# Transformation by Interpreter Specialisation

Neil D. Jones

*DIKU, University of Copenhagen, Denmark*

**Abstract**

A program may be transformed by specialising an interpreter for the language in which it is written. Efficiency of the transformed program is determined by the efficiency of the interpreter's dynamic operations; the efficiency of its static operations is irrelevant, since all will be "specialised away". This approach is *automatic* (as automatic as the specialiser is); *general*, in that it applies to all of the interpreter's input programs; and *flexible*, in that a wide range of program transformations are achievable since the transformed program inherits the style of the interpreter.

The chief practical challenge is understanding cause and effect: How should one write the interpreter so that specialisation produces efficient transformed programs? The core of this paper is a series of examples, both positive and negative, showing how the way the interpreter is written can influence the removal of interpretational overhead, and thus the efficiency and size of the target programs obtained by specialisation.

In principle this method can speed up source programs by an arbitrary amount; the limit depends only on how efficiently the interpreter can be written. An example shows that a "memoising" interpreter can yield asymptotic speedup, transforming an exponential-time program into one running in polynomial time. A final analysis is made of the speed of the output programs in relation to the speed of the interpreter from which they are derived. It is argued that this relative speedup is linear, with a constant coefficient depending on the program being transformed.

## 1 Introduction

Program specialisation (also known as *partial evaluation* or *partial deduction*) is a source-to-source optimising program transformation. Suppose the inputs to program `p` have been divided into two classes: *static* and *dynamic*. A specialiser is given program `p` together with values `s` for its static inputs. It yields as output a *specialised program* $\mathtt{p}_\mathtt{s}$.

[1] This research was partially supported by the Danish *DART* and *PLI* projects.

Program $\texttt{p}_\texttt{s}$, when run on any dynamic inputs, will yield the same output that `p` would have yielded, if given *both* its static and dynamic inputs. Specialised program $\texttt{p}_\texttt{s}$ may be significantly faster than `p` if many of `p`'s computations that depend only on the input `s` have been pre-computed.

*Ideally* a partial evaluator is a black box: a program transformer able to extract nontrivial static computations whenever possible; that never fails to terminate; and always yields specialised programs of reasonable size and maximal efficiency, so all possible computations depending only on static data have been done. *Practically* speaking, program specialisers often fall short of this goal; they sometimes loop, sometimes pessimise, and can explode code size.

A program specialiser is analogous to a spirited horse: While impressive results can be obtained when used well, the user must know what he/she is doing. This paper's thesis is that *this knowledge can be communicated* to new users of program specialisers.

**The phenomena.** In my opinion 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* program specialiser difficulties; and one can *learn and communicate* programming style that leads to good specialisation, i.e., high speedups without explosion of code size.

This paper communicates this knowledge by a check-list of advice, illustrated by a series of concrete examples that concentrate on a broad and on the whole quite successful application area: achieving *program transformation* by specialising an interpreter with respect to a source program. Examples are both positive and negative, showing how the *way the interpreter is written* affects the efficiency and size of the transformed programs. After each, we summarise what was learned, and analyses the problems that arose.

**Overview of the article.** Section 2 defines basic concepts, shows by equational reasoning that programs may be transformed by interpreter specialisation, and identifies two measures (Type 1 and Type 2) of the speedups obtained by specialisation. The nature of the transformations that can be realised by the approach is discussed, followed by descriptions of some successful applications.

Two small examples begin Section 3, whose main content is a list of the assumptions we make about program specialisation. Section 4 contains a (very) simple example interpreter, together with an imperative source program, and the result of transforming it into functional form by specialisation.

Section 5 is the heart of the paper: how to obtain good results from interpreter specialisation, in the form of a list of points of advice. It begins with problems that must be overcome to use the methodology. There are, unavoidably, references to concepts developed in later sections. Finally, the question of where relevant interpreters originate is discussed.

Section 6 focuses on the problem of ensuring that transformation will terminate. Section 7 contains several interpreter examples, after which Section 8 analyses their specialisation. Section 9 addresses the problem of code explosion. One cause is analysed: static parameters that vary independently. Two examples are given of the problem, and work-arounds are described.

The next-to-last Section 10 investigates the question of how much speedup may be obtained in principle by program specialisation. Two speedup types are identified. It is shown that Type 1 speedup can be arbitrarily large, as it amounts to a change of algorithm, and an example displaying exponential speedup is given. Final, it is shown that Type 2 speedup is linearly bounded, with a constant coefficient depending on the program being transformed.

**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 of thought or of programming style (I speak from experience with students). A major point of this article is to make manifest some of the *implicit knowledge* that we have acquired in our field ([14,24,27] and the PEPM conference series), so newcomers can avoid encountering and having to figure out how to solve the same problems "the hard way".

## 2 Concepts and notations from program specialisation

### *2.1 Main concepts*

We use familiar notations (e.g., see [27]). Data set $D$ is a set containing all first-order values, including *all program texts*. Inputs may be combined in pairs, so "$(\mathtt{d}, \mathtt{e})$" is in $D$ if both `d` and `e` are in $D$. For concreteness the reader may think of $D$ as being Lisp/Scheme lists, and programs as being written in Scheme. However the points below are not at all language-dependent.

The partial function $[\![\mathtt{p}]\!] : D \to D \cup \{\bot\}$ is the *semantic function* of program `p`: If `p` terminates then $[\![\mathtt{p}]\!](\mathtt{d})$ is in $D$, and if `p` fails to terminate on `d`, then

$[\![\mathtt{p}]\!](\mathtt{d}) = \bot$. We will assume the underlying implementation language is defined by an operational semantics, not specified here. Computations are given by *sequences* (for imperative languages) or, more generally, by *computation trees*.

**Program running times.** We assume that program run time is proportional to the size of the computation tree deducing $[\![\mathtt{p}]\!](\mathtt{d}) = \mathtt{e}$. (This will be relevant in the discussion of linear speedup). Notation: The time taken to compute $[\![\mathtt{p}]\!](\mathtt{d})$ is written $time_{\mathtt{p}}(\mathtt{d})$ (a value in $N \cup \infty$, equal to $\infty$ iff p fails to terminate on d).

*Other languages.* If unspecified, we assume that a standard implementation language **L** is intended. If another language **S** is used, we add a superscript **S** to the semantic and running time functions, e.g., $[\![\mathtt{p}]\!]^{\mathbf{S}}(\mathtt{d})$ and $time^{\mathbf{S}}_{\mathtt{p}}(\mathtt{d})$.

### *2.1.1 Program specialisation.*

Given program p and a "static" input value $\mathtt{s} \in D$, $\mathtt{p}_\mathtt{s}$ is a *result of specialising* p *to* s if for every "dynamic" input value $\mathtt{d} \in D$

$$[\![\mathtt{p}_\mathtt{s}]\!](\mathtt{d}) \doteq [\![\mathtt{p}]\!](\mathtt{s},\mathtt{d}) \tag{1}$$

We call $\mathtt{p}_\mathtt{s}$ a *specialised program* for p with respect to s. [2]

A *partial evaluator* (or *program specialiser*) is a program spec such that for every program p and "static" input $\mathtt{s} \in D$, $[\![\mathtt{spec}]\!](\mathtt{p},\mathtt{s})$ is a result $\mathtt{p}_\mathtt{s}$ of specialising p to s. Correctness is specified by the equation [3]

$$[\![\mathtt{p}]\!](\mathtt{s},\mathtt{d}) \doteq [\![[\![\mathtt{spec}]\!](\mathtt{p},\mathtt{s})]\!](\mathtt{d}) \tag{2}$$

A trivial spec is easy to build by "freezing" the static input s (Kleene's *s-m-n* Theorem of the 1930s did specialisation in this way). The practical goal is to make $\mathtt{p}_\mathtt{s} = [\![\mathtt{spec}]\!](\mathtt{p},\mathtt{s})$ *as efficient as possible*, by performing many of p's computations — ideally, all of those that depend on s alone.

A number of practical program specialisers exist. Systems involved in recent articles include *Tempo*, *Ecce*, *Logen* and *PGG* [13,31,30,48], and significant

[2] Symbol $\doteq$ is equality of partial values: $a \doteq b$ means either both sides are equal elements of $D$, or both sides are undefined. This definition is purely *extensional*, ignoring "intensional" features such as program efficiency or size. A more intensional view is often taken in logic programming [31,33].

[3] The right side of (2) can be undefined two ways: Either $[\![\mathtt{spec}]\!](\mathtt{p},\mathtt{s}) = \bot$ if the specialiser loops; or $[\![\mathtt{spec}]\!](\mathtt{p},\mathtt{s}) = \mathtt{q}$ and $[\![\mathtt{q}]\!](\mathtt{d}) = \bot$ if the specialised program loops.

earlier work has been reported concerning the systems *C-mix*, *Similix* and *Mixtus* [21,29,41].

### *2.1.2 Interpretation and its overhead*

Program `interp` is an *interpreter for language* **S** if for all **S**-programs `source` and data $\mathtt{d} \in D$

$$[\![\mathtt{interp}]\!](\mathtt{source}, \mathtt{d}) \doteq [\![\mathtt{source}]\!]^{\mathbf{S}}(\mathtt{d}) \tag{3}$$

*Interpretational overhead.* Interpreters are reputed to be slow. This is certainly true if `interp` slavishly imitates the operations of `source`; $time_{\mathtt{interp}}(\mathtt{source}, \mathtt{d})$ may be 10 times larger than $time^{\mathbf{S}}_{\mathtt{source}}(\mathtt{d})$.

On the other hand, `interp` may be written cleverly using, for example, the *memoisation* device to save time when program `source` would solve the same problem repeatedly. Such an interpreter may in fact run *faster* than the program being interpreted.

### *2.1.3 Compilation and specialisation*

A *compiler* transforms **S**-program `source` into **L**-program `target` such that

$$[\![\mathtt{target}]\!](\mathtt{d}) \doteq [\![\mathtt{source}]\!]^{\mathbf{S}}(\mathtt{d}) \tag{4}$$

for any data $\mathtt{d} \in D$. The *first Futamura projection* [17] is the equation

$$\mathtt{target} \stackrel{def}{=} [\![\mathtt{spec}]\!](\mathtt{interp}, \mathtt{source}) \tag{5}$$

It states that one may *compile by specialising an interpreter*, since

$$\begin{array}{lcll} [\![\mathtt{source}]\!]^{\mathbf{S}}(\mathtt{d}) & \doteq & [\![\mathtt{interp}]\!](\mathtt{source}, \mathtt{d}) & \text{by (3)} \\ & \doteq & [\![[\![\mathtt{spec}]\!](\mathtt{interp}, \mathtt{source})]\!](\mathtt{d}) & \text{by (2)} \\ & = & [\![\mathtt{target}]\!](\mathtt{d}) & \text{by (5)} \end{array}$$

### *2.1.4 Type 1 speedup.*

The purpose of program specialisation is to make programs run faster. *Type 1 speedup* compares the speed of the two sides of Equation (1).[4] The Type 1

[4] Type 1 speedup ignores the time $time_{\mathtt{spec}}(\mathtt{p}, \mathtt{s})$ required to specialise `p` to `s`. This is reasonable if $[\![\mathtt{p}]\!](\mathtt{s}, \mathtt{d})$ is often computed with the same `s` value but with varying

speedup realised by specialising program p to static input s is the ratio:

$$\text{speedup}_{\mathtt{s}}(\mathtt{d}) = \frac{time_{\mathtt{p}}(\mathtt{s},\mathtt{d})}{time_{\mathtt{p_s}}(\mathtt{d})} \tag{6}$$

The Type 1 speedup is 1.7, for instance, if the specialised program runs 1.7 times as fast as the original program. (If less than 1, it might better be called a slowdown). For each static input s, the Type 1 speedup is a function of d.

### *2.1.5 Type 2 speedup.*

The Type 1 speedup for a given interpreter interp would be

$$\text{speedup}_{\mathtt{source}}(\mathtt{d}) = \frac{time_{\mathtt{interp}}(\mathtt{source},\mathtt{d})}{time_{\mathtt{target}}(\mathtt{d})}$$

If source language **S** has a natural time measure it may be more naural to compare the speed of the target program to the speed of the source program it was obtained from, giving the the *Type 2 speedup*:

$$\text{speedup}(\mathtt{d}) = \frac{time^{\mathbf{S}}_{\mathtt{source}}(\mathtt{d})}{time_{\mathtt{target}}(\mathtt{d})} \tag{7}$$

We will see in Section 10 that for normal specialisers Type 1 speedup is at most a linear function of d, with a coefficient determined by the value of s. On the other hand Type 2 speedup can be arbitrarily large, shown by an example whose speedup is exponential.

## *2.2 Discussion: What transformations can be realised by specialisation?*

Suppose target = [[spec]](interp, source). For all specialisers I know:

(1) Program target inherits the *algorithm* of program source.
(2) Program target inherits the *programming style* of interpreter interp.

The reason for Point 1 is that the interpreter interp faithfully executes the operations specified by program source, so its specialised version target will execute *the same operations in the same order.*

The reason for Point 2 is that program target consists of specialised code taken from the interpreter interp: the operations it could not execute using only the values of its static input source. Some consequences are discussed in [37,46]. Further examples of Point 2: We can expect that

d values – natural if p is an interpreter and s is a source program.

- If `interp` is written in a functional language, then `target` will also be functional, regardless of how `source` was written;
- If `interp` is written in tail-recursive style, then `target` will also be tail-recursive;
- If `interp` is written in CPS (continuation-passing style), then `target` will also be in CPS;
- If `interp` uses memoisation to implement its function calls, then the corresponding specialised calls in `target` will also use memoisation.

A general-purpose program transformer can thus be built by programming a self-interpreter *in a style appropriate to the desired transformation.* Further, if some parts of the program should be handled differently than others (e.g., if some but not all calls should be memoised), this can be accomplished by adding *program annotations* to `source`, and programming `interp` to treat parts of `source` in accordance with their annotations.

**The main challenges to overcome** to use the approach in practice are:

- to write `interp` so it realises, while executing `source`, the optimisations the user wishes to obtain (style change, memoisation, instrumentation, etc.); and
- to write `interp` so as many operations as possible are given static binding times, so `target` will be as efficient as possible.

**The goal of this article.** Our goal is pragmatically to understand *cause* and *effect* in the process of realising a program transformation by interpreter specialisation.

### *2.3 Applications of interpreter specialisation.*

**Program transformation.** If **S** = **L** then `interp` is a self-interpreter, and the first Futamura projection effects a transformation from **L**-programs to **L**-programs. This approach has been widely used in the partial evaluation community, e.g., [27,46,49,18,37,23,28]. In this case, the Type 2 speedup ratio is the natural measure of the benefit of specialisation.

**Language extension.** Self-interpreters have long been popular in the Lisp, Scheme and Prolog communities because the *source language can be extended* , e.g., to allow new language constructions, simply by extendin the interpreter.

However this flexibility comes with a cost: Programs must be executed interpretively, instead of being compiled into more efficient target code.

Specialisation allows the best of both worlds. Partially evaluating an extended self-interpreter with respect to an extended source program `p'` has the effect of translating `p'` into an equivalent program `p` in the base language, which can then be compiled by an existing compiler. The net effect is thus to remove all interpretational overhead.

**Debugging.** A self-interpreter can be augmented with *instrumentation code* to keep a log of value and control flow, to check for exceptional behavior, etc. to aid debugging. Although flexible, this approach is not ideal. Interpreters tend to be slow; and an interpreter's bookkeeping operations may change timing factors and so distort communication behaviour of a concurrent program.

Specialising an instrumented self-interpreter to a source program has the effect of *inserting instrumentation code into the body of the source program* [19]. One advantage over interpretation is faster execution. Another advantage, particularly useful when debugging multithreaded programs, is that the control flow in an instrumented program is closer to that of the original source program than that of an interpreter.

**Aspects and reflection.** Kiczales, Asai and others have used this approach to implement quite nontrivial extensions of the base language's semantics. Kiczales applied specialisation to an interpreter for an *aspect-oriented language* [34]. The effect of specialisation was automatically to achieve the effect of "weaving," a rather tricky process in most aspect-oriented languages. Further, Asai and others have used this approach to compile a *reflective language* by specializing a customized meta-circular interpreter [4].

**Domain-specific languages.** If interpreter `interp` implements a special-purpose language **S**, by the first Futamura projection a specialiser can compile **S**-programs into the specialiser's output language. In this case **S**-programs may have no "natural" running times, so Type 1 speedup measures the benefit gained by specialisation. Articles on domain-specific languages include [38,39,47,50].

## 3 Assumptions about the specialisation algorithm

The example series to be given can lead to remarks of the form "yes, but I can solve that problem using feature XXX of my program specialiser". A consequence is that the advice to be given seems, at least to some extent, relative to which *type of specialisation algorithm* is used. Accounting online for such remarks, with responses ("even so, ...") could easily lead to a protracted discussion whose line would be hard to follow. Worse, it would make it hard at the end of the paper to say what really had been accomplished.

To avoid fruitless digressions, I now carefully detail the rather conservative assumptions about the specialisation algorithms used in this paper. Once readers have followed my reasoning under these assumptions, they are encouraged to devise better solutions. We begin with two examples, and then discuss specialisation techniques.

**What partial evaluation can accomplish.** The canonical first partial evaluation example is the "power" program, computing $x^n$ and specialised to a known static $n$ value. The program, and its specialisation to $n = 5$:

```
power(n,x) =  if n=0 then 1 else x * power(n-1,x)
power5(x)  =  x * x * x * x * x * 1
```

From a program transformation viewpoint, specialisation has occurred by applying a single *function definition*, some *unfold* operations, *constant propagation*. (The equality $x * 1 = x$ could also be applied to remove "`*1`").

A less trivial example program computing Ackermann's function illustrates some technical points. The program `p` to be specialised:

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

Specialisation to static `m=2` and dynamic `n` yields this program $\mathtt{p}_2$:

```
ack2(n)   =  if n=0 then ack1(1) else ack1(ack2(n-1))
ack1(n)   =  if n=0 then ack0(1) else ack0(ack1(n-1))
ack0(n)   =  n+1
```

Program $\mathtt{p}_2$ performs about half as many base operations (`+`,`-`,`=`,...) as program `p`. Specialisation occurred by creating new *function definitions* `ack2`, `ack0`, `ack1`, and *folds* to produce calls to the newly defined functions.

**How these examples can be specialised.** In the Power example the binding-time analysis (explained below) classifies `n` as static and `x` as dynamic, and *annotates* all the program's operations as follows:

```
power(n^s, x^d) = if^s n =^s 0 then^s 1^d else^s x *^d power^s(n -^s 1, x)
```

The statically annotated operations involving `n` will be computed at specialisation time. The recursive call to `power` is also marked with $^s$ for static, now meaning "to be unfolded". The specialised program contains only `*`, `x` and `1`.

The Ackermann example in binding-time annotated form is:

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

The `if`-test on `m` is marked static, to be decided at specialisation time; but specialised code will be generated for the test on `n`. The recursive calls are all marked as "not to be unfolded". This causes the calls to `ack2`, `ack1`, and `ack0` to be generated for the static input `m=2`.

### Assumptions about the specialisation algorithm.

(1) The programs to be specialised are assumed expressed in an *untyped first-order functional* language (side-effect free), using an informal syntax. However the *languages they interpret* may be functional, imperative, higher-order, logic, etc.

(2) *Offline* specialisation is assumed: Before the value of static input `s` is known, an automatic *binding-time analysis* is performed. This phase annotates all of program `p`'s function parameters, operations and calls as
- **static**: Compute/perform/unfold at specialisation time; or
- **dynamic**: Generate code to be executed at run time.

(3) We assume the binding-time analysis is *monovariant*: Each function parameter is always treated as static, or always as dynamic. (A polyvariant binding-time analysis, or an online specialiser, could yield better results on the Ackermann program by unfolding the calls `ack1(1)` and `ack0(1)` to yield respectively 3 and 2.)

(4) *Call unfolding*: A call will be unfolded at specialisation time if it will not cause a recursive call in the specialised program. [5]

[5] A common strategy in more detail: Unfold a source call that is non-recursive or one that causes a static parameter to decrease; but avoid unfolding if it would cause duplication of specialised code. Most specialisers allow a "pragma" to ensure that specialisation terminates: Any call may be marked by hand as "must not unfold".

(5) *Program point specialisation* is assumed: Each control point in the specialised program corresponds to a pair ($\ell$, *static-store*), where $\ell$ is a control point in the original program and *static-store* is a tuple of values of the program's static variables at point $\ell$.

The original Ackermann program has only one control point $\ell$: entry to the function `ack`. The specialised program's three control points `ack2`, `ack1`, `ack0` correspond to $(\texttt{ack}, \texttt{m} = 2), (\texttt{ack}, \texttt{m} = 1), (\texttt{ack}, \texttt{m} = 0)$.

(6) No global optimisations are done, e.g. the classical *dead code elimination*, or elimination of *common subexpressions*, especially ones containing function calls [6].

## 4 A simple interpreter example

The trivial imperative language `Norma` is described in Figure 1. During `Norma` program execution, input and the current values of registers `x`, `y` are represented as *unary or base 1 numbers* so `x` = 3 is represented by the length-3 list `(1 1 1)`. Labels are represented by the same device: The final `(GOTO 1 1)` transfers to instruction 2 (the test `ZERO-X?...`).

```
;; A Norma program has two numeric registers, x and y.
;; Initially x = program input and y = 0. Output = y's
;; final value.  Instruction set: jumps (unconditional
;; and conditional) and increments/decrements for x, y.
;; Program syntax:
;;
;; pgm   ::= ( instr* )
;; instr ::= (INC-X) | (DEC-X) | (INC-Y) | (DEC-Y)
;;         |  (ZERO-X? addr) |(ZERO-Y? addr) | (GOTO addr)
;; addr  ::= 1*

;; Data: a NORMA program ``source'' to compute 2 * x + 2.
  ((INC-Y)
   (INC-Y)
   (ZERO-X? 1 1 1 1 1 1 1)
   (INC-Y)
   (INC-Y)
   (DEC-X)
   (GOTO 1 1))
```

Fig. 1. Norma program syntax and example source program.

[6] Logic programming analogs: removal of code that is certain to fail, or removing duplicated goals.

```
execute(prog, x) = run( prog, prog, x, '() )

run(pgtail, prog, x, y) =
  if atom? pgtail then y else  ;; answer = y if execution
  let instr =  car(pgtail),    ;; done, else
      rest  =  cdr(pgtail) in  ;; dispatch on syntax
  let op    =  car(instr),
      arg   =  cdr(instr) in
   case op of                                ;; operator:
    'INC-X: run(rest, prog, cons('1,x), y)  ;; increment x
    'DEC-X: run(rest, prog, cdr x, y)       ;; decrement x
    'ZERO-X?: if pair? x                    ;; jump if x=0
              then  run(rest, prog, x, y)
              else  run(jump(prog,arg), prog, x, y)
    'INC-Y: run(rest, prog, x, cons('1,y))  ;; increment y
    'DEC-Y: run(rest, prog, x, cdr y)       ;; decrement y
    'ZERO-Y?:  if pair? y                   ;; jump if y=0
               then run(rest, prog, x, y)
               else run(jump(prog,arg), prog, x, y))
    'GOTO: run(jump(prog,arg), prog, x, y) ;; goto jump
     else  'ERROR-bad-instruction))))   ;; bad instruction

jump(prog, dest) = ;; find instruction "dest"
  if null? dest then  prog
  else  jump(cdr prog, cdr dest)
```

Fig. 2. Norma interpreter `Norma-int`.

**How interpreter `Norma-int` of Figure 2 works:** Given the program and the initial value of `x`, function `execute` calls `run`. In `run(pgtail, prog, x, y)`, parameters `x` and `y` are the current values of the two registers. The call from `run` thus sets `x` to the outside input and `y` to 0, represented by the empty list. Parameter `prog` always equals the `Norma` program being interpreted. The current "control point" is represented by a suffix of `prog` called `pgtail`. Its first component is the next instruction to be executed. Thus the initial call to `run` has `pgtail` = `prog`, indicating that execution begins with the first instruction in `prog`.

**Binding-time analysis:** The `execute` parameters are the interpreter program's inputs: a static input `prog` and a dynamic input `x`. Function `run` is thus initially called with static `pgtail` and `prog`, and dynamic `x`. For now we assume that `y` too is dynamic. All parts of the `run` body involving `x` or `y` are annotated as dynamic, e.g., `cdr x` or `cons('1,y)`. The remainder is annotated as static, including the central "dispatch on syntax". Further, both arguments

```
execute-1(x) = run-1(x, '(1 1))

run-1(x, y) = if pair? x
              then run-1(cdr x, cons('1,cons('1,y))) else y
```

Fig. 3. target = [[spec]](Norma−int, source) to compute $2x + 2$.

to jump are annotated as static.

**How specialisation of Norma-int to source proceeds:** The interpreter's statically annotated parts will be evaluated, and specialised program code will be generated for its dynamically annotated parts. In particular, any part involving x or y will be considered as dynamic.

The result target of specialising Norma-int to the source from Figure 1 is shown in Figure 3. It also computes $2 * x + 2$, but is a functional program. We see that target is about 10 times faster than Norma-int, if time is measured by the number of calls to base functions and to execute, run and jump.[7]

The example concretely illustrates two points made earlier:

(1) Program target inherits the *algorithm* of program source. (Here, to compute $2x + 2$ by repeated addition).
(2) Program target inherits the *programming style* of interpreter interp. (Here, imperative program source was translated to tail-recursive functional program target).

## 5 Advice on writing an interpreter for specialisation

**Problems to overcome** Two key problems must be overcome in order to get the full benefit of this approach in practice.

(1) NONHALTING: The specialiser may fail to terminate; and
(2) QUALITY: Program target may be too slow or too large.

**Points of advice.** The following points are listed here for convenient reference. Some parts, however, refer to concepts developed later.

[7] This would be less if GOTO were not implemented in such a simplistic way.

(1) Avoid the NONHALTING problem, i.e., nonterminating specialisation, by ensuring that every static parameter is of *bounded static variation* (explained in Section 6.3). If not, your specialiser is certain to fail to terminate when transforming some programs.
(2) If practical, write your interpreter by transliterating a big-step operational semantics (Section 7.3) or a denotational semantics (Section 7.4). This usually gives at least semicompositionality (Section 6.4). If you write your interpreter compositionally or semicompositionally then every syntactic argument (i.e., pieces of the interpreter's program input) will be of bounded static variation.
(3) If your interpreter is not based on a big-step operational or denotational semantics, there is no guarantee that all parameters depending only on static data will be of bounded static variation. You will thus have to do your own analyses and annotations to ensure that they are. Beware of static data values that can *grow unboundedly under dynamic control.* An example: the control stack $C$ of Section 8.1.2.
(4) To avoid problems of QUALITY, e.g., code explosion, ensure that at any program point:
   - There are no (or as few as possible) *independently varying static parameters* (Section 9.1).
   - There are no *dead static variables* (Section 9.3).
(5) If your interpreter is based on a small-step operational semantics, then apply the techniques sketched in Section 7.3.
(6) Don't expect *superlinear Type 1 speedup* (as a function of dynamic input size), unless the specialiser used is rather sophisticated (Section 10.2.1).
(7) If you wish to achieve superlinear speedup by using an unsophisticated specialiser, then *write your interpreter* to incorporate statically recognisable optimisation techniques such as those seen in Section 10.2.2 (memoisation, static detection of semantically dead code, etc.)

**Where do interpreters originate?** This question is inevitable if oneapproaches program transformation by specialising an interpreter. There are several natural answers:

(1) A definitional interpreter [40] is a program derived from an operationally oriented definition of the source language's semantics. [42,52]. Some variants: Big-step operational semantics; Small-step operational semantics; or Denotational semantics.
(2) An interpreter may originate in a runtime execution model, e.g., Java bytecode.
(3) An interpreter may originate in an ad hoc execution model, e.g., one used for instruction. Our first example, the Norma interpreter, is ad hoc.

## 6 Bounded static variation: the key to specialiser termination

### *6.1 The termination problem*

To understand the problem of specialiser termination, consider interpreter `Norma-int` with static program input `prog` and dynamic input `x`.

Clearly the `Norma-int` parameters `pgtail`, `dest` and `prog` can all be safely computed at specialisation time. Further, the recursive calls `run(rest,...)` or `jump(cdr prog,...)` decrease their static first parameters, so these calls may be unfolded without risk of nontermination. The effect is that all of the instruction dispatch code, and the entire auxiliary function `jump` can be "specialised away," i.e., done at specialisation time. This leaves less to do in `target`, and so gives a significant speedup. On the other hand, all operations involving `x` depend on data not available at specialisation time. These must be performed by the specialised program.

Parameter `y` is, however, problematic. It is initialised to 0 (the list `'()`) by the call from `execute`, and otherwise only tested, incremented or decremented. Thus *in principle* all `y` computations could be done at specialisation time. However in practice *this would be disastrous.* The reason: For the example source program computing $2 * x + 2$, the interpreter's `y` variable will take on the infinitely many possible values `()`, `(1)`, `(1 1)`,....

A solution is to *hand annotate* Figure 2, changing its first line to tell the specialiser that the `y` value should be considered as dynamic:

```
execute(prog, x) = run( prog, prog, x, generalize('()) )
```

More generally, specialisation can fail to nonterminate because of:

(1) An attempt to build an *infinitely large* command or expression; or
(2) An attempt to build a specialised program containing *infinitely many* program points (labels, defined functions, or procedures;) or
(3) An *infinite static loop* without generation of specialised code.

The basic cause is the same: The specialiser's *unfolding strategy* is the main cause of the one or the other misbehaviour. For the `Norma-int` example with the suggested unfolding strategy, problem 2 arises if `y` is considered as a static parameter. Such nontermination misbehaviour makes a specialiser unsuitable 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 have been seen to nonterminating specialisers, and each view can be (and has been) both attacked and defended. In functional or imperative specialisers, infinite static loops are often seen as the user's fault, so there is no need for the specialiser to account for them. The other two problems are usually handled by hand annotations.

Logic programming most often requires that the specialiser *always terminate*, even if this involves high specialisation times (e.g., online tests for homeomorphic embedding). A reason is that the Prolog's "negation as finite failure" semantics implies that changing termination properties can change semantics, and even answer sets.

Recent work on functional program specialisation shows how to carry out a binding-time analysis that is conservative enough to guarantee that specialisation always terminates, and still gives good results when specialising interpreters [26].

### *6.2 Pachinko, execution, and specialisation*

Normal execution is given complete input data and deterministically performs a single execution. On the other hand a specialiser is given *incomplete* (static) input data; and it must account for *all possible computations* that could arise for any dynamic input values.

An analogy: The popular Japanese "Pachinko" entertainment involves steel balls that fall through a lattice of pins. Deterministic execution corresponds to dropping only one ball (the program input), which thus traverses only one trajectory. On the other hand, specialisation to static input `s` corresponds to dropping an infinite set of balls all at once — those that share `s` as given static input, but have *any possible* dynamic input value. Reasoning about specialisation, e.g., specialised program finiteness or efficiency, thus requires reasoning about sets of computations that are usually infinite.

**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 writen as $\mathtt{f}^{(i)}$.

A *state* is a pair $(\mathtt{f}, \overline{\alpha})$ where $\overline{\alpha} \in V^{\mathrm{arity}(\mathtt{f})}$. Notation: $\overline{\alpha}_i$ is the $i$th component of parameter tuple $\overline{\alpha} \in V^{\mathrm{arity}(\mathtt{f})}$. Thus $\overline{\alpha}_i$ is the value of $\mathtt{f}^{(i)}$ in state $(\mathtt{f}, \overline{\alpha})$. State $(\mathtt{f}, \overline{\alpha})$ can *reach* state $(\mathtt{g}, \overline{\beta})$, written $(\mathtt{f}, \overline{\alpha}) \rightarrow (\mathtt{g}, \overline{\beta})$, if computation of `f` on parameter values $\overline{\alpha}$ requires the value of `g` on parameter values $\overline{\beta}$.

Example: $(\mathtt{f}, 5) \rightarrow (\mathtt{f}, 4)$ and $(\mathtt{f}, 5) \rightarrow (\mathtt{f}, 3)$ in program:

```
f(n) = if n < 2 then n else f(n-1) + f(n-2)
```

### *6.3 Definition of bounded static variation*

Let ';' be the *tuple concatenation operator*, suppose `f1` is the initial (entry) function of program `p`, and let $s, d \in N$, where $s + d = \text{arity}(\texttt{f1})$. We assume (for simplicity) that the first $s$ parameters of `f1` are static, and the remaining $d$ parameters are dynamic. The set of *statically reachable* states $\text{SR}(\overline{\sigma})$ consists of all states $(\texttt{f}, \overline{\alpha})$ reachable in 0, 1, or more steps from some initial call with $\overline{\sigma} \in V^s$ as static program input, and an arbitrary dynamic input $\overline{\delta}$:

$$\text{SR}(\overline{\sigma}) = \{(\texttt{f}, \overline{\alpha}) \mid \exists \overline{\delta} \in V^d \,:\, (\texttt{f1}, \overline{\sigma}; \overline{\delta}) \rightarrow^* (\texttt{f}, \overline{\alpha})\}$$

The $i$th parameter $\texttt{f}^{(i)}$ of `f` is of *bounded static variation*, abbreviated to BSV, iff for all $\overline{\sigma} \in V^s$, the following set is finite:

$$\text{StatVar}(\texttt{f}^{(i)}, \overline{\sigma}) \stackrel{def}{=} \{\overline{\alpha}_i \mid (\texttt{f}, \overline{\alpha}) \in \text{SR}(\overline{\sigma})\}$$

This captures the intuitive notion that `f`'s $i$th parameter only varies finitely during all computations whose inputs share a fixed static program input $\overline{\sigma}$, but may have arbitrary dynamic input values. In the interpreter `Norma-int`, for example, parameters `prog`, `pgtail` and `dest` are of BSV, while neither `x` nor `y` are of BSV.

**Theorem** If every parameter classified as "static" is of bounded static variation, then specialisation can be guaranteed to terminate.

Most BSV parameters are directly computable from the static program inputs, but exceptions do occur. An example expression of BSV, even if `x` is dynamic:

```
if x mod 2 = 0 then 'EVEN else 'ODD
```

### *6.4 Compositionality and semicompositionality of interpreters*

Henceforth we call the interpreted source program `source`. Classify each function parameter of an interpreter as *syntactic* if its values range only over phrases (e.g., names, expressions or commands) from the source language, and otherwise as *nonsyntactic*.

The interpreter is defined to be **compositional** if for any function definition, the syntactic parameters of every recursive subcall are *proper substructures of the calling function's syntactic parameters.* Clearly in a computation by

a compositional interpreter the syntactic parameters in a function call are necessarily substructures of the interpreted program `source`, or interpreter constants – and so obviously of BSV.

The compositionality requirement ensures that the interpreter can be specialised with respect to its static input `source` by simple unfolding — a process *guaranteed to terminate* since the "substructure" relation is well-founded.

A weaker requirement, also guaranteeing BSV, is for the interpreter to be **semicompositional**, meaning that called parameters must be *substructures of the original source program* `source`, but need not decrease in every call. Semicompositionality can even allow syntactic parameters to grow; the only requirement is that their value set is limited to range over the set of all substructures of `source` (perhaps including the whole of `source` itself).

Some common instances of semicompositionality: The meaning of a **while** construct can be defined in terms of itself, and procedure calls may be elaborated by applying an execution function to the body of the called procedure. Both would violate strict compositionality.

Semicompositionality does not guarantee termination at run time; but it does guarantee finiteness of the set of specialised program points — so *termination of specialisation* can be ensured by a natural unfolding strategy: unfold, unless this function has been seen before with just these static parameter values.

## 7 Several example interpreters

### *7.1 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.

### *7.2 An instructional interpreter.*

The interpreter of Figure 4 has been used for instruction at DIKU, and derives from lecture notes used at Edinburgh. Computation is by a linear sequence

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

of states of form $(S, M, C)$. $S$ is a *computation* stack, $C$ is a *control* stack containing bits of commands and expressions that remain to be evaluated/executed, and *memory* $M : \{\mathtt{x}0, \mathtt{x}1, \ldots\} \to$ Value contains the values of variables $\mathtt{x}i$. ($C$ is in essence a materialisation in data form of the program's *continuation* from the current execution point).

**Rules for Expressions**

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

**Rules for Programs**

$$\begin{array}{lll}
\text{Null}: & \langle S, M, \mathtt{skip} \cdot C\rangle & \Rightarrow \langle S, M, C\rangle \\
\text{Assign}: & \langle S, M, (\mathtt{x}i := e) \cdot C\rangle & \Rightarrow \langle i \cdot S, M, e \cdot \mathbf{assign} \cdot C\rangle \\
\text{Seq}: & \langle S, M, (p_1\,;p_2) \cdot C\rangle & \Rightarrow \langle S, M, p_1 \cdot p_2 \cdot C\rangle \\
\text{Test}: & \langle S, M, (\mathtt{if}\ b\ \mathtt{then}\ p_1\ \mathtt{else}\ p_2) \cdot C\rangle & \Rightarrow \langle p_1 \cdot p_2 \cdot S, M, b \cdot \mathbf{if} \cdot C\rangle \\
\text{Loop}: & \langle S, M, (\mathtt{while}\ b\ \mathtt{do}\ p_1) \cdot C\rangle & \Rightarrow \langle b \cdot p_1 \cdot S, M, b \cdot \mathbf{while} \cdot C\rangle
\end{array}$$

**Rules for assign, if and while**

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

Fig. 4. Instructional interpreter `int-instruct`.

For readability we describe this interpreter (call it `int-instruct`) by a set of transition rules in Figure 4. Note that the $C$ component is neither compositional nor semicompositional. The underlines, e.g., $\underline{n}$, indicate numeric runtime values, and $M[\mathtt{x}i \mapsto \underline{n}]$ is a memory $M'$ identical to $M$, except that $M'(\mathtt{x}i) = \underline{n}$.

### *7.3 Interpreters derived from operational semantics*

*Operational semantics* are easier to program than denotational semantics, 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 may, for example, be of form

$$environment \vdash Exp \Rightarrow Value$$

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

$$environment \vdash Exp \Rightarrow Exp'$$

that 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 the judgement as root, and where each node of the tree is an instance of an inference rule.

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 $\lambda$-calculus.

### *7.3.1 An interpreter derived from a big-step operational semantics.*

The interpreter `int-big-step` of Figure 5 implements a "big-step" semantics. A big-step judgement *environment* $\vdash$ *Exp* $\Rightarrow$ *Value* is implemented as a function `Eval` constructing *Value* from arguments *environment* and *Exp*. The *environment* is represented by two values: a list `ns` of variable names, and a parallel list `vs` of variable values. The source program has only one input, but its internal functions may have any number of parameters, as in the interpretation of `(call f es)`.

### *7.3.2 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 reduction in the lambda calculus. This consists of a few notions of reduction, e.g., $\alpha, \beta, \eta$, and *context rules* stating that these can be applied when occuring inside other syntactic constructs. Problems with direct implementation include:

- *Nondeterminism*: One $\lambda$-expression may have many reducible redexes;
- *Nondirectionality of computation*: Rewriting may occur either from left to right or vice versa (conversion rather than reduction); and
- *Syntax rewriting*: A subject program's syntax is continually changed in ways that are statically unpredictable during interpreter specialisation.

```
; Run: Program x Program_value -> Output_value

Run(pgm, input) = Eval(e1, ns1, cons(input, 'nil), pgm)
              where f   = first_function(pgm)
                    e1  = getbody(f, pgm)
                    ns1 = getparams(f, pgm)

; Eval: Expression x Names x Values x Program -> Value

Eval(e, ns, vs, pgm) =    case e of
  constant            : constant
  'X                  : lookparam(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, vs1, pgm) where
                       e1  = getbody(f, pgm)
                       ns1 = getparams(f, pgm)
                       vs1 = Evlist(es, ns, vs, pgm)

Evlist(es, ns, vs, pgm) =    case es of
          'nil          : nil
          'cons(e1 es1) : cons(Eval(e1, ns, vs, pgm),
                               Evlist(es1, ns, vs, pgm))

lookparam(X, ns, vs) = case ns of
               'nil: ''Error
               else : if  equal(X, car(ns)) then car(vs)
                      else lookparam(X, cdr(ns), cdr(vs))
```

Fig. 5. Big-step semantics `int-big-step` in program form.

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

- Nondeterminism: reduction may be restricted to occur only from the outermost syntactic operator, using call-by-value, call-by-name, lazy evaluation or some other fixed strategy;
- Nondirectionality: rewriting occurs only from left to right, until a normal form is obtained (one that cannot be rewritten further).

More generally, nondeterminism is often resolved by defining explicit *evaluation contexts* $C[\,]$ from [16]. Each is an "expression with a hole in it," the hole indicated by $[\,]$. Determinism may be achieved by defining evaluation contexts so the *unique decomposition property* holds: any expression $E$ has at most one decomposition of form $E = C[E']$ where $E'$ is a redex and $C[\,]$ 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. The reason is that rewriting often leads to an infinity of different specialisation-time values. For example, this can occur in the $\lambda$-calculus if $\beta$-reduction is implemented by substitution in an expression with recursion, either implicitly by the $Y$ combinator or explicitly by a `fix` construct. The problem may be alleviated by using *closures* [40] instead of rewriting, essentially a "lazy" form of $\beta$-reduction.

One approach: Some small-step semantics have a property that can be thought of as **dual 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. However, the condition is violated if (for an example) 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 8.1.2 below.

### *7.4 Interpreters derived from denotational semantics*

A *denotational semantics* [42,52] consists of a collection of mathematical *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 6.4). In practice a denotational semantics resembles a program in a modern functional language such as ML, Haskell, or Scheme. As a consequence, 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.

## 8 Analysis of the example interpreters

Some programs are well-suited to specialisation, yielding substantial speedups, and others are not. Given a specialiser `spec`, a *binding-time improvement* is the transformation of program `p` into an equivalent program `q` such that $[\![\mathtt{p}]\!] = [\![\mathtt{q}]\!]$, but specialisation of `q` to a static value `s` will do more static computation than specialisation of `p` to the same `s`. We show several examples of binding-time improvements of interpreters used for compilation by specialisation.

### *8.1 Analysis of the instructional interpreter.*

The interpreter `int-instruct` of Figure 4 works quite well for computation and proof, and has been used for many exercises and proofs of program properties in courses such as "Introduction to Semantics". On the other hand, it does not specialise at all well!

#### *8.1.1 Lack of binding-time separation.*

Suppose one is given the source program `p`, i.e., the initial control stack contents are $C = \mathtt{p}\cdot()$, 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 appear in the specialiser output. In essence, no specialisation at all occurs, and source code appears in target programs generated by specialisation.

**A binding-time improvement.** The problem is easy to fix: Just push the above-mentioned syntactic entities onto the *control stack* $C$ instead of onto $S$: 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 6, again as a set of transition rules.

**Rules for Expressions**

Constant : $\langle S, M, c \cdot C\rangle \Rightarrow \langle c \cdot S, M, C\rangle$

Variable : $\langle S, M, \mathtt{x}i \cdot C\rangle \Rightarrow \langle M(\mathtt{x}i) \cdot S, M, C\rangle$

Composite : $\langle S, M, (e_1\, op\, e_2) \cdot C\rangle \Rightarrow \langle S, M, e_1 \cdot e_2 \cdot op \cdot C\rangle$

Operator : $\langle \underline{n_2} \cdot \underline{n_1} \cdot S, M, op \cdot C\rangle \Rightarrow \langle \underline{n_1\ op\ n_2} \cdot S, M, C\rangle$

**Rules for Programs**

Null : $\langle S, M, \mathtt{skip} \cdot C\rangle \Rightarrow \langle S, M, C\rangle$

Asgn : $\langle S, M, (\mathtt{x}i{:=}e) \cdot C\rangle \Rightarrow \boxed{\langle S, M, e \cdot \mathbf{assign} \cdot i \cdot C\rangle}$

Seq. : $\langle S, M, (p_1\,;p_2) \cdot C\rangle \Rightarrow \langle S, M, p_1 \cdot p_2 \cdot C\rangle$

Test : $\langle S, M, (\mathtt{if}\, b\, \mathtt{then}\, p_1\, \mathtt{else}\, p_2) \cdot C\rangle \Rightarrow \boxed{\langle S, M, b \cdot \mathbf{if} \cdot p_1 \cdot p_2 \cdot C\rangle}$

Loop : if $p = \mathtt{while}\, b\, \mathtt{do}\, p_1, \langle S, M,\ p \cdot C\rangle \Rightarrow \boxed{\langle S, M, b \cdot \mathbf{while} \cdot p_1 \cdot p \cdot C\rangle}$

**Rules for assign, if and while** (all modified)

$\langle \underline{n} \cdot S, M, \mathbf{assign} \cdot i \cdot C\rangle \Rightarrow \langle S, M[\mathtt{x}i \mapsto \underline{n}], C\rangle$

$\langle \mathtt{true} \cdot S, M, \mathbf{if} \cdot p_1 \cdot p_2 \cdot C\rangle \Rightarrow \langle S, M, p_1 \cdot C\rangle$

$\langle \mathtt{false} \cdot S, M, \mathbf{if} \cdot p_1 \cdot p_2 \cdot C\rangle \Rightarrow \langle S, M, p_2 \cdot C\rangle$

$\langle \mathtt{true} \cdot S, M, \mathbf{while} \cdot p_1 \cdot p \cdot C\rangle \Rightarrow \langle S, M, p_1 \cdot p \cdot C\rangle$

$\langle \mathtt{false} \cdot S, M, \mathbf{while} \cdot p_1 \cdot p \cdot C\rangle \Rightarrow \langle S, M, C\rangle$

Fig. 6. Instructional interpreter with modifications.

**Bounded static variation of the control stack.** In this version stack $C$ is built up only from pieces of the original program. Further, for any given source program, $C$ can take 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). Thus $C$ may safely be annotated as "static," so source code fragments will not appear in specialised programs.

This version specialises much better. The effect is to yield a target program consisting of tests to realise transitions from one $(S, M)$ pair to another — in essence with one transition (more or less) for each point in the original program.

*8.1.2 Extension to include procedures.*

Interpretation of a program with parameterless procedure calls is straightforward. Let *Pdefs* be a list of mutually recursive procedure definitions with entry $P : p$ if procedure $P$ has command $p$ as its body.

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.

**Rule for Procedure calls:**

$$\langle S, M, (\textbf{call } P) \cdot C \rangle \Rightarrow \langle S, M, \ p \cdot C \rangle \ \text{ if } \mathit{Pdefs} \text{ contains } P : p$$

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

```
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! The result: an unpleasant choice between infinite specialisation on the one hand (for any recursive procedure); or, on the other hand, classifying $C$ as dynamic, which will result in the existence of source code at run time, and very little speedup.

*A solution using "the trick".* [8]

Add a fourth component $R$, a "return stack" (initially empty) so a state becomes $\langle S, M, C, R \rangle$. The rules given in Figure 6 are used as is, except that an unchanged $R$ component is added to each rule. Procedure calls and returns are treated as follows (in place of the rule above):

**Rule for Procedure calls:**

$$\langle S, M, (\textbf{call } P) \cdot C, R \rangle \Rightarrow \langle S, M, \ p \cdot \textbf{return}, [C] \cdot R \rangle \quad \text{if } \mathit{Pdefs} \text{ contains } P : p$$

**Rule for Procedure returns:**

$$\langle S, M, \textbf{return}, [C] \cdot R \rangle \Rightarrow \langle S, M, C, R \rangle$$

The control stack $C$ is now of bounded static variation, as it contains only commands in the original program (from the main program, or from a procedure

[8] This is a programming device (described in [27,15]) to make static otherwise dynamic values, provided they are known to be of bounded static variation.

body followed by the token **return**). The return stack $R$ is not of BSV since its depth is not statically bounded; but it always has the form $[C_1] \cdot \ldots \cdot [C_k]$ where the $C_i$ are commands in the original program or procedure bodies. This can be exploited as follows:

Program the extended Figure 6 so the interpreter code for the **return** transition pops the top $[C]$ from $R$, and compares $C$ against each of the commands $C_1, \ldots, C_n$ found in the program immediately after a procedure call. Once some $C_i = C$ is found (as it must be), the interpreter continues to apply the transitions of Figure 6 using $C_i$ as new control stack.

The point of programming the interpreter this way is that $C_i$ comes from the program being interpreted *and so is a static value*. Consequently the interpreter's control stack will be static, and will not appear in any specialised program.

The specialised code for **return** will compare the top $R$ element with the source program commands that follow procedure calls, branching on equality to the specialised code for the selected $C_i$. The effect is that source text is used at run time only to implement the return.

The exact form of the source text $C_i$ from $R$ as used in these comparisons is clearly irrelevant, since its only function is to determine where in the target program control is to be transferred — and so each $R$ element can be replaced by a *token*, one for each source program command following a procedure call. Such tokens are in effect return addresses. Moreover, the effect of the comparisons and branches could in principle be realised in constant time, by an indirect jump. [9]

Interestingly, use of "the trick" led to something close to a procedure execution device that is widely used in pragmatic compiler construction.

### *8.2 Analysis of the "big-step" interpreter*

Interpreter `int-big-step` from Figure 5 specialises well. Even though not compositional (as defined in Section 7.4), it is *semicompositional* since the parameters `e`, `ns`, and `pgm` of `Eval` are always substructures of the original source program, and so are static. In this case recursion causes no problems: There is no explicit stack, and nothing can grow unboundedly as in the extended instructional interpreter, or as in many small-step semantics.

[9] Although logical, I know of no specialiser that uses this device.

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

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

Fig. 7. Addition of LET.

**A non-semicompositional extension.** Extend interpreter `int-big-step` of Figure 5 to accept a "LET" construction. This is easy to modify by adding an extra CASE as in Figure 7. With this change, parameter `ns` can take on values that are *not substructures of* `pgm`. Parameter `ns` can grow (and without limit) since more deeply nested levels of `let` expressions will give rise to longer lists `ns`. Nonetheless, parameter `ns` is of bounded static variation.

Why? Interpretation of a function call will reset `ns` to `getparams(f, pgm)`, but this always yields a value that is part of `pgm`. Further, `ns` *only grows when* `e` *shrinks*, so for any fixed source program, `ns` can only increase by the number of nested `let`-expressions in `pgm`. Thus parameter `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.

**Conclusion:** The extended interpreter `int-big-step` will specialise well.

### *8.2.1 Dynamic binding.*

Suppose the "function call" construction in interpreter `int-big-step` of Figure 5 were implemented differently, as in Figure 8. This causes `ns` to be of *unbounded static variation*. Intuitively, the reason is that the length of `ns` is no longer tied to any syntactic property of `pgm`.

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

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

Fig. 8. Dynamic name binding.

Interestingly, the dynamic name binding in the call has an indirect effect on specialisation of `Eval`: In order to avoid nontermination, the interpreter parameter `ns` must be classified as dynamic. 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 calls to function `lookparams` will involve a run-time loop every time a source-language variable is accessed, substantially increasing target program running times and space usage.

## 9 On code explosion

The QUALITY problem can be significant when specialising an interpreter whose static parameters are all of bounded static variation. Before showing an interpreter that suffers from code explosion, we look at a simple noninterpretive program to illustrate the root of the problem.

### *9.1 Synchronicity in static parameter variation*

**A good example: binary search.** The problem is efficiently to search a table whose size is statically known. Specialisation transforms generic table lookup code into more efficient code.

Program `p` of Figure 9 performs binary search for goal parameter `x` in an ordered (increasing) global table $\mathtt{T}_0,\ldots,\mathtt{T}_n$ where $n = 2^m - 1$. Parameter `i` points to a position in the table, and `delta` is the size of the portion of the table in which `x` is currently being sought.

```
Search(n, x) = Find(T, 0, n/2, x)

Find(T, i, delta, x) =
        if delta = 0 then
          {if x = T[i] then return(i) else return(NOTIN)}
        else
        let j = (if x >= T[i+delta] then i+delta else i)
        in
        Find(T, j, delta/2, x)
```

Fig. 9. Binary search algorithm.

If the table contains 8 elements then $n = 8$ and `Search(8, x)` calls `Find(T, 0, 4, x)`. We begin by assuming that `delta` is classified as static and that `i` is dynamic. Specialising with respect to static initial `delta` $= 2^{3-1} = 4$ and dynamic `i` gives program $\mathtt{p}_8$ of Figure 10.

In general $\mathtt{p}_n$ runs in time $O(\log(n))$, with a better constant coefficient than

the original general program. Moreover, it has size $O(\log(n))$ — acceptable for all but extremely large tables.

```
Search8(x) =
   let j = if x >= T[0+4] then 0 + 4 else 0 in
   let k = if x >= T[j+2] then j + 2 else j in
   let l = if x >= T[k+1] then k + 1 else k in
     { if x  = T[l] then return(l) else return(NOTIN) }
```

Fig. 10. A small program obtained by specialising binary search.

**A bad example: binary search.** To illustrate the code explosion problem, consider the binary search program above, again with known `n` $= n$. One may in principle also classify parameter `i` as static, since its static variation is the range $0, 1, \ldots, n-1$. In the resulting program, however, the test on `x` affects the value of static `i`. A perhaps unexpected result is *code size explosion.*

```
Search8(x) =
 if x >= T[4] then
  if x >= T[6] then
   if x >= T[7] then
    [if x = T[7] then return(7) else return(NOTIN)] else
    [if x = T[6] then return(6) else return(NOTIN)] else
   if x >= T[5] then
    [if x = T[5] then return(5) else return(NOTIN)]
    [if x = T[4] then return(4) else return(NOTIN)] else
  if x >= T[2] then
   if x >= T[3] then
    [if x = T[3] then return(3) else return(NOTIN)] else
    [if x = T[2] then return(2) else return(NOTIN)] else
   if x >= T[1] then
    [if x = T[1] then return(1) else return(NOTIN)]
    [if x = T[0] then return(0) else return(NOTIN)]
```

Fig. 11. A large program obtained by specialising binary search.

Specialisation with respect to static initial `delta` $= 4$ and `i` $= 0$ now gives the program in Figure 11. The specialised program again runs in time $O(\log(n))$, and with a slightly better constant coefficient than above. On the other hand it has size $O(n)$ — exponentially larger than the previous version! This is too large to be of practical use, except for small tables.

The problem cause is that parameters `i` and `delta` are *independently varying static parameters.* A specialiser must be accounted for *all possible combinations*

```
Run(pgm, input) = Exec(Cmd1, ns1, cons(input, 'nil), pgm)
         where
           f    = first_function(pgm)
           Cmd1 = getbody(f, pgm)
           ns1  = getparams(f, pgm)

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

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

  '(C1 ; C2)    :  Exec(C2, ns1, vs1, pgm)  where
      (ns1, vs1) = Exec(C1, ns, vs, pgm),

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

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

; Eval: Expression x Names x Values x Program -> Values

Eval(Exp, ns, vs, pgm) = ...
```

Fig. 12. Interpreter for a simple imperative language.

of their values – in this case, $n-1$ combinations. This leads to specialised programs of size $O(n)$ rather than $O(\log(n))$.

### *9.2 An unfortunate programming style*

The interpreter sketched in Figure 12 implements a tiny imperative language. Unfortunately it does not specialise at all well due to code explosion similar to that of the binary search program. First, note that `Expression`, `Names` and `Program` are all of BSV and so can all be classified as "static".

This interpreter uses a quite natural idea: If a variable has not already been bound in the store, then it is added, with initial value zero. This creates no problems when running the interpreter. However, this seemingly innocent trick can cause catastrophically large target programs to be generated when specialising the interpreter to a source program. The reason is that `spec` must take account of *all execution possibilities* (the "Pachinko syndrome").

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

Fig. 13. A source program in the simple imperative language.

For an example, consider the source program of Figure 13. After `test1` is processed at specialisation time, `spec` has to allow for either branch to have been taken, so the resulting value of `ns` could be either `[]` or `[X1]`. After the second test `X2` may be either present or absent, giving 4 possibilities: `ns` = `[]`, `ns` = `[X1]`, `ns` = `[X2]`, or `ns` = `[X1,X2]`. There are thus $2^n$ possible values for `ns` when `-- some code --` is encountered, so specialisation will generate a target program whose size is (quite unnecessarily) $\Omega(2^n)$.

The problem is easily fixed, by revising the interpreter first to 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 of the source program. A further optimisation now becomes possible: The type of `Exec` can be simplified to the following, with correspondingly smaller target code.

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

*9.3 Dead static parameters.*

This is yet another manifestation of the same problen: a combinatorial explosion in the set of static value.

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.

Dead variable detection can be even more important in program specialisation than in compilation. The size of specialised program $\mathtt{p}_\mathtt{s}$ critically depends on the number of its *specialised program points*: Each has form $\ell_{(\mathtt{s}_1,\ldots,\mathtt{s}_k)}$ where $\ell$ is one of `p`'s program points, and $(\mathtt{s}_1,\ldots,\mathtt{s}_k)$ is a tuple of values of the static parameters of `p` whose scope includes point $\ell$.

Thus `p` has a dead static parameter `X`, the size of $\mathtt{p}_\mathtt{s}$ may be multiplied by the number of static values that `X` can assume. This will increase the number of

specialised program points in $\ell_{(\mathtt{s}_1,\ldots,\mathtt{s}_k)}$, even though the computations by $\mathtt{p}_\mathtt{s}$ will not be changed in any way.

Lesson: A specialiser can dramatically reduce the size of the specialised program by detecting the subject program's dead static parameters before specialisation time, and then making them dynamic. My first experience with this problem, for a simple imperative language (see [27], Section 4.9.2) was that not accounting for dead static variables led to a specialised program 500 times larger than necessary(!).

## 10 On speedup

Since a major motivation for program specialisation is speedup, we now examine more carefully the question of how much speedup can be obtained.

### *10.1 Speedup by change of evaluation algorithm*

An interpreter has the freedom to execute its source program in any way it wishes, as long as the same input-output function is computed. This can be exploited to give asymptotically superlinear Type 2 speedups.

**Selective memoisation.** A common example from the program transformation literature [1,5,32] is the use of *memo tables*: A function value, once computed, is cached so a later function call with the same arguments can be replaced by a table search. While memoisation can give dramatic speedups, it can also use large amounts of memory, and is only of benefit when the same function is in fact frequently called with identical arguments.

Let `interp` be a self-interpreter for language **L**. A natural source language extension is to annotate function calls and return values in a program with remarks saying where to save values in a memo table, and where to search a memo table. For example, an annotated Fibonacci program:

```
f(n) = [save](if n <= 1 then n else f(n-1)+[srch]f(n-2))
```

An interpreter could, on encountering [*srch*]`f(exp)`, evaluate `exp` to value $v$ and then search the `f`-memo table. If a pair $(v, w)$ is found, then return $w$ without calling `f`, else evaluate its body as usual. The effect of annotation

`f(n) =` *save* `result` is to ensure that once `result` has been evaluated, its value is saved in the `f`-memo table. With such an evaluation strategy, the Fibonacci program runs in linear time with a table size linear in `n` (assuming constant-time memo table access). The effect of specialisation is to compile the program into target code that accesses memo tables or stores into them only where specified by the user.

It would be desirable to have a program analysis to determine which functions may have "overlap," i.e., be called repeatedly with the same arguments in a computation. This could be used to decide which annotations to place where.

The 'tupling" transformation is one way to achieve the effect of memoisation purely functionally, without having a memo table. Progress towards automating this transformation is seen in [11]. Liu and Stoller have a semi-automated methodology to exploit memoisation opportunities [32], and Acar et.al. use a modal type system to analyse data flow for memoisation purposes [1].

**Limits to Type 2 speedups.** A memoising interpreter is impressive because its effect can be to interpret programs faster than they actually run. On the other hand, there are some mathematical limits: If a problem has lower-bound exponential complexity (for example), then no program can solve it essentially faster, even if memoisation or other sophisticated evaluation mechanisms are used.

### *10.2 Target speed in relation to interpreter speed*

We begin with a resume (and improvement) of the argument in [27,3] that Type 1 speedup is at most a linear multiplicative factor for the usual specialisation algorithms. A subtle point, though, is that the size of the linear coefficient can depend on the values of the static inputs.

An example: A naive string matcher taking time $O(|pattern| \cdot |subject|)$ can be specialised to the pattern to yield a matcher with runtime $O(|subject|)$ with coefficient independent of the pattern [12,2]. In this case, larger static data leads to larger speedups as a function of the size of the dynamic input; but *for each value of the static input*, the speedup is linear.

The argument below also applies to *deforestation* and *supercompilation.*

*10.2.1 Relating specialised and original computations*

Consider a given program p, and static and dynamic inputs s, d. Let $\mathtt{p}_\mathtt{s}$ be the result of specialising p to s. **A first observation** (on all functional or imperative program specialisers I have seen): There is an order-preserving *injective mapping* $\psi$

- *from* the operations done in the computation of $[\![\mathtt{p}_\mathtt{s}]\!](\mathtt{d})$
- *to* the operations done in the original computation of $[\![\mathtt{p}]\!](\mathtt{s},\mathtt{d})$.

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 specialised 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 specialised computations can be embedded 1-1 into original ones.

**A second observation, in the opposite direction**: Some operations done in the computation of $[\![\mathtt{p}]\!](\mathtt{s},\mathtt{d})$ can be forced to be specialised because they use parts of d, which is unavailable at specialisation time. We use the term *forced dynamic* for any such operation.

Since a forced dynamic operation cannot be done by spec, specialised code must be executed to realise its effect. It holds for most program specialisers that there exists a *surjective order-preserving mapping* $\phi$ from

- the forced dynamic operations in the computation of $[\![\mathtt{p}]\!](\mathtt{s},\mathtt{d})$, onto
- the corresponding operations in the computation of $[\![\mathtt{p}_\mathtt{s}]\!](\mathtt{d})$.

Figure 14 shows these correspondences between the original and specialised computations.

**Why specialisers give at most linear speedup.** Let $\mathit{time\text{-}force}_\mathtt{p}(\mathtt{s},\mathtt{d})$ be the number of forced dynamic operations done while computing $[\![\mathtt{p}]\!](\mathtt{s},\mathtt{d})$. Since all of these *must* be performed by the specialised program, we get a lower bound on its running time:

$$\mathit{time\text{-}force}_\mathtt{p}(\mathtt{s},\mathtt{d}) \leq \mathit{time}_{\mathtt{p}_\mathtt{s}}(\mathtt{d}) \tag{8}$$

All other p operations can be performed without knowing d: they depend only on s. The maximum number of such operations performed by the specialiser [10] is thus some function $f(\mathtt{s})$ of the static program input. Thus at most $f(\mathtt{s})$

[10] Assuming, of course, that specialisation terminates.

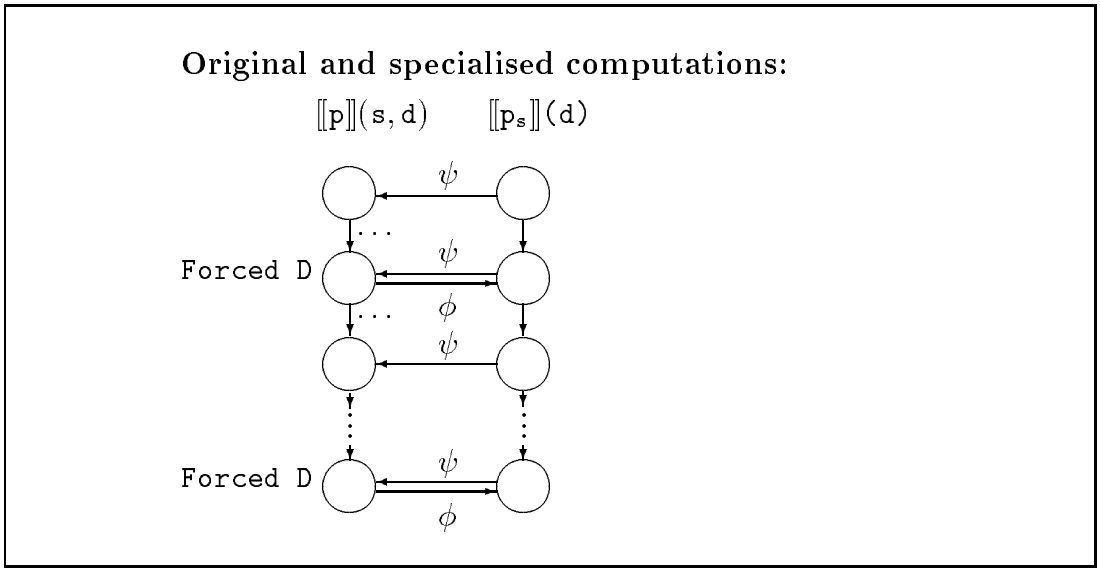


Fig. 14. Correspondences between original and specialised computations.

static operations occur between any two forced dynamic operations, yielding

$$time_{\mathtt{p}}(\mathtt{s},\mathtt{d}) \leq (1 + f(\mathtt{s})) \cdot \textit{time-force}_{\mathtt{p}}(\mathtt{s},\mathtt{d}) \tag{9}$$

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

$$\text{speedup}_{\mathtt{s}}(\mathtt{d}) = \frac{time_{\mathtt{p}}(\mathtt{s},\mathtt{d})}{time_{\mathtt{p_s}}(\mathtt{d})} \leq 1 + f(\mathtt{s}) \tag{10}$$

This observation explains the fact that, in general, program specialisers yield at most linear speedup.

### *10.2.2 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 specialised ones. This is not always the case, and specialisers working differently can obtain superlinear speedup. Three cases follow:

**Dead code elimination**: If `spec` can examine its specialised 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 $[\![\mathtt{p}]\!](\mathtt{s},\mathtt{d})$ operations not corresponding to any $[\![\mathtt{p_s}]\!](\mathtt{d})$ operations at all. In this situation, between the two `p` operations corresponding to two $\mathtt{p_s}$ operations there can occur a number of `p` operations *depending on dynamic input* that were removed by the dead code elimination technique.

**Specialisation-time detection of dead branches:** A logic programming analogy is specialisation-time detection of branches in the Prolog's *SLD computation tree* that are guaranteed to fail (and may depend on dynamic data). This is much more significant than in functional programming, where 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, less trivial than just eliding useless computations [1,5,32].

## 11 Conclusions

Given an interpreter `interp`, any program `source` may be transformed by computing

$$\texttt{target} = [\![\texttt{spec}]\!](\texttt{interp}, \texttt{source})$$

This approach is appealing because of its top-down nature: Rather than considering one transformation of one program at a time, a global transformation effect is obtained by the device of writing an apropriate interpreter.

The main challenges to overcome to use the approach in practice are:

- to write `interp` so it realises, while executing `source`, the optimisations the user wishes to obtain (style change, memoisation, instrumentation, etc.); and
- to write `interp` so as many operations as possible are given static binding times, so `target` will be as efficient as possible.

Once an interpreter `interp` has been written to allow good binding-time separation, the binding-time analysis and the specialisation phases in modern specialisers are completely automatic. Further, the the "generating extension" `interp-gen` (explained in detail in [27]), allows `target` to be directly constructed from `source`, without having to run a slow and general-purpose specialiser.

*Acknowledgements*

Thanks are due to the referees for constructive comments; to many in the Topps group at DIKU, in particular Arne Glenstrup, A.-F. Le Meur and students in the 2003 course on Program Transformation; and to participants in the 1996 Partial Evaluation Dagstuhl meeting and the 1998 Partial Evaluation

Summer School, in particular Olivier Danvy, Torben Mogensen, Robert Glück and Peter Thiemann.

## References

[1] Umut A. A. Acar, Guy E. Blelloch, and Robert Harper. Selective memoization. In *Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 14–25. ACM Press, 2003.

[2] Mads Sig Ager, Olivier Danvy, and Henning Korsholm Rohde. Fast partial evaluation of pattern matching in strings. In *Proceedings of the 2003 ACM SIGPLAN workshop on Partial evaluation and semantics-based program manipulation*, pages 3–9. ACM Press, 2003.

[3] L.O. Andersen and C.K. Gomard. Speedup analysis in partial evaluation (preliminary results). In *Partial Evaluation and Semantics-Based Program Manipulation, New Haven, Connecticut. (Sigplan Notices, vol. 26, no. 9, September 1991)*, pages 1–7, 1992.

[4] Kenichi Asai, Satoshi Matsuoka, and Akinori Yonezawa. Duplication and partial evaluation —for a better understanding of reflective languages—. *Lisp and Symbolic Computation*, 9:203–241, 1996.

[5] R. S. Bird. Improving programs by the introduction of recursion. *Communications of the ACM*, 20(11):856–863, 1977.

[6] D. Bjorner, Neil D. Jones, and A. P. Ershov. *Partial Evaluation and Mixed Computation: Proceedings of the IFIP TC2 Workshop, Gammel Avernaes, Denmark, 18-24 Oct., 1987.* Elsevier Science Inc., 1988.

[7] A. Bondorf. Automatic autoprojection of higher order recursive equations. In *Proceedings of the third European symposium on programming on ESOP '90*, pages 70–87. Springer-Verlag New York, Inc., 1990.

[8] Anders Bondorf and Olivier Danvy. Automatic autoprojection of recursive equations with global variables and abstract data types. *Science of Computer Programming*, 16:151–195, 1991.

[9] Anders Bondorf and Olivier Danvy. Automatic autoprojection of recursive equations with global variables and abstract data types. *Science of Computer Programming*, 16(2):151–195, 1991.

[10] Anders Bondorf and Olivier Danvy. Automatic autoprojection of recursive equations with global variables and abstract data types. *Science of Computer Programming*, 16(2):151–195, 1991.

[11] Wei-Ngan Chin, Siau-Cheng Khoo, and Peter Thiemann. Synchronization analyses for multiple recursion parameters (extended abstract). In *Selected Papers from the International Seminar on Partial Evaluation*, pages 33–53. Springer-Verlag, 1996.

[12] C. Consel and O. Danvy. Partial evaluation of pattern matching in strings. *Information Processing Letters*, 30(2):79–86, 1989.

[13] C. Consel, J.L. Lawall, and A-F. Le Meur. A tour of Tempo: a program specializer for the C language. *Science of Computer Programming*, (to appear).

[14] O. Danvy, R. Glück, and Peter Thiemann. *Partial Evaluation. Dagstuhl castle, Germany*, volume 1110 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.

[15] Olivier Danvy, Karoline Malmkj¿r, and Jens Palsberg. Eta-expansion does The Trick. *ACM Transactions on Programming Languages and Systems*, 18(6):730–751, November 1996.

[16] Matthias Felleisen and Robert Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 103(2):235–271, 1992.

[17] 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):45–50, 1971.

[18] J. Gallagher. Transforming logic programs by specialising interpreters. In *Proceedings of the 7th European Conference on Artificial Intelligence (ECAI-86), Brighton*, pages 109–122, 1986.

[19] J. Gallagher, M. Codish, and E. Shapiro. Specialisation of Prolog and FCP programs using abstract interpretation. *New Generation Computing*, 6(2,3):159–186, 1988.

[20] J. P. Gallagher. Tutorial on specialisation of logic programs. In *Proceedings of the 1993 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 88–98. ACM Press, 1993.

[21] Arne J. Glenstrup, Henning Makholm, and Jens P. Secher. C-MIX: Specialization of C programs. In *Partial Evaluation - Practice and Theory, DIKU 1998 International Summer School*, pages 108–154. Springer-Verlag, 1999.

[22] Robert Glück. On the generation of specializers. *Journal of Functional Programming*, 4(4):499–514, October 1994.

[23] Robert Glück and Jesper Jørgensen. Generating transformers for deforestation and supercompilation. In B. Le Charlier, editor, *Static Analysis*, volume 864 of *Lecture Notes in Computer Science*, pages 432–448. Springer-Verlag, 1994.

[24] John Hatcliff, Torben Mogensen, and Peter Thiemann. *Partial Evaluation - Practice and Theory, DIKU 1998 International Summer School*, volume 1706 of *Lecture Notes in Computer Science*. Springer-Verlag, 1999.

[25] Neil D. Jones. What not to do when writing an interpreter for specialisation. In *Selected Papers from the Internaltional Seminar on Partial Evaluation*, pages 216–237. Springer-Verlag, 1996.

[26] Neil D. Jones and Arne J. Glenstrup. Program generation, termination, and binding-time analysis. In *The ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering*, pages 1–31. Springer-Verlag, 2002.

[27] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial evaluation and automatic program generation.* Prentice-Hall, Inc., 1993.

[28] Jesper Jørgensen. Compiler generation by partial evaluation. Master's thesis, DIKU, University of Copenhagen, Denmark, 1991.

[29] Jesper Jørgensen. SIMILIX: A self-applicable partial evaluator for Scheme. In *Partial Evaluation - Practice and Theory, DIKU 1998 International Summer School*, pages 83–107. Springer-Verlag, 1999.

[30] Michael Leuschel. Advanced logic program specialisation. In *Partial Evaluation - Practice and Theory, DIKU 1998 International Summer School*, pages 271–292. Springer-Verlag, 1999.

[31] Michael Leuschel. Logic program specialisation. In *Partial Evaluation - Practice and Theory, DIKU 1998 International Summer School*, pages 155–188. Springer-Verlag, 1999.

[32] Yanhong A. Liu and Scott D. Stoller. Dynamic programming via static incrementalization. *Higher Order and Symbolic Computation*, 16(1-2):37–62, March 2003.

[33] J.W. Lloyd and J.C. Shepherdson. Partial evaluation in logic programming. *Journal of Logic Programming*, 11(3-4):217–242, 1991.

[34] Hidehiko Masuhara, Gregor Kiczales, and Chris Dutchyn. A compilation and optimization model for aspect-oriented programs. In *Proceedings of 12th International Conference on Compiler Construction (CC2003)*, volume 2622 of *Lecture Notens in Computer Science*, pages 46–60, 2003.

[35] Dylan McNamee, Jonathan Walpole, Calton Pu, Crispin Cowan, Charles Krasic, Ashvin Goel, Perry Wagle, Charles Consel, Gilles Muller, and Renauld Marlet. Specialization tools and techniques for systematic optimization of system software. *ACM Transactions on Computer Systems (TOCS)*, 19(2):217–251, 2001.

[36] A-F. Le Meur, J.L. Lawall, and C. Consel. Specialization scenarios: a pragmatic approach to declaring program specialization. *Higher-Order and Symbolic Computation*, ??(??):??–??, to appear.

[37] Torben Mogensen. Inherited limits. In John Hatcliff, Torben Mogensen, and Peter Thiemann, editors, *Partial Evaluation: Practice and Theory. Proceedings of the 1998 DIKU International Summerschool*, volume 1706 of *Lecture Notes in Computer Science*, page 202. Springer-Verlag, 1998.

[38] G. Muller, J.L. Lawall, S. Thibault, and R.E.V. Jensen. A domain-specific language approach to programmable networks. *Transactions on Systems, Man and Cybernetics*, 33(3), 2003.