# *Invited paper:* Program Generation, Termination, and Binding-time Analysis

Neil D. Jones and Arne J. Glenstrup, DIKU, University of Copenhagen
e-mail: neil@diku.dk, panic@diku.dk

**Abstract.** 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.
We first discuss the relations between problem specifications and their solutions in program form, and then narrow the discussion to an important special case: program transformation. Although the goal of fully automatic program generation is still far from fully achieved, there has been some success in a special case: partial evaluation, also known as program specialization.
A key problem in all program generation is *termination* of the generation process. This paper 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 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.

## 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 [15, 41, 51], domain-specific languages (at least, those compiled to an implementation language) [37, 55], program transformation [8, 11, 25], and both practical and theoretical work aimed at generating programs from specifications [37, 43].

Recent years have seen a rapid growth of interest in generative, automatic approaches to program management, updating, adaptation, transformation, and evolution, e.g., [43, 52, 70]. 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 [18], 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 [62] 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 [2, 15, 27, 41, 35, 69, 71, 72]).
- Don't solve *the same sub-problem* repeatedly. A successful practical example is the XSB logic programming system [59], based on *memoization*: Instead of recomputing results, store them for future retrieval when first computed.
- Avoid *multiple passes* over same data structure (tupling, deforestation [8, 11, 33, 76]).
- Use an abstract, *high-level specification language*. SETL [9] 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 [56], 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 [8].
3. Tools with automated strategies for applying transformation operations. Human interaction to check whether the transformation is "on target." Examples: Chin, Khoo, Liu [11, 50].
4. Hand-placed program annotations to guide a transformation tool. Examples: Chambers' and Engeler's DyC and Tick C [34, 57]. 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 [47, 65].
6. Computer-placed program annotations to guide transformation, e.g., binding-time annotation [6, 14, 15, 31, 32, 41, 35, 73]. 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.

## 2 Partial evaluation and program generation

Partial evaluation [15, 20, 41, 35, 51] 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 ([4, 32], etc.);
- Extending functionality of existing languages by adding binding-time options ([34], etc.);
- Functional languages ([6, 14, 38, 44, 74], etc.);
- Logic programming languages ([25, 26, 48, 51], etc.);
- Compiling or other transformation by specializing interpreters ([5, 6, 28, 41, 40, 53], etc.);
- Optimization of operating systems, e.g., remote procedure calls, device drivers, etc. ([16, 52, 57], 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 **??**.)

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]

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

```
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 [15, 41, 35] 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 $\mathtt{p}_{\mathtt{in1}}$ 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.

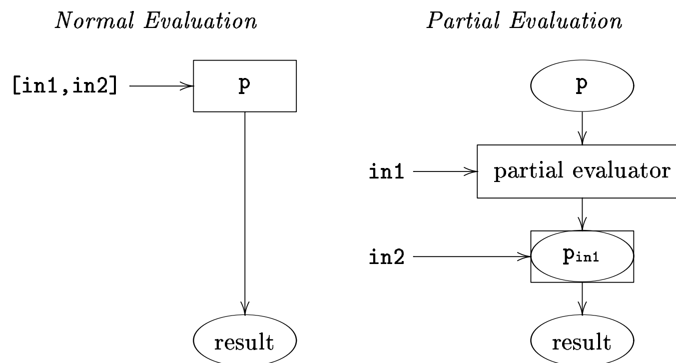



**Fig. 1.** Comparison of normal and partial evaluation. Boxes represent programs, ovals represent data objects. The *residual program* is $\mathtt{p}_{\mathtt{in1}}$.

**Example 1: the standard toy example** for partial evaluation is the program power, computing $\mathtt{x}^{\mathtt{n}}$ 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 $\mathtt{n} = 13$ yields a residual program that runs several times faster than the general one above:

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

**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* | *Specialized program (for* m = 2*)* |
|---|---|
| <pre>ack(m,n) = if m=0 then n+1 else<br>           if n=0 then<br>             ack(m-1,1) else<br>             ack(m-1,ack(m,n-1))</pre> | <pre>ack_2(n) = if n=0 then<br>                ack_1(1) else<br>                ack_1(ack_2(n-1))<br>ack_1(n) = if n=0 then<br>                ack_0(1) else<br>                ack_0(ack_1(n-1))<br>ack_0(n) = n+1</pre> |

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

**Definition 1.** *A partial evaluator* spec *is a program such that for all programs* p *and input data* in1, in2*:*

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

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

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

Program $\mathtt{p_{in1}}$ 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 $\mathtt{out:=}[\![\mathtt{p}]\!]\,\mathtt{[in1,in2]}$ is often slower than the run $\mathtt{out:=}[\![\mathtt{p_{in1}}]\!]\,\mathtt{[in2]}$, as seen in the two examples above. Said another way: It is slower to run the *general* program p on [in1,in2] than it is to run the *specialized* residual program $\mathtt{p_{in1}}$ on in2.

The reason is that, while constructing $\mathtt{p_{in1}}$, 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 $[\![\mathtt{p_{in1}}]\!]\,\mathtt{[in2]}$; but they will be performed *every time* $[\![\mathtt{p}]\!]\,\mathtt{[in1,in2]}$ is run. This is especially relevant if p is run often, with in1 changing less frequently than in2.

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

*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 $\mathtt{p_{in1}}$ 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 [21, 22, 29, 31, 64]. 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 [23, 24] are called the *Futamura projections:*

**Generic Futamura projections**

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

*Consequences:*

A. $[\![\mathtt{p}]\!]\,\mathtt{[in1,in2]} = [\![\mathtt{p_{in1}}]\!]\,\mathtt{[in2]}$

B. $\mathtt{p_{in1}} = [\![\mathtt{p\text{-}gen}]\!]\,\mathtt{[in1]}$

C. $\mathtt{p\text{-}gen} = [\![\mathtt{cogen}]\!]\,\mathtt{[p]}$

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 1. Proof of Consequence B is by simple algebra:

$$
\begin{aligned}
[\![\texttt{p-gen}]\!]\,[\texttt{in1}] &= [\![\,[\![\texttt{spec}]\!]\,[\texttt{spec,p}]\,]\!]\,[\texttt{in1}] && \text{By definition of } \texttt{p-gen}\\
&= [\![\texttt{spec}]\!]\,[\texttt{p,in1}] && \text{By Definition 1}\\
&= \texttt{p}_{\texttt{in1}} && \text{By definition of } \texttt{p}_{\texttt{in1}}
\end{aligned}
$$

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. $\texttt{p}_{\texttt{in1}}$ := [[spec]] [p,in1]
II. $\texttt{p}_{\texttt{in1}}$ := [[p-gen]] [in1]

Way I is in practice usually several times slower than Way II. This is for exactly the same reason that $\texttt{p}_{\texttt{in1}}$ 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 [28], Tempo [16] and PGG [74] are `cogen` programs written directly, rather than built using self-application of `spec` as described above and in book [41]. (Descriptions of the idea can be found in [5, 36, 73].) 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 Fromt-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.** *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` := $[\![$`spec`$]\!]$ `[interp,source]`
2. `compiler` := $[\![$`spec`$]\!]$ `[spec,interp]`
3. `cogen` := $[\![$`spec`$]\!]$ `[spec,spec]`

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

A. $[\![$`interp`$]\!]$ `[source,in]` = $[\![$`target`$]\!]$ `[in]`
B. `target` = $[\![$`compiler`$]\!]$ `[source]`
C. `compiler` = $[\![$`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` = $[\![$`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.")

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

[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 $[\![$`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`.

```
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]

---

[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 [41, 71].

### 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* (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 $\mathtt{m} = 2, 1, 0$, respectively.

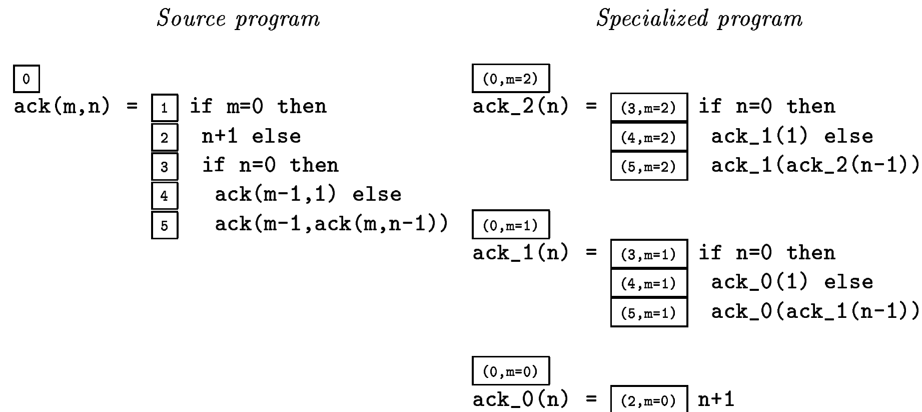



**Fig. 3.** Partial evaluation of Ackermann's function for static m=2

**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 $\texttt{exp}^{resid}$ can be computed roughly as follows:

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

1. Reduce any subexpressions $\texttt{e}_1, \ldots, \texttt{e}_n$ of exp to $\texttt{e}_1^{resid}, \ldots, \texttt{e}_n^{resid}$.
2. If exp = "$\texttt{basefn(e}_1, \ldots, \texttt{e}_n\texttt{)}$" and some $\texttt{e}_i^{resid}$ is not fully known, then
   $\texttt{exp}^{resid}$ = the **residual expression** "$\texttt{basefn(e}_1^{resid}, \ldots, \texttt{e}_n^{resid}\texttt{)}$", else
   $\texttt{exp}^{resid}$ = the **value** $[\![\texttt{basefn}]\!]\texttt{(e}_1^{resid}, \ldots, \texttt{e}_n^{resid}\texttt{)}$.
3. If exp = "if $\texttt{e}_0$ then $\texttt{e}_1$ else $\texttt{e}_2$" and $\texttt{e}_0^{resid}$ is not fully known, then
   $\texttt{exp}^{resid}$ = the residual expression "if $\texttt{e}_0^{resid}$then $\texttt{e}_1^{resid}$else $\texttt{e}_2^{resid}$", 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 exp = "$\texttt{f(e}_1, \ldots, \texttt{e}_n\texttt{)}$" can either be
   **residualized:** $\texttt{exp}^{resid} = \texttt{f(e}_{i_1}, \ldots, \texttt{e}_{i_m}\texttt{)}$ with some known $\texttt{e}_{i_j}$ omitted[6],
   or **unfolded:** $\texttt{exp}^{resid}$ = *Reduce*(definition of f, *vs*′)
   where *vs*′ is the static knowledge about $\texttt{e}_1^{resid}, \ldots, \texttt{e}_n^{resid}$.

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)

[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 [33, 48, 65, 75].

[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.

#### 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 2`x` 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 not to unfold the second `dblplus` call. The effect is to *generalize* `v`, i.e., to make it dynamic.

**Definition 3.** *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 [4, 26, 47, 48, 65, 75].

*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 [6, 12, 14, 16, 21, 28, 32, 41, 35, 53].

*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 [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 [4], since its value is not known at call site [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 take 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* [37, 43, 55] 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

[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 [12, 14].

executed at a later time. This line of thinking has been developed much further, e.g., [34] for C and, for typed functional languages, in early work on Meta-ML by Sheard and others [54, 63, 72], and subsequent work by Taha and other researchers papers [68, 69, 71]. Goals of the work include efficiency improvement by staging computations to 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 [27].

*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 $\mathtt{p_{int}}$ is an explicit, printable program. A program in a multi-level language such as Meta-ML can construct program bits as data values and then run them, but it does not produce stand-alone, separately executable programs.

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 Brown, Blonde and Black.

### 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 [6, 7, 14, 45] and Prolog [60], and there has been some success on C: the C-mix [28] and Tempo systems [16]. Further, Schultz has succeeded in doing Java specialization by translating to C, using the Tempo system, and the translating back [61].

There have been a number of practically useful applications of partial evaluation, many exploiting its ability to build program generators [32, 40, 52, 45, 67].

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 the space of all possible run-time states. Such analysis can be done by abstract interpretation [19, 42] 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 [64], and works by Das and Glenstrup [21, 22, 29, 31]. Although the generation of program generators is a significant accomplishment with promise for wider applications, program generator generation places even higher 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 to guarantee 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 [49], term rewriting [3], functional programming [1, 46, 77] and partial evaluation [21, 22, 29, 31, 64].

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 [46].

### 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," $G_c$, for each call $c$ : f $\rightarrow$ 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 for nested function calls, see [10, 39, 41]).

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:

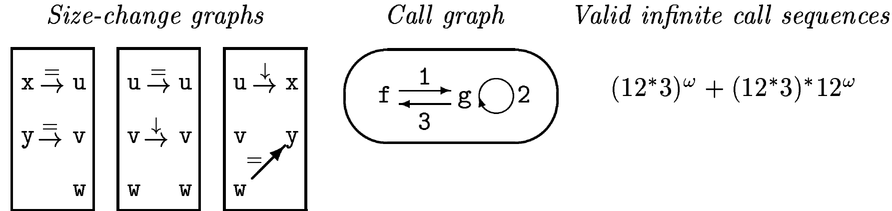
> A program terminates on all inputs if *every valid infinite call sequence* would, if executed, cause an infinite decrease in some parameter values.

**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 '=' (the change does not increase the value) or $\downarrow$ (the change decreases the value). The valid infinite call sequences can be seen from the call graph, and are indicated by $\omega$-regular expressions[9].

*Size-change graphs* *Call graph* *Valid infinite call sequences*




$G_1$ : f $\rightarrow$ g  $G_2$ : g $\rightarrow$ g  $G_3$ : g $\rightarrow$ f

[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 $\omega$ indicates an infinite number of repetitions.

*Informal argument for termination:* Examination of the size-change graphs reveals that a call sequence ending in $12^\omega$ 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. Consequently *no valid infinite call sequence is possible*, so the program terminates.

### 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 $G : \mathtt{f} \to \mathtt{g}$ and $G' : \mathtt{g} \to \mathtt{h}$ may be composed in an obvious way to yield a size-change graph $G; G' : \mathtt{f} \to \mathtt{h}$. 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 [46]:

**Theorem 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* $G : \mathtt{f} \to \mathtt{g}$ *and* $G' : \mathtt{g} \to \mathtt{h}$. *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* $\mathtt{z} \stackrel{\downarrow}{\to} \mathtt{z}$.

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.

$$G_{13} = \begin{array}{|ccc|} \hline \mathtt{x} & \stackrel{\downarrow}{\to} & \mathtt{x} \\ \mathtt{y} & & \mathtt{y} \\ \hline \end{array} \qquad\qquad G_{231} = \begin{array}{|ccc|} \hline \mathtt{u} & \stackrel{\downarrow}{\to} & \mathtt{u} \\ \mathtt{v} & & \mathtt{v} \\ \mathtt{w} & & \mathtt{w} \\ \hline \end{array}$$

### 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. We conjecture that programs computing all and only Péter's "multiple recursive" functions [56] can be proven terminating by this method. Thus, a great many computations can be treated.

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

*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 [29, 31]. The size-change termination principle arose while trying to explain our methods simply, by starting with one-stage computations as an easier special case [46]. 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 rather more complex graphs to analyze termination of logic programs [49], based on the same mathematical properties as our method. A third discovery: the Sistla-Vardi-Wolper algorithm for determinization of Büchi automata [58] 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 [49] can be run from a web page; and the Mercury termination analysis [66] has been applied to all the programs comprising the Mercury compiler. In functional programming, analyses [1] and [46] have both been implemented as part of the AGDA proof assistant [17]. Further, [77] uses dependent types for higher-order programs, not for termination analysis but to verify run-time bounds specified by the programmer with the aid of types. A recent paper on termination analysis of imperative programs is [13].

## 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. More details can be found in [30]. The results described here are significantly better than those of Chapter 14 in [41] and [29, 31]. Although the line of reasoning is similar, the following results are stronger and more precise, and 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" [65]. This test, carried out during specialization, seems expensive but strong. It is evaluated on a variety of test examples in [47].

### 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, as long as residual programs have no calls. The idea is to modify the size-change termination principle as follows (recall Section 2.3.3, Definition 3):

> 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 [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 [4], where it is *increased.*

A typical *online* specializer might see at call site [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 [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 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." This effect 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.

### 4.4 Bounded static variation

**Definition 4.** *Program* `p` *is* quasiterminating *iff for any input vector* $\overline{in}$*, the following is a* finite set*:*

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

Naturally, a terminating program is also quasiterminating.

A *necessary condition for termination of offline specialization* is that *the program is quasiterminating in its static parameters.* We define this more formally:

**Definition 5.** *Let program* `p` *contain definition* $\mathtt{f(x_1,\ldots,x_n)}$ `= 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:*

[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:

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

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

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

$$\mathit{StatVar}(\mathtt{u}, x) = \{x, x-1, \ldots, 1, 0\}$$
$$\mathit{StatVar}(\mathtt{v}, x) = \{0, 2, \ldots, 2x\}$$

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

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

(Here $\varepsilon$ is the empty list of static program inputs.) Parameter `u` must be dynamic because it depends on `x` at call site [1]. Further, BTA *must not* annotate `v` as static because 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 the size-change approach is based, *may-increase* properties as well. It is not difficult to devise abstract interpretation or constraint solving analyses to detect either modality of size-change behavior, see [41] Section 14.3, or [10, 39].

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:

$$G_1 = \begin{cases} G_1^{\uparrow}: & \begin{array}{l} \mathtt{x} \xrightarrow{=} \mathtt{u} \\ \qquad\ \mathtt{v} \end{array} \\ G_1^{\downarrow}: & \begin{array}{l} \mathtt{x} \xrightarrow{=} \mathtt{u} \\ \qquad\ \mathtt{v} \end{array} \end{cases} \qquad G_2 = \begin{cases} G_2^{\uparrow}: & \begin{array}{l} \mathtt{u} \xrightarrow{\downarrow} \mathtt{u} \\ \mathtt{v} \xrightarrow{\uparrow} \mathtt{v} \end{array} & \text{"May-increase"} \\ G_2^{\downarrow}: & \begin{array}{l} \mathtt{u} \xrightarrow{\downarrow} \mathtt{u} \\ \mathtt{v} \qquad \mathtt{v} \end{array} & \text{"Must-decrease"} \end{cases}$$

### 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* $\beta$ : *ParameterNames*[12] $\rightarrow \{S, D, U\}$ that classifies every function parameter as "static," "dynamic," or "as yet undecided." The desired division should have no $U$ 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 $\beta'(\mathtt{x}) \neq U$ implies $\beta(\mathtt{x}) = \beta'(\mathtt{x})$ for all parameters x.

*Constraints on a division* $\beta$*:*

1. $\beta \sqsupseteq \beta_0$ where $\beta_0$ is the *initial division*, mapping each program input parameter to its given binding time and all other parameters to $U$.
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 $\beta$ satisfy the Section 4.6 constraints and $\beta(\mathtt{x}) = U$.[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 using familiar graph algorithms. First, define $\mathtt{x} \equiv \mathtt{w}$ iff 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. Identify, as a candidate for upgrading, any x such that $\beta(\mathtt{x}) = U$, and $\beta(\mathtt{y}) = S$ whenever x depends on $\mathtt{y} \notin [\mathtt{x}]$.
C. Reclassify parameters $\mathtt{z} \in [\mathtt{x}]$ by $\beta(\mathtt{z}) := S$ if no idempotent $G \in \mathcal{S}$ contains $\mathtt{w} \stackrel{\uparrow}{\rightarrow} \mathtt{w}$ for any $\mathtt{w} \in [\mathtt{x}]$.

For double, C allows changing initial $\beta = [\mathtt{x} \mapsto S, \mathtt{u} \mapsto U, \mathtt{v} \mapsto U]$ into

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

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

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

---

[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.

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:

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

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 $\beta(\texttt{x}) = U$ and $\beta(\texttt{y}) = S$ where $\beta$ 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:

A. Compute the closure $\mathcal{S}$ of program p's size-change graphs.
B. Identify, as a candidate for upgrading, any x such that $\beta(\texttt{x}) = U$, and $\beta(\texttt{y}) = S$ whenever x depends on $\texttt{y} \notin [\texttt{x}]$.
C. Reclassify all parameters $\texttt{z} \in [\texttt{x}]$ by $\beta(\texttt{z}) := S$ if every idempotent $G \in \mathcal{S}$ containing $\texttt{w} \xrightarrow{\uparrow} \texttt{w}$ for some $\texttt{w} \in [\texttt{x}]$ *also* contains $\texttt{y} \xrightarrow{\downarrow} \texttt{y}$ for some y with $\beta(\texttt{y}) = S$.

For the double example, C allows changing $\beta = [\texttt{x} \mapsto S, \texttt{u} \mapsto S, \texttt{v} \mapsto U]$ into

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

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.

[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.

### 4.9 Specialization point insertion.

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

Infinite specialization-time unfolding can be prevented by doing no unfolding at all, but by Constraint 4 this could force other parameters to be dynamic. (For the interpreter example, it would force functions `lkbody` etc. and in turn also `e` and `ns` to be dynamic, which is definitely not desirable.)

A more liberal unfolding policy can be based upon a *specialization-point insertion* analysis to mark a limited set of call sites as specialization points, not to be unfolded – as few as possible, just enough to prevent infinite loop unrolling. This is not explained here, since the paper is already long enough. More details, and correctness proofs, can be found in [29] and the forthcoming [30].

## 5 Conclusion and directions for future work

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

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

**Termination analysis:**

- Apply termination analysis to multi-stage languages, real-time systems [**?**], strictness analysis, automatic theorem proving (type theory).
- 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#.
- Find annotations to ensure preservation of effects.

**Static program analyses:**

- Devise an *overlap* analysis to discover when a function can be called repeatedly with the same arguments. Memoization of such programs can yield superlinear speedups, but costs a high 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.

### 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

1. Andreas Abel and Thorsten Altenkirch. A semantical analysis of structural recursion. In *Abstracts of the Fourth International Workshop on Termination WST'99*, pages 24–25. unpublished, May 1999.
2. Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison Wesley, 1986.
3. Thomas Arts and Jürgen Giesl. Proving innermost termination automatically. In *Proceedings Rewriting Techniques and Applications RTA'97*, volume 1232 of *Lecture Notes in Computer Science*, pages 157–171. Springer, 1997.
4. Andrew Berlin and Daniel Weise. Compiling scientific code using partial evaluation. *IEEE Computer*, 23(12):25–37, 1990.
5. Lars Birkedal and Morten Welinder. Hand-writing program generator generators. In M. Hermenegildo and J. Penjam, editors, *Proceedings of the 6th International Symposium on Programming Language Implementation and Logic Programming (PLILP '94)*, pages 198–214. Springer-Verlag, September 1994.
6. Anders Bondorf. Automatic autoprojection of higher order recursive equations. *Science of Computer Programming*, 17:3–34, 1991.
7. Anders Bondorf and Jesper Jørgensen. Efficient analyses for realistic off-line partial evaluation: extended version. Technical Report 93/4, DIKU, University of Copenhagen, Denmark, 1993.
8. Rod M. Burstall and John Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, January 1977.
9. Jiazhen Cai, P. Facon, Fritz Henglein, Robert Paige, and Edmond Schonberg. Type analysis and data structure selection. In *Constructing Programs From Specifications*, pages 325–347. North-Holland, 1991.
10. Wei-Ngan Chin and Siau-Cheng Khoo. Calculating sized types. *Higher-Order and Symbolic Computation*, 14(2/3):261–300, 2002.
11. Wei-Ngan Chin, Siau-Cheng Khoo, and Tat-Wee Lee. Synchronisation analysis to stop tupling. In *Programming Languages and Systems (ESOP'98)*, pages 75–89, Lisbon, 1998. Springer LNCS 1381.

12. Niels H. Christensen, Robert Glück, and Søren Laursen. Binding-time analysis in partial evaluation: One size does *not* fit all. In D. Bjørner, M. Broy, and A. V. Zamulin, editors, *Perspectives of System Informatics. Proceedings*, volume 1755 of *Lecture Notes in Computer Science*, pages 80–92. Springer-Verlag, 2000.
13. Michael A. Colón and Henny B. Sipma. Practical methods for proving program termination. In *Conference on Computer-Aided Verification (CAV)*, Lecture Notes in Computer Science, pages ??–?? Springer-Verlag, 2002.
14. Charles Consel. A tour of Schism: a partial evaluation system for higher-order applicative languages. In *ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 66–77, 1993.
15. Charles Consel and Olivier Danvy. Tutorial notes on partial evaluation. In *ACM Symposium on Principles of Programming Languages*, pages 493–501. ACM Press, 1993.
16. Charles Consel and François Noël. A general approach for run-time specialization and its application to C. In *ACM Symposium on Principles of Programming Languages*, pages 145–156, 1996.
17. Catarina Coquand. The interactive theorem prover Agda. http://www.cs.chalmers.se/~catarina/agda/, 2001.
18. James Corbett, Matthew Dwyer, John Hatcliff, Corina Pasareanu, Robby, Shawn Laubach, and Hongjun Zheng. Bandera: Extracting finite-state models from Java source code. In *22nd International Conference on Software Engineering*, pages 439–448. IEEE Press, 2000.
19. Patrick Cousot and Radhia Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *4th POPL, Los Angeles, CA*, pages 238–252, Jan. 1977.
20. Olivier Danvy, Robert Glück, and Peter Thiemann, editors. *Partial Evaluation*, volume 1110 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.
21. Manuvir Das. *Partial Evaluation using Dependence Graphs*. PhD thesis, University of Wisconsin-Madison, February 1998.
22. Manuvir Das and Thomas Reps. BTA termination using CFL-reachability. Technical Report 1329, Computer Science Department, University of Wisconsin-Madison, 1996.
23. Yoshihiko Futamura. Partial evaluation of computation process – an approach to a compiler-compiler. *Higher-Order and Symbolic Computation*, 12(4):381–391, 1999. Reprinted from Systems · Computers · Controls 2(5), 1971.
24. Yoshihiko Futamura. Partial evaluation of computation process, revisited. *Higher-Order and Symbolic Computation*, 12(4):377–380, 1999.
25. John Gallagher and Maurice Bruynooghe. Some low-level source transformations for logic programs. In M. Bruynooghe, editor, *Proceedings of the Second Workshop on Meta-Programming in Logic, April 1990, Leuven, Belgium*, pages 229–246. Department of Computer Science, KU Leuven, Belgium, 1990.
26. John P. Gallagher. Tutorial on specialisation of logic programs. In *Proceedings of PEPM'93, the ACM Sigplan Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 88–98. ACM Press, 1993.
27. Steve Ganz, Amr Sabry, and Walid Taha. Macros as Multi-Stage computations: Type-Safe, generative, binding macros in MacroML. In Cindy Norris and Jr. James B. Fenwick, editors, *Proceedings of the Sith ACM SIGPLAN International Conference on Functional Programming (ICFP-01)*, volume 36, 10 of *ACM SIGPLAN notices*, pages 74–85, New York, September 3–5 2001. ACM Press.
28. Arne Glenstrup, Henning Makholm, and Jens Peter Secher. C-Mix — specialization of C programs. In Hatcliff et al. [35], pages 108–154.

29. Arne John Glenstrup. Terminator II: Stopping partial evaluation of fully recursive programs. Master's thesis, DIKU, University of Copenhagen, Universitetsparken 1, DK-2100 Copenhagen Ø, June 1999.
30. Arne John Glenstrup. Partial evaluation, termination analysis, and specialisation-point insertion. *In preparation*, 2002.
31. Arne John Glenstrup and Neil D. Jones. 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, June 1996.
32. Robert Glück, Ryo Nakashige, and Robert Zöchling. Binding-time analysis applied to mathematical algorithms. In J. Doležal and J. Fidler, editors, *System Modelling and Optimization*, pages 137–146. Chapman & Hall, 1995.
33. Robert Glück and Morten Heine Sørensen. A roadmap to metacomputation by supercompilation. In O. Danvy, R. Glück, and P. Thiemann, editors, *Partial Evaluation*, volume 1110 of *Lecture Notes in Computer Science*, pages 137–160. Springer-Verlag, 1996.
34. Brian Grant, Markus Mock, Matthai Philipose, Craig Chambers, and Susan J. Eggers. DyC: An expressive annotation-directed dynamic compiler for C. *Theoretical Computer Science*, 248(1–2):147–199, 2000.
35. John Hatcliff, Torben Mogensen, and Peter Thiemann, editors. Partial Evaluation: Practice and Theory. Proceedings of the 1998 DIKU International Summerschool, volume 1706. Springer-Verlag, 1999.
36. Carsten Kehler Holst and John Launchbury. Handwriting cogen to avoid problems with static typing. In *Draft Proceedings, Fourth Annual Glasgow Workshop on Functional Programming, Skye, Scotland*, pages 210–218. Glasgow University, 1991.
37. Paul Hudak. Building domain specific embedded languages. *ACM Computing Surveys*, 28A:(electronic), December 1996.
38. John Hughes. Type specialisation for the $\lambda$-calculus; or a new paradigm for partial evaluation based on type inference. In Danvy et al. [20], pages 183–215.
39. John Hughes, Lars Pareto, and Amr Sabry. Proving the correctness of reactive systems using sized types. In Olivier Danvy, Robert Glück, and Peter Thiemann, editors, *ACM Symposium on Principles of Programming Languages*, pages 410–423. ACM Press, 1996.
40. Neil D. Jones. What *Not* to do when writing an interpreter for specialisation. In Danvy et al. [20], pages 216–237.
41. Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice Hall. Download accessible from `www.diku.dk/users/neil`, 1993.
42. Neil D. Jones and Flemming Nielson. Abstract interpretation: a semantics-based tool for program analysis. In *Handbook of Logic in Computer Science*. Oxford University Press, 1994. 527–629.
43. Richard B. Kieburtz, Laura McKinney, Jeffrey Bell, James Hook, Alex Kotov, Jeffrey Lewis, Dino Oliva, Tim Sheard, Ira Smith, and Lisa Walton. A software engineering experiment in software component generation. In *18th International Conference in Software Engineering*, pages 542–553, 1996.
44. John Launchbury. *Projection Factorisations in Partial Evaluation.* Distinguished Dissertations in Computer Science. Cambridge University Press, 1991.
45. Julia L. Lawall and Peter Thiemann. Sound specialization in the presence of computational effects. In M. Abadi and T. Ito, editors, *Proceedings of the 3rd International Symposium on Theoretical Aspects of Computer Software (TACS'97)*,

number 1281 in Lecture Notes in Computer Science, pages 165–190, September 1997.
46. Chin Soon Lee, Neil D. Jones, and Amir M. Ben-Amram. The size-change principle for program termination. In *ACM Symposium on Principles of Programming Languages*, volume 28, pages 81–92. ACM Press, January 2001.
47. Michael Leuschel. On the power of homeomorphic embedding for online termination. In *Static Analysis. Proceedings*, volume 1503, pages 230–245. Springer-Verlag, September 1998.
48. Michael Leuschel and Maurice Bruynooghe. Logic program specialisation through partial deduction: Control issues. *Theory and Practice of Logic Programming*, 2002.
49. Naomi Lindenstrauss, Yehoshua Sagiv, and Alexander Serebrenik. Termilog: A system for checking termination of queries to logic programs. In Orna Grumberg, editor, *Computer Aided Verification, 9th International Conference, CAV '97, Haifa, Israel, Jun 22–25, 1997*, volume 1254 of *Lecture Notes in Computer Science*, pages 444–447. Springer, 1997.
50. Y.Ã. Liu. Efficiency by incrementalization: an introduction. *Journal of Higher-Order and Symbolic Computation*, 13(4):289–313, 2000.
51. John W. Lloyd and John C. Shepherdson. Partial evaluation in logic programming. *Journal of Logic Programming*, 11:217–242, 1991.
52. Dylan McNamee, Jonathan Walpole, Calton Pu, Crispin Cowan, Charles Krasic, Ashvin Goel, Perry Wagle, Charles Consel, Gilles Muller, and Renaud Marlet. Specialization tools and techniques for systematic optimization of system software. *ACM Transactions on Computer Systems*, 19(2):217–251, 2001.
53. Torben Mogensen. Partially static structures in a self-applicable partial evaluator. In D. Bjørner, A.P. Ershov, and N.D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 325–347. North-Holland, 1988.
54. Eugenio Moggi, Walid Taha, Zine-El-Abidine Benaissa, and Tim Sheard. An idealized MetaML: Simpler, and more expressive. In *European Symposium on Programming*, volume 1576, pages 193–207, 1999.
55. Oregon Graduate Institute Technical Reports. P.O. Box 91000, Portland, OR 97291–1000, USA. Available online from `ftp://cse.ogi.edu/pub/tech-reports/README.html`.
56. Rosza Péter. *Rekursive Funktionen (Recursive Functions)*. Académiai Kiadó, Budapest (Academic Press, New York), 1951 (1976).
57. Massimiliano Poletto, Wilson C. Hsieh, Dawson R. Engler, and M. Frans Kaashoek. 'C and tcc: A language and compiler for dynamic code generation. *ACM Transactions on Programming Languages and Systems*, 21(2):324–369, March 1999.
58. Aravinda Prasad Sistla, Moshe Y. Vardi, and Pierre Wolper. The complementation problem for Büchi automata with applications to temporal logic. *Theoretical Computer Science*, 49:217–237, 1987.
59. Konstantinos F. Sagonas, Terrance Swift, and David Scott Warren. XSB as an efficient deductive database engine. In Richard T. Snodgrass and Marianne Winslett, editors, *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data, Minneapolis, Minnesota, May 24–27, 1994*, pages 442–453. ACM Press, 1994.
60. Danny De Schreye, Robert Glück, Jesper Jørgensen, Michael Leuschel, Bern Martens, and Morten H. B. Sørensen. Conjunctive partial deduction: Foundations, control, algorithms, and experiments. *Journal of Logic Programming*, 41(2&3):231–277, 1999.
61. Ulrik Schultz. Partial evaluation for class-based object-oriented languages. In *PADO*, pages 173–197, 2001.

62. Peter Sestoft. Bibliography on partial evaluation and mixed computation (bibtex format, online `ftp://ftp.diku.dk/diku/semantics/partial-evaluation/`). Technical report, DIKU (Computer Science, University of Copenhagen), 2001.
63. Tim Sheard. Using MetaML: A staged programming language. *Lecture Notes in Computer Science*, 1608:207–??, 1999.
64. Litong Song and Yoshihiko Futamura. A new termination approach for specialization. In *[70]*, pages 72–91, 2000.
65. Morten Heine Sørensen and Robert Glück. An algorithm of generalization in positive supercompilation. In J.W. Lloyd, editor, *Logic Programming: Proceedings of the 1995 International Symposium*, pages 465–479. MIT Press, 1995.
66. Chris Speirs, Zoltan Somogyi, and Harald Søndergaard. Termination analysis for Mercury. In Pascal Van Hentenryck, editor, *Static Analysis, Proceedings of the 4th International Symposium, SAS '97, Paris, France, Sep 8–19, 1997*, volume 1302 of *Lecture Notes in Computer Science*, pages 160–171. Springer, 1997.
67. Michael Sperber and Peter Thiemann. Generation of LR parsers by partial evaluation. *ACM Transactions on Programming Languages and Systems*, 22(2):224–264, March 2000.
68. Walid Taha. A sound reduction semantics for untyped CBN multi-stage computation. or, the theory of MetaML is non-trival. *ACM SIGPLAN Notices*, 34(11):34–43, November 1999. Extended abstract.
69. Walid Taha. *Multi-Stage Programming: Its Theory and Applications*. PhD thesis, Oregon Graduate Institute of Science and Technology, 1999. Available from [55].
70. Walid Taha, editor. *Semantics, Applications, and Implementation of Program Generation*, volume 1924 of *Lecture Notes in Computer Science*, Montréal, 2000. Springer-Verlag.
71. Walid Taha, Paul Hudak, and Zhanyong Wan. Directions in functional programming for real(-time) applications. In *the International Workshop on Embedded Software (ES '01)*, volume 221 of *Lecture Notes in Computer Science*, pages 185–203, Lake Tahoe, 2001. Springer-Verlag.
72. Walid Taha, Henning Makholm, and John Hughes. Tag elimination and Jones-optimality. In *PADO*, pages 257–275, 2001. http://cs-www.cs.yale.edu/homes/taha/publications/preprints/pado00.dvi.
73. Walid Taha and Tim Sheard. MetaML and multi-stage programming with explicit annotations. *Theoretical Computer Science*, 248(1–2):211–242, October 2000.
74. Peter Thiemann. A unified framework for binding-time analysis. In M. Bidoit and M. Dauchet, editors, *TAPSOFT '97: Theory and Practice of Software Development, Lille, France, April 1997. (Lecture Notes in Computer Science, vol. 1214)*, pages 742–756. Springer-Verlag, 1997.
75. Peter Thiemann. Aspects of the pgg system: Specialization for standard scheme. In Hatcliff et al. [35], pages 412–432.
76. Valentin F. Turchin. A supercompiler system based on the language Refal. *SIGPLAN Notices*, 14(2):46–54, February 1979.
77. Philip Wadler. Deforestation: Transforming programs to eliminate trees. In H. Ganzinger, editor, *ESOP'88. 2nd European Symposium on Programming, Nancy, France, March 1988. (Lecture Notes in Computer Science, vol. 300)*, pages 344–358. Springer-Verlag, 1988.
78. Hongwei Xi. Dependent types for program termination verification. volume 15, pages 91–132, 2002.