in domatch-seq to domatch
  Call 3 in parsep to parsep-openb
  Call 4 in parsep to parsep-closeb
  Call 1 in parsep-char to parsep
```

C.2.12 *strmatch*

```
;;; Naive pattern string matcher
;;; strmatch returns a list of indices indicating the
;;; positions in str where patstr occurs
(define (strmatch patstr str)
  (domatch (string->list patstr) (string->list str) 0))

(define (domatch patcs cs n)
  (if (pair? cs)
      (if (prefix patcs cs)                          ; 1
          (cons n (domatch patcs (cdr cs) (+ n 1))) ; 2
          (domatch patcs (cdr cs) (+ n 1)))         ; 3
      (if (equal? patcs cs) (cons n '()) '())))
```

```
(define (prefix precs cs)
  (or (null? precs)
      (and (pair? cs) (and (equal? (car precs) (car cs))
                           (prefix (cdr precs) (cdr cs))))))
```

```
Parameter binding times:
  prefix:    precs : B  cs : D
  domatch:   patcs : B  cs : D  n : S
  strmatch:  patstr : B  str : D
Specialisation points:
  Call 3 in domatch to domatch
  Call 2 in domatch to domatch
---------------------------

Parameter binding times:
  prefix:    precs : D  cs : B
  domatch:   patcs : D  cs : B  n : B
  strmatch:  patstr : D  str : B
Specialisation points:
None.
```

C.2.13 *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)))))


Parameter binding times:
  etype:      venv : B  tenv : B  e : B  t : B
  unify:      venv : B  t1 : B  t2 : B
  subst:      venv : B  a : B  t : B
  tsubst:     a : B  t : B  t1 : B
  vtype:      venv : B  x : B
  freshtvar:  tenv : B
  typeinf:    inittenv : B  e : B
Specialisation points:
None.
```

C.3 Basic operations

C.3.1 *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))))


Parameter binding times:
  add:    x : S  y : D
  goal:   x : B  y : B
Specialisation points:
  Call 1 in add to add
----------------------------

Parameter binding times:
  add:    x : D  y : B
  goal:   x : B  y : B
Specialisation points:
None.
```

C.3.2 *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)))
      '()))


Parameter binding times:
  addlist:  xs : B  ys : D
  goal:     xs : B  ys : D
Specialisation points:
None.
```

C.3.3 *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))))
```

```
Parameter binding times:
  anchored:  x : B  y : D
  goal:      x : B  y : B
Specialisation points:
None.
----------------------------

Parameter binding times:
  anchored:  x : D  y : S
  goal:      x : B  y : B
Specialisation points:
  Call 1 in anchored to anchored
```

C.3.4 *append*

```
(define (goal x y) (append x y))
(define (append xs ys)
  (if (equal? xs '()) ys (cons (car xs) (append (cdr xs) ys))))


Parameter binding times:
  append:  xs : B  ys : D
  goal:    x : B  y : D
Specialisation points:
None.
```

C.3.5 *assocrw*

```
;;; 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 (assocrw 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))


Parameter binding times:
  rewrite:  exp : S
  assocrw:  exp : B
Specialisation points:
  Call 4 in rewrite to rewrite
  Call 3 in rewrite to rewrite
  Call 5 in rewrite to rewrite
  Call 2 in rewrite to rewrite
  Call 6 in rewrite to rewrite
  Call 1 in rewrite to rewrite
```

C.3.6 *badd*

```
(define (goal x y) (badd x y))
(define (badd x y)
  (if (equal? y '()) x (badd '(1) (badd x (cdr y)))))
```

```
Parameter binding times:
  badd:   x : B  y : D
  goal:   x : B  y : D
Specialisation points:
  Call 1 in badd to badd
  Call 2 in badd to badd
```

C.3.7 *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))


Parameter binding times:
  dec:          n : B
  h:            r : B  s : B
  g:            u : B  v : B  w : D
  f:            x : B  y : B  z : B  d : D
  contrived-1:  a : B  b : D
Specialisation points:
None.
```

C.3.8 *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))


Parameter binding times:
  dec:          n : B
  h:            r : B  s : B
  g:            u : B  v : B  w : D
  f:            x : B  y : B  z : B  d : D
  contrived-2:  a : B  b : D
Specialisation points:
  Call 1 in h to h
```

C.3.9 *decrease*

```
(define (goal x) (decrease x))
(define (decrease x) (if (equal? x '()) 42 (decrease (cdr x))))


Parameter binding times:
  decrease:  x : B
  goal:      x : B
Specialisation points:
None.
```

C.3.10 *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))


Parameter binding times:
  revconsapp:  xs : B  r : B
  deeprevapp:  xs : B  rest : B
  deeprev:     x : B
  goal:        x : B
Specialisation points:
None.
```

C.3.11 *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)))


Parameter binding times:
  bool?:     p : B
  conj?:     p : B
  disj?:     p : B
  disjconj:  p : B
Specialisation points:
None.
```

C.3.12 *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))))))

Parameter binding times:
  duplicate:  xs : B
  goal:       x : B
Specialisation points:
None.
```

C.3.13 *equal*

```
(define (goal x) (equal x))
(define (equal x) (if (equal? x '()) 42 (equal x)))
```

```
Parameter binding times:
  equal:  x : B
  goal:   x : B
Specialisation points:
  Call 1 in equal to equal
```

C.3.14 *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))
```

```
Parameter binding times:
  odd?:     x : B
  even?:    x : B
  evenodd:  x : B
Specialisation points:
None.
```

C.3.15 *exponential*

```
;;; These functions produce n*2^n size-change graphs

(define (f1 x1 y1)
  (if x1
      (f1 y1 x1)
      (f1 x1 y1)))

(define (f2 x1 y1 x2 y2)
  (if x1
      (f2 y1 x1 x2 y2)
      (f2 x2 y2 x1 y1)))

(define (f3 x1 y1 x2 y2 x3 y3)
  (if x1
      (f3 y1 x1 x2 y2 x3 y3)
      (f3 x3 y3 x1 y1 x2 y2)))

(define (f4 x1 y1 x2 y2 x3 y3 x4 y4)
  (if x1
      (f4 y1 x1 x2 y2 x3 y3 x4 y4)
      (f4 x4 y4 x1 y1 x2 y2 x3 y3)))

(define (f5 x1 y1 x2 y2 x3 y3 x4 y4 x5 y5)
  (if x1
      (f5 y1 x1 x2 y2 x3 y3 x4 y4 x5 y5)
      (f5 x5 y5 x1 y1 x2 y2 x3 y3 x4 y4)))
```

```
Parameter binding times:
  f5:  x1 : B  y1 : B  x2 : B  y2 : B  x3 : B  y3 : B  x4 : B  y4 : B  x5 : B  y5 : B
  f4:  x1 : B  y1 : B  x2 : B  y2 : B  x3 : B  y3 : B  x4 : B  y4 : B
  f3:  x1 : B  y1 : B  x2 : B  y2 : B  x3 : B  y3 : B
  f2:  x1 : B  y1 : B  x2 : B  y2 : B
  f1:  x1 : B  y1 : B
Specialisation points:
  Call 2 in f1 to f1
  Call 1 in f3 to f3
  Call 1 in f4 to f4
  Call 1 in f2 to f2
  Call 1 in f5 to f5
  Call 1 in f1 to f1
  Call 2 in f2 to f2
  Call 2 in f3 to f3
  Call 2 in f4 to f4
  Call 2 in f5 to f5
```

C.3.16 *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))


Parameter binding times:
  op:     x1 : D  x2 : D
  foldr:  a : D  xs : B
  foldl:  a : D  xs : B
  fold:   a : D  xs : B
Specialisation points:
None.
----------------------------
Parameter binding times:
  op:     x1 : D  x2 : D
  foldr:  a : B  xs : D
  foldl:  a : D  xs : D
  fold:   a : B  xs : D
Specialisation points:
  Call 2 in foldr to foldr
  Call 1 in foldl to foldl
```

C.3.17 *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))))

Parameter binding times:
  game:   p1 : B  p2 : B  moves : B
  goal:   p1 : B  p2 : B  moves : B
Specialisation points:
None.
```

C.3.18 *increase*

```
(define (goal x) (increase x))
(define (increase x) (if (equal? x '()) 42 (increase (cons 1 x))))


Parameter binding times:
  increase:  x : S
  goal:      x : B
Specialisation points:
  Call 1 in increase to increase
```

C.3.19 *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))))


Parameter binding times:
  lookup:     fnum : D  p : B
  intlookup:  e : D  p : B
  run:        e : D  p : B
Specialisation points:
  Call 1 in intlookup to intlookup
```

C.3.20 *letexp*

```
;; Testing the let construction
(define (goal x y) (letexp x y))
(define (letexp x y) (let* ((z (cons 1 x))) (letexp z y)))


Parameter binding times:
  letexp:  x : S  y : B
  goal:    x : B  y : B
Specialisation points:
  Call 1 in letexp to letexp
```

C.3.21 *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)))

Parameter binding times:
  list?:  xs : B
  goal:   x : B
Specialisation points:
None.
```

C.3.22 *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))))))


Parameter binding times:
  even?:  x : B
  lte?:   x : B  y : D
  goal:   x : B  y : D
Specialisation points:
None.
----------------------------

Parameter binding times:
  even?:  x : D
  lte?:   x : D  y : B
  goal:   x : D  y : B
Specialisation points:
  Call 1 in even? to even?
```

C.3.23 *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))

Parameter binding times:
  f:      x : B
  map:    xs : B
  goal:   xs : B
Specialisation points:
None.
```

C.3.24 *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))


Parameter binding times:
  member?:  x : D  xs : B
  goal:     x : B  xs : B
Specialisation points:
None.
----------------------------
```

```
Parameter binding times:
  member?:  x : B  xs : D
  goal:     x : B  xs : B
Specialisation points:
  Call 1 in member? to member?
```

C.3.25 *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)))))))

Parameter binding times:
  merge:  xs : B  ys : D
  goal:   xs : B  ys : D
Specialisation points:
  Call 2 in merge to merge
```

C.3.26 *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))))


Parameter binding times:
  add:    x : D  y : D
  mul:    x : B  y : D
  goal:   x : B  y : D
Specialisation points:
  Call 1 in add to add
```

C.3.27 *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))))


Parameter binding times:
  app:       xs : B  ys : B
  naiverev:  xs : B
  goal:      xs : B
Specialisation points:
None.
```

C.3.28 *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))))


Parameter binding times:
  dec:      x : B
  nestdec:  x : B
  goal:     x : B
Specialisation points:
None.
```

C.3.29 *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)))


Parameter binding times:
  eql:      x : B
  nesteql:  x : B
  goal:     x : B
Specialisation points:
  Call 1 in eql to eql
  Call 1 in nesteql to nesteql
```

C.3.30 *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)))))


Parameter binding times:
  immatcopy:  x : S
  nestimeql:  x : S
  goal:       x : B
Specialisation points:
  Call 1 in immatcopy to immatcopy
  Call 1 in nestimeql to nestimeql
```

C.3.31 *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)))))


Parameter binding times:
  inc:      x : S
  nestinc:  x : S
  goal:     x : B
Specialisation points:
  Call 1 in inc to inc
  Call 1 in nestinc to nestinc
```

C.3.32 *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)))))


Parameter binding times:
  nolexicord:  a1 : B  b1 : B  a2 : B  b2 : B  a3 : B  b3 : B
  goal:        a1 : B  b1 : B  a2 : B  b2 : B  a3 : B  b3 : B
Specialisation points:
None.
----------------------------

Parameter binding times:
  nolexicord:  a1 : B  b1 : B  a2 : B  b2 : B  a3 : D  b3 : D
  goal:        a1 : B  b1 : B  a2 : B  b2 : B  a3 : B  b3 : D
Specialisation points:
None.
```

C.3.33 *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))
```

```
Parameter binding times:
  ordered?:  xs : B
  goal:      xs : B
Specialisation points:
None.
```

C.3.34 *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))
```

```
Parameter binding times:
  member?:   x : B  xs : D
  overlap?:  xs : B  ys : D
  goal:      xs : B  ys : D
Specialisation points:
  Call 1 in member? to member?
```

C.3.35 *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))))
```

```
Parameter binding times:
  revapp:      xs : S  rest : S
  mapconsapp:  x : S  xss : S  rest : S
  select:      x : S  revprefix : S  postfix : S
  permute:     xs : S
  goal:        xs : B
Specialisation points:
  Call 1 in mapconsapp to mapconsapp
  Call 4 in select to select
  Call 1 in revapp to revapp
  Call 1 in permute to select
```

C.3.36 *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))))

Parameter binding times:
  revapp:  xs : B  rest : D
  goal:    x : B  y : D
Specialisation points:
None.
```

C.3.37 *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))))


Parameter binding times:
  revapp:   xs : B  rest : B
  selects:  x : B  revprefix : B  postfix : B
  select:   xs : B
Specialisation points:
None.
```

C.3.38 *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))))

Parameter binding times:
  append:   xs : S  ys : S
  reverse:  xs : S
  shuffle:  xs : S
  goal:     xs : B
Specialisation points:
  Call 2 in reverse to reverse
  Call 1 in append to append
  Call 1 in shuffle to shuffle
```

C.3.39 *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)


Parameter binding times:
  r:      x : B  y : D
  h:      x : B  y : D
  g:      x : B  y : D
  f:      x : B  y : D
  sp1:    x : B  y : D
Specialisation points:
  Call 1 in h to h
  Call 2 in h to f
```

C.3.40 *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 binding times:
  mapconsapp:  x : B  xss : B  rest : B
  subsets:     xs : B
  goal:        xs : B
Specialisation points:
None.
```

C.3.41 *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 (null? y) 42
      (f (if (null? x) x       (cdr x))       ; 1
         (if (null? x) (cdr y) (cons 1 y)))))
(define (g x y)
  (if (null? y) 42
      (if (null? x)
          (g x       (cdr y))                 ; 1
          (g (cdr x) (cons 1 y)))))           ; 2


Parameter binding times:
  g:      x : B  y : D
  f:      x : B  y : D
  goal:   x : B  y : D
Specialisation points:
  Call 1 in g to g
  Call 1 in f to f
----------------------------

Parameter binding times:
  g:      x : D  y : S
  f:      x : D  y : D
  goal:   x : D  y : B
Specialisation points:
  Call 1 in g to g
  Call 2 in g to g
  Call 1 in f to f
```

C.3.42 *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)))


Parameter binding times:
  t:      x : B  y : D
  r:      x : B  y : D
  p:      x : B  y : D
  q:      x : B  y : D
  e:      a : B  b : D
  goal:   x : B  y : D
Specialisation points:
  Call 3 in p to p
  Call 3 in q to q
  Call 3 in r to r
  Call 2 in p to r
```

C.4 Sorting functions

C.4.1 *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))


Parameter binding times:
  merge:       xs1 : S  xs2 : S
  splitmerge:  xs : S  xs1 : S  xs2 : S
  mergesort:   xs : S
  goal:        xs : B
Specialisation points:
  Call 2 in merge to merge
  Call 1 in merge to merge
  Call 1 in splitmerge to splitmerge
  Call 1 in mergesort to splitmerge
```

C.4.2 *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))))
      '()))

Parameter binding times:
  remove:   x : S  xs : S
  appmin:   min : S  rest : S  xs : S
  minsort:  xs : S
  goal:     xs : B
Specialisation points:
  Call 2 in appmin to appmin
  Call 1 in appmin to appmin
  Call 1 in remove to remove
  Call 1 in minsort to appmin
```

C.4.3 *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))

Parameter binding times:
  app:        xs : S  ys : S
  part:       x : S  xs : S  xs1 : S  xs2 : S
  quicksort:  xs : S
  goal:       xs : B
Specialisation points:
  Call 3 in part to part
  Call 1 in app to app
  Call 1 in part to part
  Call 2 in part to part
  Call 1 in quicksort to part
```