

What *Not* to Do When Writing an Interpreter for Specialisation

Neil D. Jones
DIKU, University of Copenhagen
e-mail: neil@diku.dk

Abstract. A partial evaluator, given a program and a known “static” part of its input data, outputs a *specialised* or *residual* residual program in which computations depending only on the static data have been performed in advance.

Ideally the partial evaluator would be a “black box” able to extract nontrivial static computations whenever possible; which never fails to terminate; and which always produces residual programs of reasonable size and maximal efficiency, so all possible static computations have been done. *Practically* speaking, partial evaluators fall short of this goal; they sometimes loop, sometimes pessimise, and can explode code size.

A partial evaluator is analogous to a spirited horse: while impressive results can be obtained when used well, the user must know what he/she is doing. Our thesis is that *this knowledge can be communicated* to new users of these tools.

This paper presents a series of examples, concentrating on a quite broad and on the whole successful application area: using specialisation to remove interpretational overhead. It presents both positive and negative examples, to illustrate the effects of program style on the efficiency and size of the of target programs obtained by specialising an interpreter with respect to a known source program.

1 Introduction

1.1 The phenomena

In my opinion (and, I hope, in yours too after looking at the following examples), there exist a number of *real phenomena* in specialisation, i.e. frequently occurring specialisation problems that are nontrivial to solve; there exist styles of programming that can be expected to give *any* partial evaluator difficulties; and one can *learn and communicate* programming style that leads to good specialisation, i.e. high speedups without explosion of code size.

1.2 Making implicit knowledge explicit

Some of the following examples will make the experienced reader grit his or her teeth, saying “I would never dream of programming my interpreter in such an absurd way!” This is indeed true — but inexperienced users often commit just these errors or unfortunate ways of programming (I speak from experiences with

students.) A major point of this article is to make manifest some of the *implicit knowledge* we have acquired in our field, so newcomers can avoid encountering and having to figure out how to solve the same problems in “the school of hard knocks.”

1.3 Dependence on particular specialisation algorithms

The advice alluded to in this paper’s title would seem, at least to some extent, relative to which *class of specialisation algorithms* is used. I have carefully tried to write so as to avoid having to take account of remarks of the form “yes, but I can solve that problem using feature XXX of my partial evaluator.”

Accounting for such possible remarks, followed by responses (“even so, ...”), etc. could easily tend to lead to an infinite regress, or to a protracted discussion whose line would be hard to follow. Even worse, it would make it hard at the end to say what really had been accomplished by the discussion.

2 Concepts and notations

We use familiar notations, for example D is a data set that contains all first-order values, including *all program texts*. Inputs may be paired, e.g. $d.e$ is in D if d is in D and e are in D ($d.e$ may also be written $(d.e)$ or (d,e)). For concreteness the reader may think of D as Lisp/Scheme lists, and programs as written in Scheme. However the points below are not language-dependent, and some example interpreters will be written in imperative languages.

The *semantic function* of a program p is a partial function $\llbracket p \rrbracket : D \rightarrow D \cup \{\perp\}$, so $\llbracket p \rrbracket(d) = d' \in D$ is the result of running p on input $d \in D$ (and is \perp if p fails to terminate on d .) Further, $time_p(d) \in \mathbb{N}$ is the time taken to compute $\llbracket p \rrbracket(d)$ (and is ∞ if p fails to terminate on d .)

We will assume the underlying implementation language is defined by an (unspecified) operational semantics. Computations are given by *sequences* (for imperative languages) or, more generally, by *computation trees*. Further, program run time $time_p(d)$ is assumed roughly proportional to the size of the computation tree deducing $\llbracket p \rrbracket(d) = d'$ (relevant to the discussion of linear speedup).

Given program p and “static” input $s \in D$, program p_s is the result of *specialising* p to s if $\llbracket p \rrbracket(s.d) = \llbracket p_s \rrbracket(d)$ for every “dynamic” input $d \in D$. Well-known terms are to call p_s a *residual program* for p with respect to s , or a version of p *specialised to* s .

A *partial evaluator* (or *specialiser*) is a program mix such that for every program p and “static” input $s \in D$, program $\llbracket mix \rrbracket(p.s)$ is a version of p specialised to s .

Efficiency: clearly a trivial mix is easy to build (e.g. by Kleene’s *s-m-n* Theorem of the 1930s.) The practical goal is to make $p_s = \llbracket mix \rrbracket(p.s)$ *as efficient as possible*, by performing many of p ’s computations — ideally, all those that

depend on s alone. The *speedup* realised by *mix* on program p and static input s is the ratio (note that for each parameter s it is a function of d):

$$\text{speedup}_s(d) = \frac{\text{time}_p(s, d)}{\text{time}_{p_s}(d)}$$

Some programs are well-suited to specialisation, yielding substantial speedups, and others are not. Given a specialiser *mix*, a *binding-time improvement* is the transformation of a program p into an equivalent program q such that $\llbracket p \rrbracket = \llbracket q \rrbracket$, but specialisation of q to various s gives larger speedups than specialisation of p .

Other languages. If unspecified, as above, we assume a standard implementation language L is intended. Otherwise, if another language S is used, we make this explicit as follows. The *semantic function* of a program p is a partial function $\llbracket p \rrbracket_S : D \rightarrow D \cup \{\perp\}$, so $\llbracket p \rrbracket^S(d)$ is the result of running p on input $d \in D$ (\perp if p fails to terminate on d .) Further, $\text{time}_p^S(d)$ is the time taken to compute $\llbracket p \rrbracket^S(d)$ (and is ∞ if p fails to terminate on d .)

Interpreters. Program i is an *interpreter for language S* if for all S -programs s and data $d \in D$

$$\llbracket i \rrbracket(s.d) = \llbracket s \rrbracket^S(d)$$

3 Assumptions in this paper

For definiteness and to avoid having to “hit a moving target,” I now list the assumptions of this paper. Readers are welcome to reply by describing how other techniques can solve some or all of these problems; but should be honest enough to admit it if their techniques introduce yet other problems.

3.1 Restrictions in this discussion

1. The programs to be specialised are assumed expressed in an *untyped first-order functional* language, using an informal syntax. Consequently all our interpreter texts, etc. are in this form; however the *languages they interpret* may be functional, imperative, higher-order, logic, etc.
2. Offline specialisation is assumed: before actual specialisation begins, all of program p ’s operations and calls or other control transfers are *annotated* as either **static**: do at specialisation time; or as **dynamic**: generate code to be executed at run time.
3. Every function parameter (in the program to be specialised) is either *totally static* or *totally dynamic*. Thus there is no *partially static data* in these examples.
4. We assume that a BTA (*binding-time analysis*) does the annotations, e.g. separately as in Schism, or integrated into the text as in Similix:

```

append(X_S,Y_D) = if_S X =_S 'nil then Y
                  else cons_S(hd_S(X), append_S(tl_S X, Y))

```

5. No *distributivity rules* are used, e.g. to transform from 1 to 2 below:

```

1: static1      +      (if dynamic_test then static2 else static3)
2: if dynamic_test then (static1+static2) else (static1+static3)

```

This has been called “CPS specialisation” – a rather unfortunate name in my opinion since the concept makes perfect sense without continuations, or even without higher-order functions. For instance, distributivity or “CPS specialisation” can even be done in logic programming:

```

p(D, Result) :- test(D, S), Result is S1 + S.
test([],      S2).
test([X|Xs], S3).

```

is equivalent to

```

p([], Result)      :- Result is S1 + S2.
p([X|Xs], Result) :- Result is S1 + S3.

```

6. Don’t use *The Trick* to make static otherwise dynamic values, or η -conversion to accomplish similar effects. (The Trick will be mentioned later, but I want to avoid it unless explicitly mentioned.)
7. No global optimisations are done, e.g.

dead code elimination (a logic programming analog is removing code that is certain to fail); or

common subexpression elimination, especially ones containing function calls (a logic programming analog is removing duplicated goals).

3.2 Remarks on the restrictions

The lack of partially static data can be overcome in more than one way; for instance it is of less significance for online specialisers such as Fuse [11], super-compilation [10], or most logic programming specialisers [9, 4] ; but then new problems arise concerning control, termination, etc. An offline specialiser such as Similix [1] or Schism [2] usually does analyses, often involving some form of tree grammar [7], to statically describe all possible runtime data shapes and to exploit this.

Alternately, a *bifurcation* transformation [8, 3] splits a program with partially static data into one part with completely static data, and another part with both static and dynamic data.

Online and offline specialisation. I maintain that many of specialisation *problems* are essentially independent of which type of specialisation *algorithm* one uses, or of whether one is specialising functional, logic, or imperative programs. This holds even though a particular problem might manifest itself in different ways, e.g. in an offline functional specialiser as a *nontermination problem*, or in logic programming as causing an *unnecessarily conservative* specialisation, leading to little or no speedup.

To be more concrete: several of the “binding-time improvements” exemplified below to avoid nontermination or code size explosion would improve specialisation of logic programs as well as functional programs.

Claim: binding-time improvements performed to improve offline specialiser results often improve online specialisation results too. This can either be by causing more to be statically computable, or by reducing the size of the residual program.

4 Bounded static variation: the key to termination of the specialiser

Nontermination at specialisation time can occur for one of two reasons: to

- Attempt to build a single infinitely large command, expression or literal; or to
- Attempt to build a residual program containing infinitely many program points (labels, defined function, or procedures).

The basic cause is the same: the specialiser’s *unfolding strategy* is the main reason that the one or the other misbehaviour occurs. In either case nontermination makes a specialiser less suitable for use by nonexperts, and quite unsuitable for automatic use. Further, failure to terminate gives the user very little to go on to discover the problem’s cause, in order to repair it.

Two attitudes to nontermination have been seen: in *functional programming*, infinite static loops are often seen as the user’s fault, so there is no need for the specialiser to account for them. *Logic programming* most often requires that the specialiser *always terminate*. One reason is the Prolog “negation as finite failure semantics, which means that changing termination properties can change semantics, and even answer sets. Both views can be either defended or attacked.

Current work on functional program specialisation concerns a BTA conservative enough to guarantee that specialisation always terminates, and still gives good results when specialising interpreters [5].

4.1 Pachinko, execution, and specialisation

Normal execution follows only one sequential control thread. Specialisation, however, must account for *all possible control threads* the subject program could enter into, for all possible dynamic inputs.

An analogy: the popular Japanese “Pachinko” entertainment involves steel balls that fall through a lattice of pins. Sequential execution corresponds to dropping only one ball, which thus traverses only one trajectory. On the other hand, specialisation corresponds to dropping an infinite set of balls at once — all those which share the given static input value, but have arbitrary dynamic input values. Reasoning about specialisation, e.g. residual program finiteness or efficiency, requires reasoning about such a (most often infinite) set of trajectories.

Reachable states. Let function f defined in program p have number of parameters denoted by “arity(f)” and let parameter values range over the set V . The i th parameter of f will be written as f_i .

A *state* is a pair $(f, \bar{\alpha})$ where $\bar{\alpha} \in V^{\text{arity}(f)}$. Notation: $\bar{\alpha}_i$ is the i th component of parameter tuple $\bar{\alpha} \in V^{\text{arity}(f)}$. Thus $\bar{\alpha}_i$ is the value of f_i in state $(f, \bar{\alpha})$.

State $(f, \bar{\alpha})$ can *reach* state $(g, \bar{\beta})$, written $(f, \bar{\alpha}) \rightarrow^* (g, \bar{\beta})$, if computation of f on parameter values $\bar{\alpha}$ immediately requires the value of g on parameter values $\bar{\beta}$. (Formal definition of this is straightforward.)

4.2 Definition of bounded static variation.

Let ‘,’ be the parameter *tuple concatenation operator*, suppose f_1 is the initial (entry) function of program p , and let $s, d \in \mathbb{N}$, where $s + d = \text{arity}(f)$. We assume (for simplicity) that the first s parameters of f_1 are static, and the remaining d parameters are dynamic. The set of *statically reachable states* $\mathcal{SR}(\bar{\sigma})$ consists of all states $(f, \bar{\alpha})$ reachable in 0, 1, or more steps from some initial call with $\bar{\sigma} \in V^s$ as static program input, and an arbitrary dynamic input $\bar{\delta}$:

$$\mathcal{SR}(\bar{\sigma}) = \{(f, \bar{\alpha}) \mid \exists \bar{\delta} \in V^d : (f_1, \bar{\sigma}; \bar{\delta}) \rightarrow^* (f, \bar{\alpha})\}$$

The i th parameter f_i of f is of *bounded static variation*, written $\text{BSV}(f_i)$, iff for all $\bar{\sigma} \in V^s$, the following set is finite:

$$\{\bar{\alpha}_i \mid (f, \bar{\alpha}) \in \mathcal{SR}(\bar{\sigma})\}$$

This captures the intuitive notion that f ’s i th parameter only varies finitely during all computations in which the static program input is fixed to some $\bar{\sigma}$, and dynamic program input varies freely.

Such a parameter may, in principle, be classified as “static.” Most such parameters are computable from the static program inputs; but exceptions do occur, for example (if x is dynamic):

$$\text{sign}(x) = 1 \text{ if } x > 0, \text{ else } 0$$

5 Compositionality and semicompositionality of interpreters

Suppose an interpreter’s parameters have been classified as either nonsyntactic, or as *syntactic*, in which case their values range only over over phrases from the source language (perhaps or perhaps not extracted from the interpreted source program). The interpreter is defined to be **compositional** if for any definition of a function with syntactic parameters, the syntactic parameters of all function calls are *proper substructures* of the calling function’s syntactic parameters. All such parameters are, quite obviously, of BSV (bounded static variation.)

In a compositional interpreter, in any computation, *all* syntactic parameters of any functions called must be substructures of the source program being

interpreted. A weaker requirement, also guaranteeing BSV, is to be **semicompositional**, in which case called parameters must be *substructures of the source program being interpreted*, but need not be properly smaller in every call.

The compositionality requirement ensures that the interpreter can be specialised with respect to its source program input by simple unfolding — a process *guaranteed to terminate*, though likely to be far from fully satisfactory unless additional specialisation-time computations are done. Semicompositionality does not guarantee termination, but guarantees finiteness of the set of specialised program points — so termination can be ensured by a sufficiently conservative unfolding strategy.

6 Where do interpreters come from?

An interpreter is just a program. It can be used to define a new language (so a source program meaning is whatever the interpreter yields when given it and *its* input); or to implement a language already defined by other means, e.g. by an operational or denotational semantics.

6.1 Interpreters derived from execution models

Many interpreters simply embody some model of runtime execution, e.g. Pascal’s “stack of activation records” or Algol 60’s “thunk” mechanism. Following is a simple example.

An interpreter used in DIKU instruction. Computation is by a linear sequence

$$(S_1, M_1, C_1) \rightarrow (S_2, M_2, C_2) \rightarrow \dots$$

of states of form (S, M, C) . S is a *computation stack*, $M : \{x_0, x_1, \dots\} \rightarrow \text{Value}$ is a *memory* containing the values of variables x_i , and C is a *control stack*, containing bits of commands and expressions that remain to be evaluated/executed. (Remark: C is in essence a materialisation in the form of data of the program’s *continuation* from the current point of execution.)

For readability we describe this interpreter (call it *int-instruc*) by a set of transition rules in Figure 6.1. Note that the C component is neither compositional nor semicompositional. The underlines, e.g. n , indicate numeric runtime values.

6.2 Interpreters derived from denotational semantics

A *denotational semantics* consists of

- a collection of *domain equations* specifying the partially ordered value sets to which the meanings (denotations) of program pieces and auxiliaries such as environments will belong, and
- a collection of *semantic functions* mapping program pieces to their meanings in a compositional way (as defined in Section 5).

Rules for Expressions

Constant :	$\langle S, M, c \cdot C \rangle$	$\Rightarrow \langle c \cdot S, M, C \rangle$
Variable :	$\langle S, M, \mathbf{x}i \cdot C \rangle$	$\Rightarrow \langle M(\mathbf{x}i) \cdot S, M, C \rangle$
Composite :	$\langle S, M, (e_1 \text{ op } e_2) \cdot C \rangle$	$\Rightarrow \langle S, M, e_1 \cdot e_2 \cdot \text{op} \cdot C \rangle$
Operator :	$\langle \underline{n_2} \cdot \underline{n_1} \cdot S, M, \text{op} \cdot C \rangle$	$\Rightarrow \langle \underline{n_1 \text{ op } n_2} \cdot S, M, C \rangle$

Rules for Programs

	Null :	$\langle S, M, \text{null} \cdot C \rangle$	$\Rightarrow \langle S, M, C \rangle$
	Assign :	$\langle S, M, (\mathbf{x}i := e) \cdot C \rangle$	$\Rightarrow \langle i \cdot S, M, e \cdot \text{assign} \cdot C \rangle$
[tb]	Sequence :	$\langle S, M, (p_1 ; p_2) \cdot C \rangle$	$\Rightarrow \langle S, M, p_1 \cdot p_2 \cdot C \rangle$
	Test :	$\langle S, M, (\text{if } b \text{ then } p_1 \text{ else } p_2) \cdot C \rangle$	$\Rightarrow \langle p_1 \cdot p_2 \cdot S, M, b \cdot \text{if} \cdot C \rangle$
	Loop :	$\langle S, M, (\text{while } b \text{ do } p_1) \cdot C \rangle$	$\Rightarrow \langle b \cdot p_1 \cdot S, M, b \cdot \text{while} \cdot C \rangle$

Rules for assign, if and while

$\langle \underline{n} \cdot i \cdot S, M, \text{assign} \cdot C \rangle$	$\Rightarrow \langle S, M[\mathbf{x}i \mapsto \underline{n}], C \rangle$
$\langle \text{true} \cdot p_1 \cdot p_2 \cdot S, M, \text{if} \cdot C \rangle$	$\Rightarrow \langle S, M, p_1 \cdot C \rangle$
$\langle \text{false} \cdot p_1 \cdot p_2 \cdot S, M, \text{if} \cdot C \rangle$	$\Rightarrow \langle S, M, p_2 \cdot C \rangle$
$\langle \text{true} \cdot b \cdot p_1 \cdot S, M, \text{while} \cdot C \rangle$	$\Rightarrow \langle S, M, p_1 \cdot (\text{while } b \text{ do } p_1) \cdot C \rangle$
$\langle \text{false} \cdot b \cdot p_1 \cdot S, M, \text{while} \cdot C \rangle$	$\Rightarrow \langle S, M, C \rangle$

Figure 6.1: Instructional interpreter.

In practice a denotational semantics resembles a program in a modern functional language such as ML, Haskell, or Scheme. Indeed the designers of such languages have to a certain extent built denotational concepts into them. One instance is syntax and implementation techniques to make programs with higher-order functions easy to write and not too expensive to use. Another notable instance is *continuations*, which have come from the metalevel of language descriptions (an invention devised to formalise **goto**) down to the subject language level.

A difference from operational definitions is that fixpoints are conceptually evaluated “all at once” in denotational semantics, and only unfolded “upon demand” in functional languages. Still, the net effect is that many denotational language definitions can simply be transliterated into functional programs.

One problem with a denotational semantics is that it necessarily uses a “universal domain,” capable of expressing *all possible run-time values* of all programs that can be given meaning. This, combined with the requirement of compositionality, often leads to an (over-)abundance of higher-order functions. These two together imply that domain definitions very often must be both *recursive and higher-order*, creating both mathematical and implementational complications (e.g. reflexive domains and contravariance in a function space functor).

Examples are familiar, and omitted for brevity.

6.3 Interpreters derived from operational semantics

Operational semantics seems more manageable, since emphasis is on judgements formed from first-order objects (although finite functions are allowed). They come in two flavours: *big-step* in which a judgement is of form

$$\dots \vdash Exp \Rightarrow \text{Final answer}$$

and *small-step* in which a judgement is of form

$$\dots \vdash Exp \Rightarrow Exp'$$

which describes expression evaluation by reducing one expression repeatedly to another, until a final answer is obtained. In both cases a language definition is a set of *inference rules*. A judgement is proven to hold by constructing a proof tree with it as root, and where each local subtree is an instance of an inference rule. Examples are omitted for brevity, but the approaches should be clear from the following program implementation examples.

Transliterating these two types of operational semantics into program form gives interpreters of rather different characteristics. Big-step semantics are conceptually nearer denotational semantics than small-step semantics, since source syntax and program meanings are clearly separated. Small-step semantics work by symbolic source-to-source code reduction, and so are nearer equational or rewrite theories, for example traditional treatments of the λ -calculus.

An interpreter derived from a big-step operational semantics. The interpreter `int-big-step` of Figure 6.2 implements a “big-step” semantics.

Interpreters derived from small-step operational semantics. At first sight, a small-step operational semantics seems quite unsuitable for specialisation. A familiar example is the usual definition of reducibility in the lambda calculus. This consists of a few rules for reduction, e.g. α, β, η , and *context rules* stating that these can be applied when occurring inside other syntactic constructs. Problems with direct implementation include:

- *Nondeterminism*: many redexes may be reducible within the same λ -expression;
- *Nondirectionality of computation*: rewriting may occur either from left to right or vice versa (conversion versus reduction); and
- *Syntax rewriting*: the subject program’s syntax is continually changed, in ways impossible statically to predict while specialising an interpreter.

These are also problems for implementations, often resolved as follows:

- *Nondeterminism*: reduction is sometimes restricted to occur only from the outermost syntactic operator, using call-by-value or some other fixed strategy;
- *Nondirectionality*: rewriting occurs only from left to right, until a normal form is obtained.

More generally, nondeterminism is often resolved by defining explicit *evaluation contexts* $C[\]$. Each is an “expression with a hole in it” indicated by the $[\]$.

```

; Type of Run: Program x Program_value -> Output_value

Run(pgm, input) = Eval(e1, ns1, (list input), pgm) where
                    e1 = lookbody(first_fcn(pgm), pgm)
                    ns1 = looknames(first_fcn(pgm), pgm)

; Type is: Expression x Namelist x Valuelist x Program ->
Value

Eval(e, ns, vs, pgm) =    case e of
    constant            : constant
    'X                  : lookpar(X, ns, vs)

    '(e1 binop e2)      : apply(binop, v1, v2) where
                        v1 = Eval(e1, ns, vs, pgm)
                        v2 = Eval(e2, ns, vs, pgm)

    '(if e0 e1 e2)      : if Eval(e0, ns, vs, pgm)
                        then Eval(e1, ns, vs, pgm)
                        else Eval(e2, ns, vs, pgm)

    '(call f es)        : Eval(e1, ns1, vs, pgm) where
                        e1 = lookbody(f, pgm)
                        ns1 = looknames(f, pgm)
                        vs1 = Evlist(es, ns, vs, pgm)

Evlist(es, ns, vs, pgm) =    case es of
    'nil                : nil
    'cons(e1 es')       : cons(Eval(e1, ns, vs, pgm),
                                Evlist(es', ns, vs, pgm))

```

Figure 6.2: Big-step semantics, in program form.

Determinism may be achieved by defining evaluation contexts so the *unique decomposition property* holds: any expression E has exactly one decomposition of form $E = C[E']$ where $C[]$ is an evaluation context. Given this, computation proceeds through a series of rewritings of such configurations.

The problem of syntax rewriting causes particular problems for specialisation. One aid is the use of *closures* instead of rewriting, essentially a “lazy” form of β -reduction. This easily allows building a semicompositional big-step interpreter, but some problems remain for small-step interpreters.

Some small-step semantics have a property that can be thought of as *dual to semicompositionality*: every decomposition $E = C[E']$ that occurs in a given computation is such that $C[]$ consists of *the original program with a hole in it*, and E' is a normal form value, e.g. a number or a closure.

Such a dual semicompositional semantics can usually be specialised well. The condition is violated, however (for an example), if the semantics implements

function or procedure calls by substituting the body of the called function or procedure in place of the call itself (a similar example is seen in Section 7.2 below).

7 Bounded or unbounded static variation?

7.1 Analysis of the instructional interpreter.

The interpreter `int-instruc` works quite well for computation and proof, and has been used for many exercises and proofs of program properties in the course “Introduction to Semantics.” On the other hand, it does not specialise at all well! The source of the problem: suppose one is given the source program p , i.e. the initial control stack contents are $C = p \cdot ()$, but that the memory M is unknown, i.e. dynamic.

Clearly the value stack S must also be dynamic, since it receives values from M as in the second transition rule. But this implies that *anything* put into S and then retrieved from it again must also be dynamic. In particular this includes: the index i of a variable to be assigned; the **then** and **else** branches of an **if** statement; and the test of a **while** statement. Consequently *all of these* become dynamic, and so essentially no specialisation at all occurs.

A better version for specialisation. The problem is easy to fix: just place the above-mentioned syntactic entities on the *control stack*: the index i of a variable to be assigned; the **then** and **else** branches of an **if** statement; and the test of a **while** statement. This leads to the interpreter of Figure 7.1, again as a set of transition rules.

Bounded static variation of the control stack. In this version stack C is built up only from pieces of the original program. Further, given any one source program C takes on only finitely many different possible values. Even though C can grow when a **while** command is processed, it is easy to see that it cannot grow unboundedly for any one, fixed, source progra. Thus C may safely be annotated as “static.”

This version specialises much better. The effect is to yield a target program consisting of labeled transitions from one (S, M) pair to another — in essence with one transition (more or less) for each point in the original program, and no source code at all. Many of these transitions involve no change to either S or M , and so are removable by a trivial “transition compression.”

7.2 A small but problematic extension: procedures.

Procedures are easily added by adding a fourth component $(S, M, C, Pdefs)$ where $Pdefs$ is a list of mutually recursive procedure definitions. A procedure call is handled by looking up the procedure’s name in $Pdefs$, and replacing the call by the body of the called procedure.

Rules for Expressions

Constant : $\langle S, M, c \cdot C \rangle \Rightarrow \langle c \cdot S, M, C \rangle$
 Variable : $\langle S, M, \mathbf{x}i \cdot C \rangle \Rightarrow \langle M(\mathbf{x}i) \cdot S, M, C \rangle$
 Composite : $\langle S, M, (e_1 \text{ op } e_2) \cdot C \rangle \Rightarrow \langle S, M, e_1 \cdot e_2 \cdot \text{op} \cdot C \rangle$
 Operator : $\langle \underline{n_2} \cdot \underline{n_1} \cdot S, M, \text{op} \cdot C \rangle \Rightarrow \langle \underline{n_1 \text{ op } n_2} \cdot S, M, C \rangle$

Rules for Programs

Null : $\langle S, M, \text{null} \cdot C \rangle \Rightarrow \langle S, M, C \rangle$
 Assign : $\langle S, M, (\mathbf{x}i := e) \cdot C \rangle \Rightarrow \langle S, M, e \cdot \text{assign} \cdot i \cdot C \rangle$
 Sequence : $\langle S, M, (p_1 ; p_2) \cdot C \rangle \Rightarrow \langle S, M, p_1 \cdot p_2 \cdot C \rangle$
 Test : $\langle S, M, (\text{if } b \text{ then } p_1 \text{ else } p_2) \cdot C \rangle \Rightarrow \langle S, M, b \cdot \text{if} \cdot p_1 \cdot p_2 \cdot C \rangle$
 Loop : $\langle S, M, p = (\text{while } b \text{ do } p_1) \cdot C \rangle \Rightarrow \langle S, M, b \cdot \text{while} \cdot p_1 \cdot p \cdot C \rangle$

Rules for assign, if and while

$\langle \underline{n} \cdot S, M, \text{assign} \cdot i \cdot C \rangle \Rightarrow \langle S, M[\mathbf{x}i \mapsto \underline{n}], C \rangle$
 $\langle \text{true} \cdot S, M, \text{if} \cdot p_1 \cdot p_2 \cdot C \rangle \Rightarrow \langle S, M, p_1 \cdot C \rangle$
 $\langle \text{false} \cdot S, M, \text{if} \cdot p_1 \cdot p_2 \cdot C \rangle \Rightarrow \langle S, M, p_2 \cdot C \rangle$
 $\langle \text{true} \cdot S, M, \text{while} \cdot p_1 \cdot p \cdot C \rangle \Rightarrow \langle S, M, p_1 \cdot p \cdot C \rangle$
 $\langle \text{false} \cdot S, M, \text{while} \cdot p_1 \cdot p \cdot C \rangle \Rightarrow \langle S, M, C \rangle$

Figure 7.1: Modified instructional interpreter.

Rule for Procedure declarations

$\langle S, M, (\text{procedure } P : p) \cdot C, Pdefs \rangle \Rightarrow \langle S, M, C, P : p \cdot Pdefs \rangle$

Rule for Procedure calls

$\langle S, M, (\text{call } P) \cdot C, \dots P : p \dots \rangle \Rightarrow \langle S, M, p \cdot C, \dots P : p \dots \rangle$

Even though this looks quite innocent, it causes control stack C *no longer to be of bounded static variation*. To see why, consider say the factorial function defined recursively:

```

Procedure Fac :
  if N = 0 then Result := 1
  else {N := N-1; call Fac; N := N+1; Result:= N * Result}
  
```

The control stack's depth will be proportional to the initial value of N , which is dynamic!

Can this problem be solved? Yes, by using “the trick.” The point is that even though $Pdefs$ is a dynamic data structure, each of its components comes from the source program p , which fact can be exploited to gain good specialisation.

7.3 Analysis of the “big-step” interpreter

Interpreter `int-big-step` from Figure 6.2 specialises well. Even though not compositional (as defined in Section 6.2), it is *semicompositional* since the parameters `e`, `ns`, and `pgm` of `Eval` are always substructures of the original source program.

In this case recursion causes no problems; there is no stack, and nothing can grow unboundedly as in the extended instructional interpreter, or as in many small-step semantics.

A non-semicompositional variant. Now consider the interpreter of Figure 7.1, extended by a “LET” construction. This is easy to modify by adding an extra CASE:

```
Eval(e, ns, vs, pgm) =   case e of ...

      '(let Y = e1 in e2) : Eval(e2, (cons 'X ns), (cons v1 vs), pgm)
                           where   v1 = Eval(e1, ns, vs, pgm)
```

Now `ns` can take on values that are *not substructures of* `pgm`. Even though `ns` can grow, it only grows when `e` shrinks; further, it can be *reset* by a function call, but only to values that are part of `pgm`. Consequently `ns` *cannot grow unboundedly* as a function of the size of `int-big-step`’s static input `pgm`, and so is of bounded static variation.

7.4 Dynamic binding.

Suppose in this interpreter the “function call” construction were implemented differently, as follows:

```
Eval(e, ns, vs, pgm) =   case e of ...
      '(call f es) : Eval(e1, (append ns1 ns), (list v0), pgm) where
                           e1  = lookbody(f, pgm)
                           ns1 = looknames(f, pgm)
                           v0  = Evlist(es, ns, vs, pgm)
```

This can easily be seen to cause `ns` to become of *unbounded static variation*. Interestingly, the dynamic name binding in the call has an indirect effect on `Eval`: the interpreter parameter `ns` is forced to become dynamic. Intuitively, the reason is that the length of `ns` is no longer tied to any syntactic property of `pgm`. For more technical details, see [5].

The consequence is (as in Lisp) that target programs obtained by specialising `int-big-step` will contain parameter *names* as well as their values, and function `lookpar` will become a run-time procedure rather than a compile-time one, substantially increasing target program running times and space usage.

8 On speedup

Assuming that the specialiser terminates, it is natural to ask: how much speedup can be obtained; how does the amount of speedup relate to the way the subject program is written; and how can one avoid code size explosion?

We begin with a resume (and improvement) of the argument [6] that one can (usually) expect at most speedup by a linear multiplicative factor. We will see, however, that the size of the factor can depend on the static inputs, e.g. one often observes, for instance in pattern matching, that larger static data leads to larger speedups as a function of the size of d . The argument below also applies to *deforestation* and *supercompilation*.

8.1 Embedding computations into original computations

In most partial evaluators the residual computations can be described more simply than those of the specialiser. Consider a given program p , and static and dynamic inputs s, d . Let p_s be the result of specialising p to s .

A first observation (on all functional or imperative partial evaluators I have seen): there exists an order-preserving *injective mapping* ψ

- from the operations done in the computation of $\llbracket p_s \rrbracket(d)$
- to the operations done in the original computation of $\llbracket p \rrbracket(s, d)$.

The reason is that specialisers normally just sort computations by p into *static operations*: those done during specialisation, and *dynamic operations*: those done by the residual program, for which code is generated. Most specialisers do not rearrange the order in which computation is performed, or invent new operations not in p , so residual computations can be embedded 1-1 into original ones.

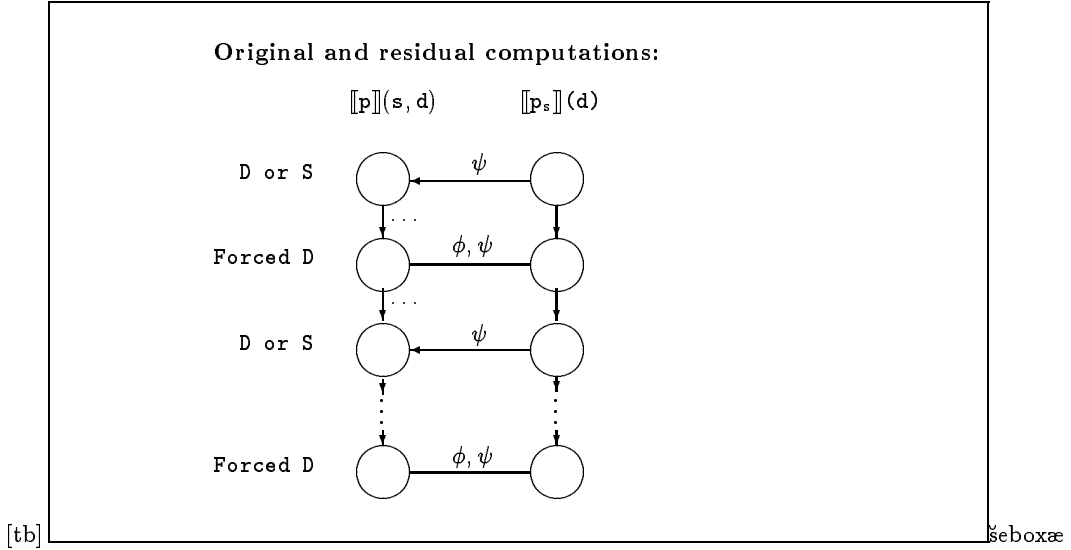
Analysis in the opposite direction: some operations done in the computation of $\llbracket p \rrbracket(s, d)$ can be forced to be residual, because they use parts of d , which is unavailable at specialisation time. Call such an operation in the $\llbracket p \rrbracket(s, d)$ computation *forced dynamic*. Since it cannot be done by *mix*, residual code must be executed to realise its effect.

It holds for most partial evaluators (but not all) that there exists an order-preserving *surjective order-preserving mapping* ϕ from

- the forced dynamic operations in the computation of $\llbracket p \rrbracket(s, d)$, onto
- corresponding operations in the computation of $\llbracket p_s \rrbracket(d)$.

Why most specialisers give at most linear speedup. Let $time-force_p(s, d)$ be the number of forced dynamic operations done while computing $\llbracket p \rrbracket(s, d)$. Assuming that all of these *must* be performed by the specialised program, we get a lower bound on its running time: $time-force_p(s, d) \leq time_{e_p}(d)$.

All other p operations can be performed without knowing d : they depend only on s . The maximum length of any sequence of such operations (assuming of course that specialisation terminates) is thus some function $f(s)$ of the static program input. Thus at most $f(s)$ static operations occur between any two forced dynamic operations, yielding $time_p(s, d) \leq (1 + f(s)) \cdot time-force_p(s, d)$.



Combining these, we see that $time_p(s, d) \leq (1 + f(s)) \cdot time_{p_s}(d)$, so

$$time_p(s, d) / time_{p_s}(d) = \text{Speedup} \leq 1 + f(s)$$

This second observation explains the fact that, in general, partial evaluators yield at most linear speedup, as illustrated in Figure 8.1.

8.2 Ways to break the linear speedup barrier.

The argument just given rests on the existence of an order-preserving “onto” mapping of forced operations in original computations to residual ones. This is not always the case, and specialisers working differently can obtain superlinear speedup. Three cases follow:

Dead code elimination: if `mix` can examine its residual program and discover “semantically dead” code whose execution does not influence the program’s final results, such code can be removed. If it contains dynamic operations, then there will be dynamic $\llbracket p \rrbracket(s, d)$ operations not corresponding to any $\llbracket p_s \rrbracket(d)$ operations at all. In this situation, between the two `p` operations corresponding to two `ps` operations there can occur a number of `p` operations *depending on dynamic input*, which were removed by the dead code elimination technique.

A **logic programming analogy** is the specialiser-time detection of branches in the SLD computation tree which are guaranteed to fail (and may depend on dynamic data). This is much more significant than in functional programming, in which time-consuming sequences of useless operations are most likely a sign of bad programming — whereas the ability to prune useless branches of search trees is part of the essence of logic programming.

Removing repeated subcomputations is another way to achieve super-linear speedup, and somehow seems less trivial than just eliding useless computations. The program transformation literature abounds with large speedups achieved by using memoisation to avoid recomputation of results, the standard one being to go from the exponential-time Fibonacci function to a linear-time (or even logarithmic-time) one. Integrating such more powerful transformations into fully automated partial evaluators remains a challenging problem, and in my opinion one well worth attempting.

9 On code explosion

Big problems can arise even when everything not classified as dynamic is of bounded static variation!!

9.1 Dead static parameters.

A parameter is *semantically dead* at a program point if changes to its value cannot affect the output of the current program. Syntactic approximations to this property are widely used in optimising compilers, especially to minimise register usage.

Now the size of specialised program p_s critically depends on the number of its *specialised program points*, each of form $\ell_{(s_1, \dots, s_k)}$ where ℓ is one of p 's program points, and (s_1, \dots, s_k) are the values of static parameters whose scope includes point ℓ .

Clearly dead static parameters will merely increase the size of specialised program p_s , without changing its computation in any way – so removal of dead static parameters can reduce its size. Our first experience with this problem led to a residual program 500 times smaller, for a simple imperative language!

9.2 Synchronicity in static parameter variation.

A good example: binary search. The following imperative program p performs binary search in ordered (growing) table T_0, \dots, T_{2^n-1} , with initial call $\text{Find}(T, 0, m, x)$ and $m = 2^{n-1}$. Parameter x is the target to be found, i points to a position in the table, and delta is the size of the portion $i \dots i + \text{delta}$ of the table currently being searched.

We begin by assuming that delta is classified as static and that i is dynamic. Using a term from [6], the program is *weakly oblivious*, since the comparison with x does not affect the value assigned to delta .

```
Find(T, i, delta, x) =
  Loop: if delta = 0 then
    if x = T[i] then return(i) else return(NOTFOUND);
  if x >= T[i+delta] then i := i + delta;
  delta := delta/2;
  goto Loop
```


Specialising with respect to static `delta = m = 4` and dynamic `i` gives program `p4`:

```

if x >= T[i+4] then i := i+4;
if x >= T[i+2] then i := i+2;
if x >= T[i+1] then i := i+1;
if x = T[i] then return(i) else return(NOTFOUND)

```

In general `pn` runs in time $O(\log(n))$, and with a better constant coefficient than the original general program. Moreover, it has size $O(\log(n))$ — acceptable for all but extremely large tables.

A bad example: binary search. To illustrate the problems that can occur, consider the same binary search program above with `n` static. One may certainly classify both parameters `delta` and `i` as static, since both range over $0, 1, \dots, n-1$. The resulting program is, however, not oblivious since the test on `x` affects the value of static `i`. Said differently, parameters `i` and `delta` are *independently varying static parameters*.

Specialisation with respect to static `delta = 4` and `i = 0` now gives the program in Figure 9.1.

```

if x >= T[4] then
  if x >= T[6] then
    if x >= T[7] then
      [if x = T[7] then return(7) else return(NOTFOUND)] else
      [if x = T[6] then return(6) else return(NOTFOUND)] else
    if x >= T[5] then
      [if x = T[5] then return(5) else return(NOTFOUND)]
      [if x = T[4] then return(4) else return(NOTFOUND)] else
  if x >= T[2] then
    if x >= T[3] then
      [if x = T[3] then return(3) else return(NOTFOUND)] else
      [if x = T[2] then return(2) else return(NOTFOUND)] else
    if x >= T[1] then
      [if x = T[1] then return(1) else return(NOTFOUND)]
      [if x = T[0] then return(0) else return(NOTFOUND)]

```

[tb]

Seboxæ

The specialised program again runs in time $O(\log(n))$, and with a somewhat better constant coefficient than above. On the other hand it has size $O(n)$ — exponentially larger than the weakly oblivious version! This is unacceptably large, except for rather small tables. Further, experiments show its running time to be not much faster than those of the previous much more compact specialisation.

9.3 Some consequences of binding-time improvements

Binding-time improvements can lead to unexpectedly slow target code. Two typical examples follow.

Interpretation of a program with procedure calls is straightforward if no regard is taken of static-dynamic separation, e.g. a stack containing both source program text fragments and runtime values may be used. A binding-time improvement can, with some work, split this stack into separated static and dynamic parts. A complication is that the stack depth is dynamic, but “the trick” referred to in [6] can be used to keep the topmost (current) name part of the stack frame static.

Target programs produced by specialising such an interpreter may be slower than expected when doing procedure *returns*. The point is that a procedure return is traditionally compiled into an indirect goto. If this is not available, or if the partial evaluator cannot handle it (and few can!), the one-instruction “return jump” will be translated into a series of tests and control transfers, one to each possible return point.

An exactly analogous problem occurs for the lambda calculus: semicompositionality can be ensured by using closures and “the trick.” But the consequence is that every function call will be translated into a series of tests and calls.

9.4 An unfortunate programming style.

The interpreter sketch of Figure 9.2 essentially implements a “big-step” semantics for a tiny imperative language. Nonetheless it does not specialise well (though specialisation does in fact terminate).

This uses a quite natural idea: if a variable has not already been bound in the store, then it is in effect added, with initial value zero. While it creates no problems for execution, this apparently quite innocent trick can cause catastrophically large target programs to be generated when specialising the interpreter to a source program. The reason is that *mix* must take account of *all execution possibilities*.

For an example, consider a source program of the following form:

```
IF test1 THEN X1 := X1 + 1;
IF test2 THEN X2 := X2 + 1;
...
IF testn THEN Xn := Xn + 1;
```

After *test1* at specialisation time, *mix* will have to allow for either branch to have been taken, so the resulting store = (*ns*, *vs*) could either have *X1* present or absent. After the second test, 4 possibilities must be accounted for. As a consequence, specialisation will generate a target program whose size is exponential in *n*.

The problem is easily fixed, by revising the interpreter to first scan the entire source program, collecting an initial store in which every variable is bound to zero. The result will yield a target program whose size is proportional to that

```

Run(pgm, input) = Exec(Cmd1, 'ns1. (list input), pgm) where
    Cmd1 = lookbody(first_fcn(pgm), pgm)
    ns1 = looknames(first_fcn(pgm), pgm)

; Exec : Command x Names x Values x Program -> Names x Values

Exec(Cmd, ns, vs, pgm) = case e of

    '(C1 ; C2)      : Exec(C2, ns, Exec(C1, ns, vs, pgm), pgm)

    '(X := X + 1) : if member(X, ns)
        then (ns,          update(X, 1+lookpar(X,ns,vs), ns, vs))
        else (cons(X, ns), cons(1,vs))

    '(IF e THEN C1 ELSE C2) :
        if eval(e0, ns, vs, pgm) then Exec(C1, ns, vs, pgm)
        else Exec(C2, ns, vs, pgm)

Eval(Cmd, ns, vs, pgm) = ...

```

[tb]

Seboxæ

of the source program. A further optimisation is that the type of `Exec` can be merely

```
Exec: Command x Names x Values x Program -> Values
```

with correspondingly smaller target code.

References

1. A. Bondorf and O. Danvy, 'Automatic autoprojection of recursive equations with global variables and abstract data types', *Science of Computer Programming*, 16:151–195, 1991.
2. C. Consel, 'New insights into partial evaluation: The Schism experiment', in H. Ganzinger (ed.), *ESOP '88, 2nd European Symposium on Programming, Nancy, France, March 1988 (Lecture Notes in Computer Science, vol. 300)*, pp. 236–246, Berlin: Springer-Verlag, 1988.
3. A. De Niel, E. Bevers, and K. De Vlamincx, 'Partial evaluation of polymorphically typed functional languages: The representation problem', in M. Billaud et al. (eds.), *Analyse Statique en Programmation Équationnelle, Fonctionnelle, et Logique, Bordeaux, France, Octobre 1991 (Bigre, vol. 74)*, pp. 90–97, Rennes: IRISA, 1991.
4. J. Gallagher: Specialization of logic programs. PEPM 93. pp. 88-98. ACM Press, 1993.
5. N.D. Jones, 'BTA Algorithms to Ensure Termination of Offline Partial Evaluation', submitted to *Second Andrei Ershov Memorial Conference, Akademgorodok, Russia*, June 1996.

6. Neil D. Jones, C.K. Gomard, P. Sestoft, "Partial Evaluation and Automatic Program Generation," Prentice Hall International Series in Computer Science, 1993.
7. T. Mogensen, 'Partially static structures in a self-applicable partial evaluator', in D. Bjørner, A.P. Ershov, and N.D. Jones (eds.), *Partial Evaluation and Mixed Computation*, pp. 325–347, Amsterdam: North-Holland, 1988.
8. T. Mogensen, 'Separating binding times in language specifications', in *Fourth International Conference on Functional Programming Languages and Computer Architecture, London, England, September 1989*, pp. 14–25, Reading, MA: Addison-Wesley, 1989.
9. D. Sahlin, 'The Mixtus approach to automatic partial evaluation of full Prolog', in S. Debray and M. Hermenegildo (eds.), *Logic Programming: Proceedings of the 1990 North American Conference, Austin, Texas, October 1990*, pp. 377–398, Cambridge, MA: MIT Press, 1990.
10. V.F. Turchin, 'The concept of a supercompiler', *ACM Transactions on Programming Languages and Systems*, 8(3):292–325, July 1986.
11. D. Weise, R. Conybeare, E. Ruf, and S. Seligman, 'Automatic online partial evaluation', in J. Hughes (ed.), *Functional Programming Languages and Computer Architecture, Cambridge, Massachusetts, August 1991 (Lecture Notes in Computer Science, vol. 523)*, pp. 165–191, Berlin: Springer-Verlag, 1991.

Contents

1 Introduction	1
1.1 The phenomena	1
1.2 Making implicit knowledge explicit	1
1.3 Dependence on particular specialisation algorithms	2
2 Concepts and notations	2
3 Assumptions in this paper	3
3.1 Restrictions in this discussion	3
3.2 Remarks on the restrictions	4
4 Bounded static variation: the key to termination of the specialiser	5
4.1 Pachinko, execution, and specialisation	5
4.2 Definition of bounded static variation.	6
5 Compositionality and semicompositionality of interpreters	6
6 Where do interpreters come from?	7
6.1 Interpreters derived from execution models	7
6.2 Interpreters derived from denotational semantics	7
6.3 Interpreters derived from operational semantics	9
7 Bounded or unbounded static variation?	11
7.1 Analysis of the instructional interpreter.	11
7.2 A small but problematic extension: procedures.	11
7.3 Analysis of the “big-step” interpreter	13
7.4 Dynamic binding.	13
8 On speedup	14
8.1 Embedding computations into original computations	14
8.2 Ways to break the linear speedup barrier.	15
9 On code explosion	16
9.1 Dead static parameters.	16
9.2 Synchronicity in static parameter variation.	16
9.3 Some consequences of binding-time improvements	18
9.4 An unfortunate programming style.	18