An Introduction to Online and Offline Partial Evaluation Using a Simple Flowchart Language

John Hatcliff *

Department of Computing and Information Sciences
Kansas State University
hatcliff@cis.ksu.edu **

Abstract. These notes present basic principles of partial evaluation using the simple imperative language FCL (a language of flowcharts introduced by Jones and Gomard). Topics include online partial evaluators, offline partial evaluators, and binding-time analysis. The goal of the lectures is to give a rigorous presentation of the semantics of partial evaluation systems, while also providing details of actual implementations. Each partial evaluation system is specified by an operational semantics, and each is implemented in Scheme and Java. Exercises include proving various properties about the systems using the operational semantics, and modifying and extending the implementations.

1 Introduction

These notes give a gentle introduction to partial evaluation concepts using a simple flowchart language called FCL. The idea of using FCL to explain partial evaluation is due to Gomard and Jones [11, 15], and much of the material presented here is simply a reworking of the ideas in their earlier tutorial for offline partial evaluation. I have added analogous material for online partial evaluation, presented the binding-time analysis for offline partial evaluation using a two-level language, and specified FCL evaluation and partial evaluation using a series of operational semantics definitions. The operational semantics definitions provide enough formalization so that one can prove the correctness of the given specializers with relative ease.

The goal of this tutorial is to present partial evaluators that

- are easy to understand (they have a very clean semantic foundation),
- are simple enough for students to code quickly, and that
- capture the most important properties that one encounters when specializing programs written in much richer languages.

* Supported in part by NSF under grant CCR-9701418, and NASA under award NAG 21209.

** 234 Nichols Hall, Manhattan KS, 66506, USA. Home page: http://www.cis.ksu.edu/~hatcliff

For each specializer presented, there is an accompanying Scheme and Java implementation. The Java implementations include web-based documentation and are generally more sophisticated than the Scheme implementations. On the other hand, the Scheme implementations are much shorter and can be understood more quickly. For programming exercises, students are given code templates for the specializers and are asked to fill in the holes. The more mundane portions of the implementation (code for parsing, stores, and other data structures) are provided. Other exercises involve using the operational semantics to study particular aspects of the specializers, and applying the specializers to various examples. The current version of these materials can be found on my home page (`http://www.cis.ksu.edu/~hatcliff`).

Each section of the notes ends with some references for further reading. The references are by no means exhaustive. Since the presentation here is for a simple imperative language, the given references are mostly for related work on imperative languages. Even though the references there are slightly out of date, the best place to look for pointers to work on partial evaluation in general is still the Jones-Gomard-Sestoft book on partial evaluation [15]. In addition, one should look at the various PEPM proceedings and the recent special issue of ACM Computing Surveys (1998 Symposium on Partial Evaluation [6]), and, of course, the other lecture material presented at this summer school.

This tutorial material is part of a larger set of course notes, lecture slides, and implementations that I have used in courses on partial evaluation at Oklahoma State University and Kansas State University. In addition to online and offline partial evaluation, the extended set of notes uses FCL to introduce other topics including constraint-based binding-time analysis, generating extension generators, slicing, and abstraction-based program specialization. Correctness proofs for many of the systems are also given. These materials can also be found on my home page (the `URL` is given above).

Acknowledgements

I'm grateful to Shawn Laubach for the many hours he spent on the implementations and on helping me organize these notes. Other students in my partial evaluation course at Oklahoma State including Mayumi Kato and Muhammad Nanda provided useful feedback. I'd like to thank Matt Dwyer, Hongjun Zheng and other members of the SANTOS Laboratory at Kansas State (`http://www.cis.ksu.edu/~santos`) for useful comments and support. Finally, Robert Glück and Neil Jones deserve special thanks for encouraging me to prepare this material, for enlightening discussions, and for their very helpful and detailed comments on earlier drafts of this article.

2 The Flowchart Language

The section presents the syntax and semantics of a simple *flowchart language* FCL. This will be our primary linguistic vehicle for presenting fundamental

```
(m n)
(init)

  init: result := 1;
        goto test;

  test: if <(n 1)
           then end
           else loop;

  loop: result := *(result m);
        n := -(n 1);
        goto test;

  end:  return result;
```

Fig. 1. An FCL program to compute m^n

concepts of program analysis and specialization. FCL is a good language to use for introducing basic concepts because it is extremely simple. Yet as Gomard and Jones note [11], all the concepts required for partial evaluation of FCL reappear again when considering more realistic languages.

2.1 Syntax

We usually think of a flowchart as a diagram containing boxes of various shapes connected by some arrows. One way to formalize flowcharts is using the notion of a graph. For example, one might start by saying that a flowchart is a directed graph with various types of nodes. This method of representing computation is intuitively appealing, but it somewhat awkward to work with in practice since we have grown accustomed to programming using text-based languages instead of diagramatic languages.

We will give a linguistic formalization of flowcharts. That is, we define a programming language FCL, and programs in FCL will correspond to the pictures that usually pop into our minds when we think of flowcharts. We begin an example, and then move to a formal presentation of the syntax of FCL.

A FCL program Figure 1 presents an FCL program that computes the power function. The input parameters the program are `m` and `n`. These are simply variables that can referenced and assigned throughout the program. There are no other declarations in FCL. Other variables such as `result` can be introduced at any time. The initial value of a variable is 0.

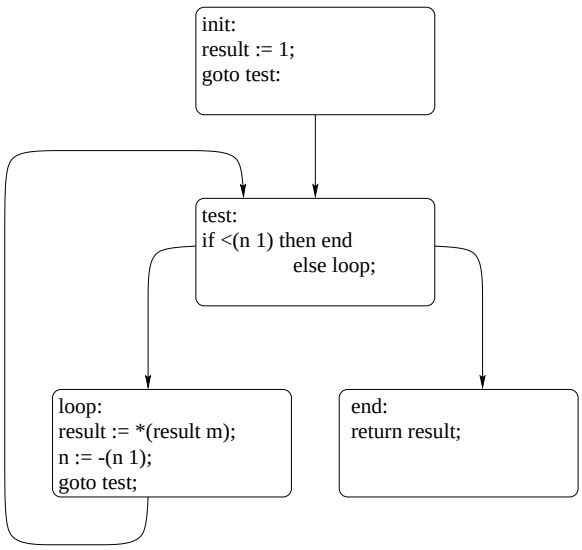

Fig. 2. A flowchart diagram corresponding to the FCL power program

FCL programs are essentially lists of *basic blocks*. The initial basic block to be executed is specified immediately after the parameter list. In the power program, the initial block is specified by the line `(init)`.

Each basic block consists of a *label* followed a (possibly empty) list of *assignments*. Each block concludes with a *jump* that transfers control from that block to another one. For example, in the power program, the first block has label `init`, an assignment list of length one (`result := 1;`), and a jump `goto test;`.

FCL contains three kinds of jumps: an unconditional jump `goto` *label*, a conditional jump `if` *test* `goto` *label* `else` *label*, and a special jump `return` *exp* that terminates program execution and yields the value of *exp*. Instead of including boolean values, any non-zero value represents *true* and zero represents *false*.

Figure 2 displays the flowchart diagram corresponding to the FCL power program of Figure 1. Each node in the diagram corresponds to a basic block in the FCL program. The diagram helps illustrate that the basic aspects of computation in flowcharts are *transformations of computer memory* (computed by the assignments in each basic block) and *control transfers* (computed by the jumps).

Formal definition of FCL syntax Figure 3 presents the syntax of FCL. The intuition should be fairly clear given the example presented above. We will usually omit the dot $\cdot$ representing an empty list in assignment lists al.

Syntax Domains

$p \in$ Programs[FCL]	$x \in$ Variables[FCL]
$b \in$ Blocks[FCL]	$e \in$ Expressions[FCL]
$l \in$ Block-Labels[FCL]	$c \in$ Constants[FCL]
$a \in$ Assignments[FCL]	$j \in$ Jumps[FCL]
$al \in$ Assignment-Lists[FCL]	$o \in$ Operations[FCL]

Grammar

$$
\begin{array}{rcl}
p & ::= & (x^*)\ (l)\ b^+ \\
b & ::= & l :\ al\ j \\
a & ::= & x := e; \\
al & ::= & a\ al \mid \cdot \\
e & ::= & c \mid x \mid o(e^*) \\
j & ::= & \textbf{goto}\ l; \mid \textbf{return}\ e; \mid \textbf{if}\ e\ \textbf{then}\ l_1\ \textbf{else}\ l_2;
\end{array}
$$

Fig. 3. Syntax of the Flowchart Language FCL

The syntax is parameterized on Constants[FCL] and Operations[FCL]. Presently, we only consider computation over numerical data and so we use the following definitions.

$$
\begin{array}{rcl}
c & ::= & \mathtt{0} \mid \mathtt{1} \mid \mathtt{2} \mid \ldots \\
o & ::= & + \mid - \mid * \mid = \mid < \mid > \mid \ldots
\end{array}
$$

One can easily add additional data types such as lists.

The syntactic categories of FCL (*e.g.*, programs, basic blocks, *etc.*) are given in of Figure 3 (*e.g.*, Programs[FCL], Blocks[FCL], *etc.*). To denote the components of a syntactic category for a particular program p, we write Blocks[p], Expressions[p], *etc.* For example, Block-Labels[p] is the set of labels that label blocks in p. Specifically,

$$\text{Labels}[p] \stackrel{\text{def}}{=} \{l \in \text{Block-Labels[FCL]} \mid \exists b\,.\, b \in \text{Blocks}[p] \text{ and } b = l :\ a\ j\}.$$

We will assume that all FCL programs p that we consider are *well-formed* in that sense that every label used in a jump in p appears in Block-Labels[p].

Definition 1 (well-formed FCL programs). *A* FCL *program p is well-formed if*

- (**goto** $l;$, $\in$ Jumps[p] *imples* $l \in$ Block-Labels[p], *and*
- (**if** e **then** l_1 **else** $l_2;$) $\in$ Jumps[p] *imples* $l_1, l_2 \in$ Block-Labels[p].

2.2 Semantics

We now turn our attention toward formalizing the *behaviour* of FCL programs in terms of *execution traces*.

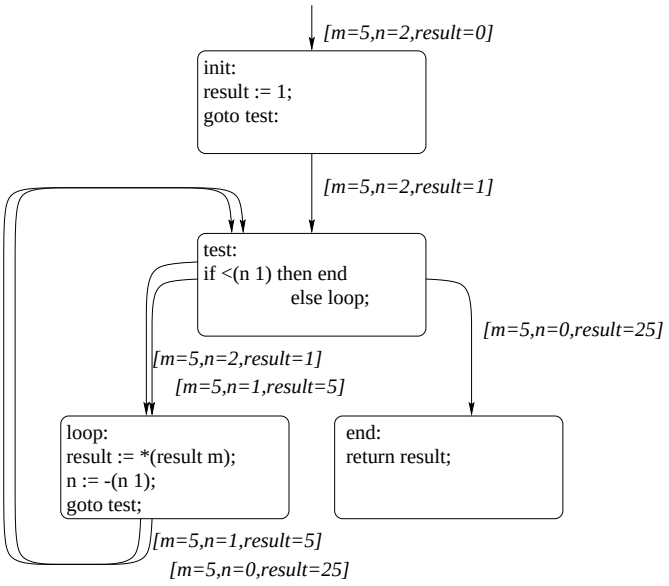

Fig. 4. Execution trace's path through the flowchart

Execution traces Intuitively, an execution trace shows the steps a program makes between *computational states* where a computational state consists of a label indicating the current basic block and the current value of the store. For example, the following is a trace of the power program computing 5^2.

$$
\begin{array}{ll}
 & (\mathtt{init}, [\mathtt{m} \mapsto 5, \mathtt{n} \mapsto 2, \mathtt{result} \mapsto 0]) \\
\textit{step 1:} & \rightarrow (\mathtt{test}, [\mathtt{m} \mapsto 5, \mathtt{n} \mapsto 2, \mathtt{result} \mapsto 1]) \\
\textit{step 2:} & \rightarrow (\mathtt{loop}, [\mathtt{m} \mapsto 5, \mathtt{n} \mapsto 2, \mathtt{result} \mapsto 1]) \\
\textit{step 3:} & \rightarrow (\mathtt{test}, [\mathtt{m} \mapsto 5, \mathtt{n} \mapsto 1, \mathtt{result} \mapsto 5]) \\
\textit{step 4:} & \rightarrow (\mathtt{loop}, [\mathtt{m} \mapsto 5, \mathtt{n} \mapsto 1, \mathtt{result} \mapsto 5]) \\
\textit{step 5:} & \rightarrow (\mathtt{test}, [\mathtt{m} \mapsto 5, \mathtt{n} \mapsto 0, \mathtt{result} \mapsto 25]) \\
\textit{step 6:} & \rightarrow (\mathtt{end}, [\mathtt{m} \mapsto 5, \mathtt{n} \mapsto 0, \mathtt{result} \mapsto 25]) \\
\textit{step 7:} & \rightarrow (\langle \mathsf{halt}, 25 \rangle, [\mathtt{m} \mapsto 5, \mathtt{n} \mapsto 0, \mathtt{result} \mapsto 25])
\end{array}
$$

Here we have introduced an special label $\langle \mathsf{halt}, 25 \rangle$ not found in the original program. In general, $\langle \mathsf{halt}, v \rangle$ labels a final program state where the return value is v.

Computation tree An execution trace of program p gives a particular path through p's flowchart. For example, Figure 4 shows the path corresponding to the trace above. The trace steps are given as labels on the flowchart arcs.

Of course, a flowchart is just a finite representation of a (possibly infinite) tree obtained by unfolding all the cycles in the flowchart graph. We call such trees

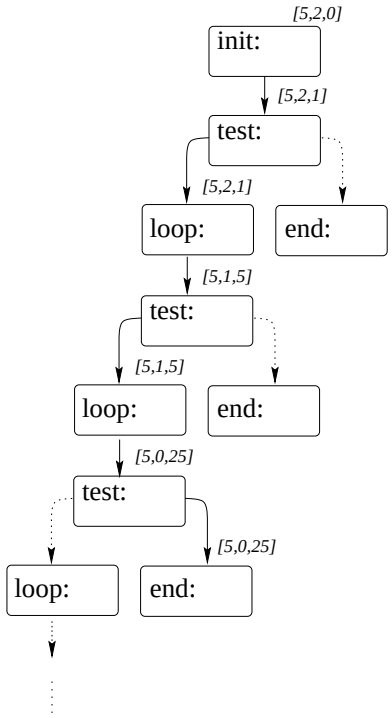

Fig. 5. Execution trace's path through the computation tree

computation trees. Instead of viewing as an execution trace as a path through a flowchart, it will be more convenient for us to view a trace as a path through an computation tree. For example, Figure 5 shows the tree path corresponding to the flowchart path of 4. When reasoning about program specialization in the following chapters, we prefer tree paths instead of flowchart paths since every node in the tree path has exactly one state associated with it.

Formal definition of FCL semantics We now formalize the notion of an execution trace using an *operational semantics*. Our main task will be to define the *transition relation* $\rightarrow$ between FCL computational states. Given this formal high-level specification of execution, it should be straightforward for the reader to build an actual implementation in the programming language of his or her choice.

Figure 6 presents the operational semantics for FCL. The semantics relies on the following definitions.

- **Values:** Each expression in the language will evaluate to some *value* $v \in$ Values[FCL]. The semantics is parameterized on the set Values[FCL], just like the syntax is parameterized on constants and operations. We previously decided to include constants and operations for numerical computation. Accordingly, we will define

$$v \in \text{Values[FCL]} = \{0, 1, 2, \ldots\}.$$

  Another view of values is that they abstract the underlying implementation's representation of data objects. For example, it is important to distinguish between the numeral 2 (*i.e.*, syntax, source code – written in `teletype` font) and the corresponding value 2 (*i.e.*, semantics, abstraction of underlying representation – written in roman font) that is obtained when 2 is evaluated. We will use $[\![\cdot]\!]$ to denote an injective mapping from syntactic objects to their semantic counterparts. For example $[\![2]\!] = 2$.
  The meanings of operations for numerical computation are as expected. For example,

$$\begin{aligned} [\![+]\!](n_1, n_2) &= n_1 + n_2 \\ [\![-]\!](n_1, n_2) &= \begin{cases} 0 & \text{if } n_1 - n_2 < 0 \\ n_1 - n_2 & \text{otherwise} \end{cases} \\ &\ldots \\ [\![=]\!](n_1, n_2) &= \begin{cases} 1 \text{ if } n_1 = n_2 \\ 0 \text{ otherwise} \end{cases} \\ &\ldots \end{aligned}$$

  Since we do not include boolean values, we define predicates is-true? and is-false? such that is-false?(0) and is-true?(v) for all $v > 0$.
- **Stores:** A *store* $\sigma \in$ Stores[FCL] holds the current values of the program variables. Thus, a store corresponds to the computer's memory. Formally, a store $\sigma \in$ Stores[FCL] is partial function from Variables[FCL]

Expressions

$$\frac{}{\sigma \vdash_{expr} c \Rightarrow [\![c]\!]} \qquad \frac{}{\sigma \vdash_{expr} x \Rightarrow \sigma(x)}$$

$$\frac{\sigma \vdash_{expr} e_i \Rightarrow v_i \qquad [\![o]\!](v_1 \ldots v_n) = v}{\sigma \vdash_{expr} o(e_1 \ldots e_n) \Rightarrow v}$$

Assignments

$$\frac{\sigma \vdash_{expr} e \Rightarrow v}{\sigma \vdash_{assign} x := e; \Rightarrow \sigma[x \mapsto v]} \qquad \frac{}{\sigma \vdash_{assigns} \cdot \Rightarrow \sigma}$$

$$\frac{\sigma \vdash_{assign} a \Rightarrow \sigma' \qquad \sigma' \vdash_{assigns} al \Rightarrow \sigma''}{\sigma \vdash_{assigns} a\ al \Rightarrow \sigma''}$$

Jumps

$$\frac{}{\sigma \vdash_{jump} \textbf{goto}\ l; \Rightarrow l} \qquad \frac{\sigma \vdash_{expr} e \Rightarrow v}{\sigma \vdash_{jump} \textbf{return}\ e; \Rightarrow \langle \mathsf{halt}, v \rangle}$$

$$\frac{\sigma \vdash_{expr} e \Rightarrow v \qquad \mathsf{is\text{-}true?}(v)}{\sigma \vdash_{jump} \textbf{if}\ e\ \textbf{then}\ l_1\ \textbf{else}\ l_2; \Rightarrow l_1} \qquad \frac{\sigma \vdash_{expr} e \Rightarrow v \qquad \mathsf{is\text{-}false?}(v)}{\sigma \vdash_{jump} \textbf{if}\ e\ \textbf{then}\ l_1\ \textbf{else}\ l_2; \Rightarrow l_2}$$

Blocks

$$\frac{\sigma \vdash_{assigns} al \Rightarrow \sigma' \qquad \sigma' \vdash_{jump} j \Rightarrow l'}{\sigma \vdash_{block} l :\ al\ j \Rightarrow (l', \sigma')}$$

Transitions

$$\frac{\sigma \vdash_{block} \Gamma(l) \Rightarrow (l', \sigma')}{\Vdash_{\Gamma} (l, \sigma) \rightarrow (l', \sigma')}$$

Semantic Values

$$\begin{array}{lll} l & \in \text{Labels[FCL]} & = \text{Block-Labels[FCL]} \cup (\{\mathsf{halt}\} \times \text{Values[FCL]}) \\ \sigma & \in \text{Stores[FCL]} & = \text{Variables[FCL]} \rightharpoonup \text{Values[FCL]} \\ \Gamma & \in \text{Block-Maps[FCL]} & = \text{Block-Labels[FCL]} \rightharpoonup \text{Blocks[FCL]} \\ s & \in \text{States[FCL]} & = \text{Labels[FCL]} \times \text{Stores[FCL]} \end{array}$$

Fig. 6. Operational semantics of FCL programs

to Values[FCL]. If we are evaluating a program p, we want σ to be defined for all variables occuring in p and undefined otherwise. That is, $dom(\sigma) = \text{Variables}[p]$ where $dom(\sigma)$ denotes set of variables for which σ is defined. If a σ satisfies this property, we say that σ *is compatible with* p. Each assignment statement may change the contents of the store. We write $\sigma[x \mapsto v]$ denote the store that is just like σ except that variable x now maps to v. Specifically,

$$\forall x \in \text{Variables[FCL]} \,.\, (\sigma[x' \mapsto v])(x) \stackrel{\text{def}}{=} \begin{cases} v & \textit{if } x = x' \\ \sigma(x') & \textit{if } x \neq x'. \end{cases}$$

For the meantime, we will assume that execution of a program p begins with an initial store σ_{init} where all variables occurring in p have been initialized to the value 0. More precisely, the *initial store* σ_{init} *for program* p is defined as follows:

$$\forall x \in \text{Variables[FCL]} \,.\, \sigma_{init}(x) \stackrel{\text{def}}{=} \begin{cases} 0 & \text{if } x \in \text{Variables}[p] \\ \textit{undefined} & \text{otherwise.} \end{cases}$$

- **Block maps:** In a typical implementation of FCL, one would use some sort of data structure to fetch a block b given b's label. We call such a data structure a *block map* because it maps labels to blocks. Formally, a block map $\Gamma \in \text{Block-Maps[FCL]}$ is a partial function from Block-Labels[FCL] to Blocks[FCL]. Thus, Γ is a lookup table for blocks using labels as keys. A block map Γ will be defined for all labels occurring in the program being described and undefined otherwise. For example, if Γ is a block map for the power program of Figure 1, then

$$\Gamma(\texttt{init}) \;=\; \texttt{init: result := 1;}\\ \texttt{goto test;}.$$

- **Computational states:** A *computational state* is a snap-shot picture of a point in a program's execution. It tells us (1) the current position of execution within the program (*i.e.*, the label of the current block), and (2) the current value of each of the program variables. Formally, a computational state $s \in \text{States[FCL]}$ is a pair (l, σ) where $l \in{=} \text{Labels[FCL]}$ and $\sigma \in \text{Stores[FCL]}$.

In Figure 6, a "big-step" semantics is used to define the evaluation of expressions, assignments, jumps, and blocks. The intuition behind the rules for these constructs is as follows.

- $\sigma \vdash_{expr} e \Rightarrow v$ means that under store σ, expression e evaluates to value v. Note that expression evaluation cannot change the value of the store.
- $\sigma \vdash_{assign} a \Rightarrow \sigma'$ means that that under store σ, the assignment a yields the updated store σ'.
- $\sigma \vdash_{assigns} al \Rightarrow \sigma'$ means that under the store σ, the list of assignments al yields the updated store σ'.
- $\sigma \vdash_{jump} j \Rightarrow l$ means that under the store σ, jump j will cause a transition to the block labelled l.

– $\sigma \vdash_{block} b \Rightarrow (l', \sigma')$ means that under the store σ, block b will cause a transition to the block labelled l' with updated store σ'.

The final rule of Figure 6 defines the Γ-indexed transition relation

$$\rightarrow_\Gamma \subseteq (\text{Labels[FCL]} \times \text{Stores[FCL]}) \times (\text{Labels[FCL]} \times \text{Stores[FCL]}).$$

This gives a "small-step" semantics for program evaluation. We will write

$$\Vdash_\Gamma (l, \sigma) \rightarrow (l', \sigma')$$

when

$$((l, \sigma), (l', \sigma')) \quad \in \quad \rightarrow_\Gamma .$$

The intuition is that there is a transition from state (l, σ) to state (l', σ') in the program whose block map is Γ. We will drop the Γ it is clear from the context.

Example derivation We can obtain the power program trace in Section 2.2 simply by following the rules in Figure 6. As an example, we will build a derivation that justifies the third transition:

$$\begin{aligned} &(\texttt{loop}, [\texttt{m} \mapsto 5, \texttt{n} \mapsto 2, \texttt{result} \mapsto 1]) \\ \rightarrow &(\texttt{test}, [\texttt{m} \mapsto 5, \texttt{n} \mapsto 1, \texttt{result} \mapsto 5]). \end{aligned}$$

We begin with the derivations for evaluation of expressions in block `loop` (see Figure 1) taking

$$\begin{aligned} \sigma &= [\texttt{m} \mapsto 5, \texttt{n} \mapsto 2, \texttt{result} \mapsto 1] \\ \sigma'' &= [\texttt{m} \mapsto 5, \texttt{n} \mapsto 1, \texttt{result} \mapsto 5] \end{aligned}$$

The following derivation ∇_1 specifies the evaluation of $\texttt{*(result n)}$.

$$\nabla_1 = \frac{\overline{\sigma \vdash_{expr} \texttt{result} \Rightarrow \sigma(\texttt{result})} \qquad \overline{\sigma \vdash_{expr} \texttt{m} \Rightarrow \sigma(\texttt{m})} \qquad [\![*]\!](1\ 5) = 5}{\sigma \vdash_{expr} \texttt{*(result m)} \Rightarrow 5}$$

The derivation is well-formed since $\sigma(\texttt{result}) = 1$ and $\sigma(\texttt{m}) = 5$. In the remainder of the derivations below, we omit application of the store to variables and simply give the resulting values (for example, we simply write 5 instead of $\sigma(\texttt{m})$).

Next, we give the derivation for the evaluation of the assignment statement $\texttt{result := *(result m)}$; (we repeat the conclusion of derivation ∇_1 for clarity):

$$\nabla_2 = \frac{\begin{array}{c}\nabla_1 \\ \sigma \vdash_{expr} \texttt{*(result m)} \Rightarrow 5\end{array}}{\sigma \vdash_{assign} \texttt{result := *(result m);} \Rightarrow \sigma[\texttt{result} \mapsto 5]}.$$

Now let

$$\sigma' \stackrel{\text{def}}{=} \sigma[\texttt{result} \mapsto 5] = [\texttt{m} \mapsto 5, \texttt{n} \mapsto 2, \texttt{result} \mapsto 5].$$

Similarly, we can build a derivation showing the evaluation of -(n 1):

$$\nabla_3 = \frac{\overline{\sigma' \vdash_{expr} \mathtt{n} \Rightarrow 2} \qquad \overline{\sigma' \vdash_{expr} 1 \Rightarrow [\![1]\!]} \qquad [\![-]\!](2\ 1) = 1}{\sigma' \vdash_{expr} \mathtt{-(n\ 1)} \Rightarrow 1}.$$

Next, we give the derivation for the evaluation of the assignment statement n := -(n 1);:

$$\nabla_4 = \frac{\begin{array}{c}\nabla_3 \\ \sigma' \vdash_{expr} \mathtt{-(n\ 1)} \Rightarrow 1\end{array}}{\sigma' \vdash_{assign} \mathtt{n := -(n\ 1);} \Rightarrow \sigma'[\mathtt{n} \mapsto 1]}$$

Note

$$\sigma'' \;=\; \sigma'[\mathtt{n} \mapsto 1] \;=\; [\mathtt{m} \mapsto 5, \mathtt{n} \mapsto 1, \mathtt{result} \mapsto 5].$$

Now we can build the following derivation showing the evaluation of the list of assignments containing exactly one assignment n := -(n 1);.

$$\nabla_5 = \frac{\begin{array}{c}\nabla_4 \\ \sigma' \vdash_{assign} \mathtt{n := -(n\ 1);} \Rightarrow \sigma'[\mathtt{n} \mapsto 1]\end{array} \qquad \overline{\sigma'' \vdash_{assign} \cdot \Rightarrow \sigma''}}{\sigma' \vdash_{assigns} \mathtt{n := -(n\ 1);} \Rightarrow \sigma''}.$$

Piecing together the previous derivations gives the following derivation for the assignments of the loop block.

$$\nabla_6 = \frac{\begin{array}{c}\nabla_2 \\ \sigma \vdash_{assign} \mathtt{result := *(result\ m);} \Rightarrow \sigma'\end{array} \qquad \begin{array}{c}\nabla_5 \\ \sigma' \vdash_{assigns} \mathtt{n := -(n\ 1);} \Rightarrow \sigma''\end{array}}{\sigma \vdash_{assigns} \mathtt{result := *(result\ m); n := -(n\ 1);} \Rightarrow \sigma''}$$

Now we can construct the derivation for the evaluation of the loop block. Here, we let $al = \mathtt{result := *(result\ m); n := -(n\ 1);}$.

$$\nabla_7 = \frac{\begin{array}{c}\nabla_6 \\ \sigma \vdash_{assigns} al \Rightarrow \sigma''\end{array} \qquad \overline{\sigma'' \vdash_{jump} \mathbf{goto}\ \mathtt{test;} \Rightarrow \mathtt{test}}}{\sigma \vdash_{block} \Gamma(\mathtt{loop}) \Rightarrow (\mathtt{test}, \sigma'')}$$

Now we can construct the derivation for the transition from the loop block to the test block.

$$\frac{\begin{array}{c}\nabla_7 \\ \sigma \vdash_{block} \Gamma(\mathtt{loop}) \Rightarrow (\mathtt{test}, \sigma'')\end{array}}{\Vdash_\Gamma (\mathtt{loop}, \sigma) \to (\mathtt{test}, \sigma'')}$$

Execution properties Finally, we can formalize a notion of FCL program evaluation.

Definition 2. *Let $p = (x_1 \ldots x_n)\ (l)\ b^+$ be a well-formed program, let σ_{init} be an initial store for p, and let $v_1, \ldots, v_n \in$ Values[FCL]. Define program evaluation $[\![\cdot]\!]$ as follows:*

$$[\![p]\!](v_1, \ldots, v_n) \stackrel{\text{def}}{=} \begin{cases} v & \text{if } (l, \sigma_{init}[x_1 \mapsto v_1, \ldots, x_n \mapsto v_n]) \to^* (\langle \mathsf{halt}, v \rangle, \sigma) \\ \text{undefined} & \text{otherwise.} \end{cases}$$

Below are some possible properties of transitions and execution sequences (based on [14]).

Definition 3. *Let* $p \in$ Programs[FCL] *be a well-formed program, and let* Γ *be a block map for* p*.*

- *A block* $l \in$ Block-Labels[p] *is* one-way *if*

$$(l, \sigma_1) \rightarrow (l', \sigma') \text{ and } (l, \sigma_2) \rightarrow (l'', \sigma'')$$

  implies $l' = l''$*.*
- *The state* (l, σ) *is* terminal *if there does not exit a* l' *and* σ' *such that*

$$(l, \sigma) \rightarrow (l', \sigma').$$

- *The program* p *is* deterministic *if for all states* (l, σ)*,*

$$(l, \sigma) \rightarrow (l', \sigma') \text{ and } (l, \sigma) \rightarrow (l'', \sigma'')$$

  implies $l' = l''$ *and* $\sigma' = \sigma''$*.*

Intuitively, a block is one-way if it does not end with an **if** jump.

2.3 Exercises

1. Write an FCL program to compute the factorial function.
2. Give the execution trace for the prime number generator of Figure 7 with input `3`. In the generator program, `%` implements the *mod* operator, and `/` implements integer division.
3. Given the execution trace for the prime number generator of Figure 7 with input `3` constructed in the previous exercise, give the derivation justifying the first transition from block `prime` to block `next`.
4. Prove that all well-formed programs $p \in$ Programs[FCL] are deterministic.
5. Add lists and associated operations (`car`, `cdr`, `cons`, *etc.*) to FCL. You might also consider adding S-expressions as in Scheme or LISP (*i.e.*, include symbols and a `quote` construct for constructing literal data).
6. **Project:** Following the operational semantics definition, program the FCL interpreter in the language of your choice.

2.4 Further reading

As noted, the presentation of the FCL language syntax is based on that of Gomard and Jones [11, 15]. The operational semantics presentation is original, and it is designed to match closely the more abstract presentation of program points and transitions given by Jones in his work "The Essence of Program Transformation by Partial Evaluation and Driving" [14]. Winskel's textbook [28] on programming language semantics gives a nice introduction to operational semantics.

```
(n)
(start)

  start:  m := 0;
          s := 2;
          k := 2;
          goto loop;

  loop:   if <(k +(/(s 2) 1) then check else prime;

  check:  d := %(s k);
          k := +(k 1);
          if =(d 0) then next else loop;

  prime:  m := +(m 1);
          p := s;
          if =(m n) then done else next;

  next:   k := 2;
          s := +(s 1);
          goto loop;

  done:   return p;
```

Fig. 7. An FCL program to generate the n^{th} prime

3 Online Partial Evaluation

In this chapter, we see our first form of program specialization: *online partial evaluation*. We begin by considering how the notion of an execution trace changes when we move from the situtation where we have complete data to a situation where some data is unknown. We discuss how specialization might proceed for the power program. After this motivation, we systematically introduce the basic components of online partial evaluation.

3.1 A motivating example

A partial evaluator takes values for a portion $\vec{x_s}$ of a program p's input parameters $(\vec{x_s}, \vec{x_d})$ and tries to perform as much computation as possible. Code for computations that depend on the remaining parameters $\vec{x_d}$ is placed in a residual program. Thus, a partial evaluator blends *interpretion* and *compilation*: it interprets (*i.e.*, evaluates) constructs that depend on $\vec{x_s}$, and emits code for constructs that may depend on $\vec{x_d}$.

As an example, this section considers what would be involved in specializing the FCL power program of Figure 1. One may first consider how the execution trace of Section 2.2 would have to change when considering partial input data only. Intuitively, we expect a trace where some values in the store are unknown. Some of the expressions, assignments, and jumps in blocks may be computed because they depend only on known data. However, others cannot be computed because the depend on unknown data. Once we find all the states that could be reached with the partial data, we then consider how we could partially compute each block based on the partial data.

In summary, partial evaluation can be viewed as a three-step process [15, Chapter 4].

1. *Collect all reachable states:* Starting from an initial state (l, σ) where σ contains only partial data, collect all reachable states (l_i, σ_i) into a set $\mathcal{S}$
2. *Program point specialization:* For each reachable state $(l_i, \sigma_i) \in \mathcal{S}$, place a new version of block l that is specialized with respect to σ_i in the residual program.
3. *Transition compression:* Optimize the residual program by merging blocks connected by trivial **goto** jumps.

Although this three-step view is very enlightening, in practice, most partial evaluators carry out these steps in a single phase. After giving several examples using the three-step view, we will give a formalization of on-line partial evaluation that uses a single phase.

Abstract traces Section 2.2 showed the execution of trace of the power program computing 5^2. Let us consider what a trace might look like if we only know

the exponent parameter `n` to be 2 but know nothing about the value `m`. We use the special tag D to represent an unknown value.

$$
\begin{aligned}
& (\texttt{init}, [\texttt{m} \mapsto D, \texttt{n} \mapsto 2, \texttt{result} \mapsto 0]) && (1)\\
\rightarrow\ & (\texttt{test}, [\texttt{m} \mapsto D, \texttt{n} \mapsto 2, \texttt{result} \mapsto 1]) && (2)\\
\rightarrow\ & (\texttt{loop}, [\texttt{m} \mapsto D, \texttt{n} \mapsto 2, \texttt{result} \mapsto 1]) && (3)\\
\rightarrow\ & (\texttt{test}, [\texttt{m} \mapsto D, \texttt{n} \mapsto 1, \texttt{result} \mapsto D]) && (4)\\
\rightarrow\ & (\texttt{loop}, [\texttt{m} \mapsto D, \texttt{n} \mapsto 1, \texttt{result} \mapsto D]) && (5)\\
\rightarrow\ & (\texttt{test}, [\texttt{m} \mapsto D, \texttt{n} \mapsto 0, \texttt{result} \mapsto D]) && (6)\\
\rightarrow\ & (\texttt{end}, [\texttt{m} \mapsto D, \texttt{n} \mapsto 0, \texttt{result} \mapsto D]) && (7)\\
\rightarrow\ & (\texttt{halt}, [\texttt{m} \mapsto D, \texttt{n} \mapsto 0, \texttt{result} \mapsto D]) && (8)
\end{aligned}
$$

This trace is called an *abstract trace* because the data value for `m` has been abstracted by the symbol D.

Here is the intuition behind each step in the trace.

- **Line 1**: The trace begins at the `init` block with the value of `m` marked as unknown. In `init`, `result` is initialized to `1`. This is reflected in the value of the store when `test` is reached for the first time.
- **Line 2**: The value of `n` is known, and so we can compute the test `<(n 1)` (it is false) and move to `loop`.
- **Line 3**: The expression `*(result m)` cannot be computed since `m` is unknown. Therefore, `result` must be assigned a value of unknown. Since `n` is known, the result of decrementing `n` is placed in the store. Thus we arrive at `test` with $\texttt{n} \mapsto 1$ and $\texttt{result} \mapsto D$.
- **Line 4**: The value of `n` is known, and so we can compute the test `<(n 1)` (it is false) and move to `loop`.
- **Line 5**: The expression `*(result m)` cannot be computed since `m` and `result` are unknown. Thus, `result` is assigned a value of unknown. The result of decrementing `n` is placed in the store. Thus we arrive at `test` with $\texttt{n} \mapsto 0$ and $\texttt{result} \mapsto D$.
- **Line 6**: The value of `n` is known, and so we can compute the test `<(n 1)` (it is true) and move to `end`.
- **Line 7**: The block `end` has no assignments and so we move to the terminal state with the store unchanged. We omit a value in the `halt` label since the value of the `result` variable is unknown.

Figure 8 shows the computation tree path corresponding to the trace above. It is important to note that for any value $v \in$ Values[FCL], the path of a trace with initial store

$$\sigma_{init} \quad = \quad [\texttt{m} \mapsto v, \texttt{n} \mapsto 2, \texttt{result} \mapsto 0]$$

must lie within the shaded area of the computation tree. This gives us insight about the structure of the residual program that the partial evaluator will produce.

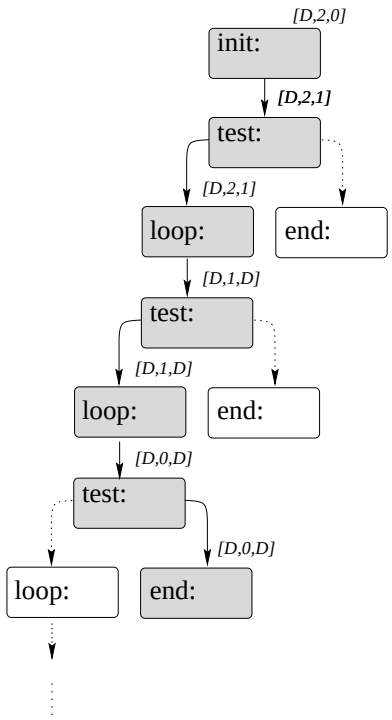

Fig. 8. Execution trace's path through the computation tree

The partial evaluator's task is to take the value 2 for n and produce an optimized residual program that is capable of handling any value v for m. In what sense is the residual program optimized?

- Since the trace using σ_{init} will always lie in the shaded area of the tree, the residual program can safely avoid having blocks corresponding to the unshaded nodes of the tree. It needs only to include blocks for the shaded portion of the tree. In fact, we will see that the next step in partial evaluation constructs a residual program that has exactly the same structure as the shaded portion of the computation tree.
- Notice that we will in general have several different reachable states $(l, \sigma), \ldots, (l, \sigma')$ for each block l. For example, in the trace above there are multiple states for blocks `test` and `loop`. Since the partial evaluator has collected all reachable states, it can customize each node in the shaded area of computation tree with the known information in the store that is flowing into that node. For example, we can create two versions of the loop block: one specialized to $\texttt{n} \mapsto 2$ and $\texttt{result} \mapsto 1$, and another specialized to $\texttt{n} \mapsto 1$.

In other words, the partial evaluator uses the information that n has initial value of 2

- to prune unnecessary branches from the computation tree (those leading to unshaded area), and
- to specialize each reachable block (those in the shaded area) with respect to the different store values that are guaranteed to be flowing into that block.

Program point specialization For each reachable state (l, σ), we create a residual block labeled l' by specializing block l with respect to σ. For simplicity, we will simply use the pair (l, σ) as the value of label l'. For example, the residual block for state $(\texttt{init}, [\texttt{m} \mapsto D, \texttt{n} \mapsto 2, \texttt{result} \mapsto 0])$ will have label `(init, [D,2,0])` (we drop the variables in the store for simplicity). In practice, a partial evaluator will generate a fresh label l' for each reachable state (l, σ) and maintain a table associating generated labels with reachable states.[1]

- State $(\texttt{init}, [\texttt{m} \mapsto D, \texttt{n} \mapsto 2, \texttt{result} \mapsto 0])$: In the `init` block, the assignment `result := 1;` does not depend on the dynamic data m so it can be completely executed as indicated in the construction of the trace. Customizing the `init` block involves computing the only assignment and customizing the **goto** so that we jump to the customized block for the next state in the trace.

  ```
  (init, [D,2,0]): goto (test, [D,2,1]);
  ```

- State $(\texttt{test}, [\texttt{m} \mapsto D, \texttt{n} \mapsto 2, \texttt{result} \mapsto 1])$: Since the value of n is known, the test expression of the **if** construct can be evaluated and the appropriate branch executed. Since in this state, $\texttt{n} \mapsto 2$, the branch to be taken is the false branch.

[1] This is the case in our Scheme implementation.

```
(test, [D,2,1]): goto (loop, [D,2,1]);
```

– State (`loop`, $[\texttt{m} \mapsto D, \texttt{n} \mapsto 2, \texttt{result} \mapsto 1]$): The value of `m` is not known, and so the first assignment must be *residualized*.

```
result := *(1 m);
```

Note that the variable reference `result` on the right-hand side of the assignment can be replaced by its value (since it is known). The following assignment to `n` can be computed since `n` is known. Finally, we customize the jump as usual. The resulting block is as follows.

```
(loop, [D,2,1]): result := *(1 m);
                 goto (test, [D,1,D]);
```

– State (`test`, $[\texttt{m} \mapsto D, \texttt{n} \mapsto 1, \texttt{result} \mapsto D]$): Again, the test expression in the conditional jump can be computed. The resulting specialized block is given below.

```
(test, [D,1,D]): goto (loop, [D,1,D]);
```

– State (`loop`, $[\texttt{m} \mapsto D, \texttt{n} \mapsto 1, \texttt{result} \mapsto D]$):
Since the value of `m` is not known, the first assignment must be *residualized* as before.

```
result := *(result m);
```

However, in contrast to the previous entry to `loop`, `result` is not known this time and so the variable name is residualized on the right-hand side of the assignment. The following assignment to `n` can be computed since `n` is known. Finally, we customize the jump as usual. The result is as follows.

```
(loop, [D,1,D]): result := *(result m);
                 goto (test, [D,0,D]);
```

– State (`test`, $[\texttt{m} \mapsto D, \texttt{n} \mapsto 0, \texttt{result} \mapsto D]$): Again, the test expression in the conditional jump can be computed. However, this time the test expression is true, and so control is transferred to the appropriately specialized `end` block. The resulting specialized block is given below.

```
(test, [D,0,D]): goto (end, [D,0,D]);
```

– State (`end`, $[\texttt{m} \mapsto D, \texttt{n} \mapsto 0, \texttt{result} \mapsto D]$): The `end` block contains nothing but the **return** `result`; jump (which cannot be further specialized). The resulting block is as follows.

```
(end, [D,0,D]): return result;
```

Figure 9 gives the residual program that results from specializing the FCL power program with respect to $\texttt{n} \mapsto 2$. The residual program contains all the specialized blocks produced above, as well the declaration of the remaining parameter `m` and the label of the initial block (`init`, $[\texttt{D}, 2, 0]$).

Figure 10 presents the flowchart diagram for the specialized program of Figure 9. As promised, the structure of the residual program matches the structure of the shaded area of the computation tree in Figure 8.

```
(m)
((init,[D,2,0]))

  (init,[D,2,0]): goto (test,[D,2,1]);

  (test,[D,2,1]): goto (loop,[D,2,1]);

  (loop,[D,2,1]): result := *(1 m);
                  goto (test,[D,1,D]);

  (test,[D,1,D]): goto (loop,[D,1,D]);

  (loop,[D,1,D]): result := *(result m);
                  goto (test,[D,0,D]);

  (test,[D,0,D]): goto (end,[D,0,D]);

  (end,[D,0,D]):  return result;
```

Fig. 9. The FCL power program specialized to $\mathtt{n} = 2$.

Transition compression There is a tight relationship between the original power program and the specialized version in Figure 9: there is a transition in the source program $(l, \sigma) \rightarrow (l', \sigma')$ iff there is a corresponding transition from block (l, σ) to block (l', σ') in the residual program. For example,

$$\begin{aligned}&(\mathtt{init}, [\mathtt{m} \mapsto 5, \mathtt{n} \mapsto 2, \mathtt{result} \mapsto 0])\\ \rightarrow\ &(\mathtt{test}, [\mathtt{m} \mapsto 5, \mathtt{n} \mapsto 2, \mathtt{result} \mapsto 1])\end{aligned}$$

in the source program and

$$\begin{aligned}&((\mathtt{init}, [\mathtt{D},2,0]), [\mathtt{m} \mapsto 5, \mathtt{result} \mapsto 0])\\ \rightarrow\ &((\mathtt{test}, [\mathtt{D},2,1]), [\mathtt{m} \mapsto 5, \mathtt{result} \mapsto 1])\end{aligned}$$

in the residual program. However, the transition in the residual program isn't that interesting since it only involves chaining through a **goto**.

More generally, the control flow of the specialized program in Figure 9 is trivial in the sense that it does not contain any branches or loops. Thus, the **goto** jumps that exist are not necessary. Figure 11 shows a more natural specialized program where the transitions associated with the **goto** jumps have been eliminated.

Transition compression yields programs that are much more compact. However, one does not want to apply it indiscriminately — doing so may cause the partial evaluator to loop. For example, trying to eliminate all **goto**'s in the program below leads to infinite unfolding.

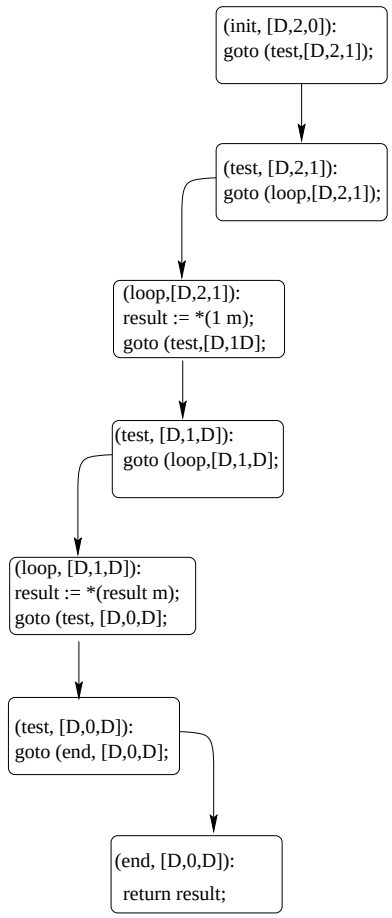

Fig. 10. Flowchart for residual power program

```
(m)
((init,[D,2,0]))

  (init,[D,2,0]): result := *(1 m);
                  result := *(result m);
                  return result;
```

Fig. 11. The specialized FCL power program after transition compression

```
l1: a := b + 1;
    goto l2;

l2: b := a + 1;
    goto l1;
```

Various heuristics are applied to obtain effective and non-overly-aggressive transition compression. We will discuss two strategies in Section 3.3.

Assessment We have seen the online partial evaluation can be viewed as a three step process: (1) collect all reachable states, (2) program point specialization, (3) transition compression.

There were some opportunities for specialization that we did not pursue.

- *Algebraic properties:* The multiplication operation in

  ```
  result := *(1 m);
  ```

  could be further reduced to

  ```
  result := m;
  ```

  using the knowledge that 1 is the identity element for multiplication. However, most partial evaluators do not incorporate such transformations since they will be taken care of by an optimizing compiler.
- *Unfolding variables:* In the following code,

  ```
  result := *(1 m);
  result := *(result m);
  ```

  this first assignment could be unfolded into the second to obtain

  ```
  result := *(*(1 m) m);
  ```

  This gives one less assignment operation. However, applying this transformation indiscriminately can cause code-size blowup and duplicate computation. For example, transforming the code below

  ```
  x := +(y 2);
  z := *(*(x x) x);
  ```

  yields

  ```
  z := *(*(+(y 2) +(y 2)) +(y 2));
  ```

  One can use a counting analysis to tell when unfolding will not cause code explosion. As with the use of algebraic identities above, it is generally best to leave these types of transformations to the compiler.

3.2 Handling branching and infinite abtract traces

The structure of the abstract trace of the power program in the preceding section had a particularly simple structure: it was finite and non-branching. In this section, we see how to handle more complicated traces.

Handling branching traces When specializing the power program in the preceding section, we were always able to decide which branch to execute in a conditional jump because the tests in the conditional jumps only depended on static data. Thus, the trace that we constructed when collecting all reachable configurations corresponded to a single non-branching path through the computation tree (see Figure 8).

In general, specializing a program may lead to a conditional whose test depends on dynamic data. In this case, we cannot decide which branch of the conditional to execute at specialization time — we have to consider the possibility that *either* branch of the conditional can be executed when the residual program is run. Thus, we need to residualize the conditional, and generate abstract traces for *both* branches of the condition. This leads to traces that are not just single non-branching paths, but instead to traces that branch at each dynamic conditional.

When generating an abstract trace for the program of Figure 12 using the initial store $[\mathtt{a} \mapsto D, \mathtt{b} \mapsto 2]$, the conditional in `init` cannot be decided. Therefore, we have two possible successor states $(\mathtt{less}, [\mathtt{a} \mapsto D, \mathtt{b} \mapsto 2])$ and $(\mathtt{more}, [\mathtt{a} \mapsto D, \mathtt{b} \mapsto 2])$, and we must continue building the trace from both of these.

In coding up an algorithm to generate branching abstract traces, we can handle branching traces using a *pending set* $\mathcal{P}$ of states that have yet to be processed. Managing the set in a LIFO manner gives a depth-first generation of the trace. For example, the steps in depth-first generation are as follows.

Processing	*Pending Set* $\mathcal{P}$
	$\{(\mathtt{init}, [\mathtt{a} \mapsto D, \mathtt{b} \mapsto 2])\}$
$(\mathtt{init}, [\mathtt{a} \mapsto D, \mathtt{b} \mapsto 2])$	$\{(\mathtt{less}, [\mathtt{a} \mapsto D, \mathtt{b} \mapsto 2]), (\mathtt{more}, [\mathtt{a} \mapsto D, \mathtt{b} \mapsto 2])\}$
$(\mathtt{less}, [\mathtt{a} \mapsto D, \mathtt{b} \mapsto 2])$	$\{(\mathtt{done}, [\mathtt{a} \mapsto 2, \mathtt{b} \mapsto 2]), (\mathtt{more}, [\mathtt{a} \mapsto D, \mathtt{b} \mapsto 2])\}$
$(\mathtt{done}, [\mathtt{a} \mapsto 2, \mathtt{b} \mapsto 2])$	$\{(\mathtt{more}, [\mathtt{a} \mapsto D, \mathtt{b} \mapsto 2])\}$
$(\mathtt{more}, [\mathtt{a} \mapsto D, \mathtt{b} \mapsto 2])$	$\{(\mathtt{done}, [\mathtt{a} \mapsto 0, \mathtt{b} \mapsto 2])\}$
$(\mathtt{done}, [\mathtt{a} \mapsto 0, \mathtt{b} \mapsto 2])$	$\emptyset$

Here, we begin with the initial state in $\mathcal{P}$ and continue until $\mathcal{P}$ is empty. Note, that we do not need to insert `halt` states into $\mathcal{P}$ since we know that they will have no successors. We have described $\mathcal{P}$ as a *set* because one can process the pending states in any order (*e.g.*, LIFO or FIFO).

Handling infinite traces with repeating states Figure 13 gives a flowchart program that will compute $m \times n \times term$. When this program is specialized wrt $n = 2$, the test at block `test-n` can be decided, but the test at block

```
(a b)
(init)

  init: if <(a 10)
           then less
           else greater;

  less: a := b;
        goto done;

  more: a := 0;
        goto done;

  done: return +(a b);
```

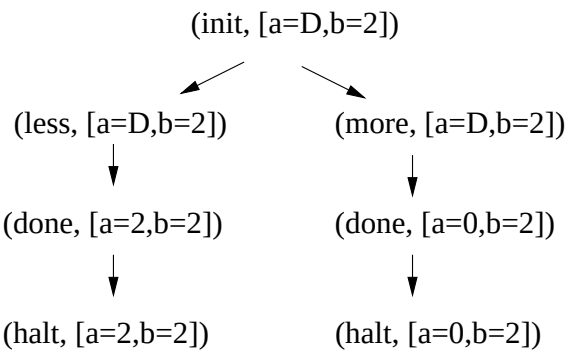

Fig. 12. FCL program with branching abstract trace for $[\mathtt{a} \mapsto D, \mathtt{b} \mapsto 2]$

```
(m n term)
(init)

  init:   result := term;
          save-n := n;
          goto test-m;

  test-m: if <(0 m)
           then test-n
           else done-m;

  test-n: if <(0 n)
           then loop
           else done-n;

  loop:   result := +(result term);
          n := -(n 1);
          goto test-n;

  done-n: m := -(m 1);
          n := save-n;
          goto test-m;

  done-m: result := -(result term);
          return result;
```

Fig. 13. An FCL program to compute $m \times n \times term$

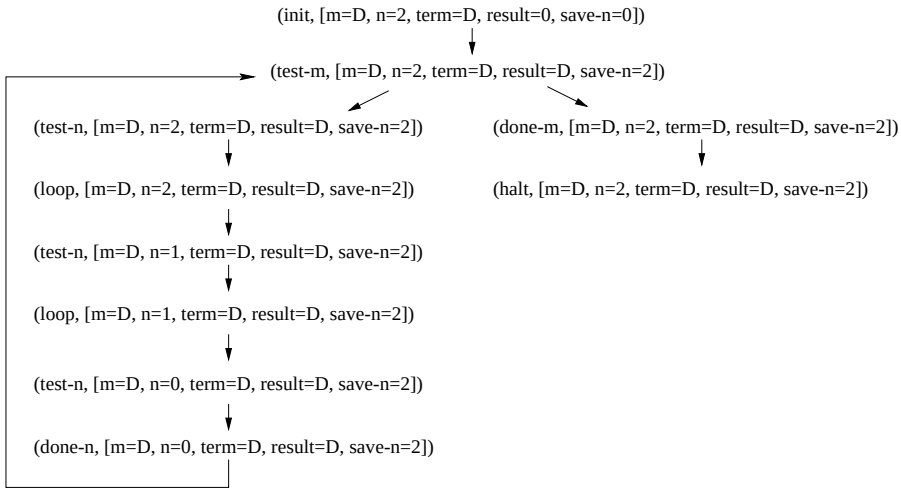

Fig. 14. Trace of term multiplier program specialization ($n \mapsto 2$)

`test-m` cannot. Thus, each time we process `test-m` we will generate two following configurations: one for `test-n` and one for `done-m`.

Figure 14 shows the trace that generates reachable configurations when specializing the program with respect to $n \mapsto 2$. Besides the branching path, there is another important difference from the power program specialization. In this example, if one always follows the true branch from the `test-m` node, a `halt` node will never be reached. Conceptually, a trace where only true branches are chosen from `test-m` is infinite, but in fact, a point will be reached where the configurations in the trace start to repeat themselves — that is, the trace contains a cycle. When we come to a configuration that we have already seen before, we can safely stop that branch of the trace because we know we have included all reachable configurations along that branch.

To deal with *branching traces* and *potentially infinite traces with repeating states*, partial evaluators include two data structures:

- a "seen before" set $\mathcal{S}$ containing the configurations that have already be processed, and
- a "pending" set $\mathcal{P}$ containing the configurations that are waiting to be processed.

Taking $\mathcal{R}$ to be a list of residual program blocks, and $(l_{init}, \sigma_{init})$ to be the initial configuration for specialization, the basic algorithm for online partial evaluation (without transition compression) is given in Figure 15.

Partial evaluation begins with an empty seen-before set $\mathcal{S}$ and the initial configuration in the pending set $\mathcal{P}$. While the pending set is not empty, we repeatedly pull a configuration off the pending set, create a specialized block for

```
S := ∅;
P := {(l_init, σ_init)};
while P is not empty do
    (l, σ) := remove(P);
    if (l, σ) ∉ S then
        insert((l, σ),S);
        /* let b' be the result of specializing the */
        /* block labelled l and let N be the new */
        /* configurations generated. */
        insert(b',R);
        for each (l', σ') ∈ N do
            if l' ≠ halt then
                insert((l', σ'),P);
            endif
        endfor
    endif
endwhile
```

Fig. 15. Outline of algorithm for online partial evaluation

that configuration, get the new configurations N that result from processing the current block l, and we add these to the pending set. The function *remove*($\mathcal{P}$) updates $\mathcal{P}$ by removing a state (l, σ) and returns the state as a result. Similarly, *insert* updates a set or list by adding the given element.

From an abstract view, the algorithm collects the reachable configurations from the computation tree and generates a specialized block for configuration. The pending set $\mathcal{P}$ holds a frontier of the tree of reachable nodes, and the $\mathcal{S}$ holds all the nodes in the tree above the frontier.

Notice that each iteration of the **while** loop transforms the pending set $\mathcal{P}$, the seen before set $\mathcal{S}$, and the residual program $\mathcal{R}$. In the next section, we will give an abstract presentation of the algorithm of Figure 15, and each iteration of the **while** loop will correspond to a transition

$$\langle \mathcal{P} \, , \, \mathcal{S} \, , \, \mathcal{R} \rangle \; \longmapsto \; \langle \mathcal{P}' \, , \, \mathcal{S}' \, , \, \mathcal{R}' \rangle .$$

Using this notation, the transition sequence below shows the first few iterations of the online partial evaluation algorithm when specializing the term multiplier program with respect to $\mathtt{n} \mapsto 2$. Here, b_1, b_2, b_3, and b_4 represent the specialized versions of `init`, `test-m`, `test-n`, and `loop`, respectively (generating these

specialized versions is left as an exercise for the reader).

$$\left\langle \begin{array}{l} \{(\texttt{init}, [\texttt{m} \mapsto D, \texttt{n} \mapsto 2, \texttt{term} \mapsto D, \texttt{result} \mapsto 0, \texttt{save-n} \mapsto 0])\}, \\ \emptyset, \\ . \end{array} \right\rangle$$

$$\longmapsto \left\langle \begin{array}{l} \{(\texttt{test-m}, [\texttt{m} \mapsto D, \texttt{n} \mapsto 2, \texttt{term} \mapsto D, \texttt{result} \mapsto D, \texttt{save-n} \mapsto 2])\}, \\ \{(\texttt{init}, [\texttt{m} \mapsto D, \texttt{n} \mapsto 2, \texttt{term} \mapsto D, \texttt{result} \mapsto 0, \texttt{save-n} \mapsto 0])\}, \\ b_1 \end{array} \right\rangle$$

$$\longmapsto \left\langle \begin{array}{l} \{(\texttt{test-n}, [\texttt{m} \mapsto D, \texttt{n} \mapsto 2, \texttt{term} \mapsto D, \texttt{result} \mapsto D, \texttt{save-n} \mapsto 2]), \\ \quad (\texttt{done-m}, [\texttt{m} \mapsto D, \texttt{n} \mapsto 2, \texttt{term} \mapsto D, \texttt{result} \mapsto D, \texttt{save-n} \mapsto 2])\}, \\ \{(\texttt{init}, [\texttt{m} \mapsto D, \texttt{n} \mapsto 2, \texttt{term} \mapsto D, \texttt{result} \mapsto 0, \texttt{save-n} \mapsto 0]), \\ \quad (\texttt{test-m}, [\texttt{m} \mapsto D, \texttt{n} \mapsto 2, \texttt{term} \mapsto D, \texttt{result} \mapsto D, \texttt{save-n} \mapsto 2])\}, \\ b_1\ b_2 \end{array} \right\rangle$$

$$\longmapsto \left\langle \begin{array}{l} \{(\texttt{loop}, [\texttt{m} \mapsto D, \texttt{n} \mapsto 2, \texttt{term} \mapsto D, \texttt{result} \mapsto D, \texttt{save-n} \mapsto 2]), \\ \quad (\texttt{done-m}, [\texttt{m} \mapsto D, \texttt{n} \mapsto 2, \texttt{term} \mapsto D, \texttt{result} \mapsto D, \texttt{save-n} \mapsto 2])\}, \\ \{(\texttt{init}, [\texttt{m} \mapsto D, \texttt{n} \mapsto 2, \texttt{term} \mapsto D, \texttt{result} \mapsto 0, \texttt{save-n} \mapsto 0]), \\ \quad (\texttt{test-m}, [\texttt{m} \mapsto D, \texttt{n} \mapsto 2, \texttt{term} \mapsto D, \texttt{result} \mapsto D, \texttt{save-n} \mapsto 2]), \\ \quad (\texttt{test-n}, [\texttt{m} \mapsto D, \texttt{n} \mapsto 2, \texttt{term} \mapsto D, \texttt{result} \mapsto D, \texttt{save-n} \mapsto 2])\}, \\ b_1\ b_2\ b_3 \end{array} \right\rangle$$

$$\longmapsto \left\langle \begin{array}{l} \{(\texttt{test-n}, [\texttt{m} \mapsto D, \texttt{n} \mapsto 1, \texttt{term} \mapsto D, \texttt{result} \mapsto D, \texttt{save-n} \mapsto 2]), \\ \quad (\texttt{done-m}, [\texttt{m} \mapsto D, \texttt{n} \mapsto 2, \texttt{term} \mapsto D, \texttt{result} \mapsto D, \texttt{save-n} \mapsto 2])\}, \\ \{(\texttt{init}, [\texttt{m} \mapsto D, \texttt{n} \mapsto 2, \texttt{term} \mapsto D, \texttt{result} \mapsto 0, \texttt{save-n} \mapsto 0]), \\ \quad (\texttt{test-m}, [\texttt{m} \mapsto D, \texttt{n} \mapsto 2, \texttt{term} \mapsto D, \texttt{result} \mapsto D, \texttt{save-n} \mapsto 2]), \\ \quad (\texttt{test-n}, [\texttt{m} \mapsto D, \texttt{n} \mapsto 2, \texttt{term} \mapsto D, \texttt{result} \mapsto D, \texttt{save-n} \mapsto 2]), \\ \quad (\texttt{loop}, [\texttt{m} \mapsto D, \texttt{n} \mapsto 2, \texttt{term} \mapsto D, \texttt{result} \mapsto D, \texttt{save-n} \mapsto 2]), \}, \\ b_1\ b_2\ b_3\ b_4 \end{array} \right\rangle$$

$$\longmapsto \ldots$$

3.3 Semantics of online partial evaluation

We now formalize the online partial evaluation outlined in Figure 15. We do this using an operational semantics that follows the same structure as the interpreter for FCL given in Figure 6 of Chapter 2.

To understand how the semantics changes when moving from interpretation to partial evaluation, notice that processing each expression in Section 3.1 yielded either a value (if the expression depended on only static variables) or a chunk of code (if the expression depended on dynamic variables). For example, in the block

```
loop: result := *(result m);
      n := -(n 1);
      goto test;
```

processing the expression `-(n 1)` always produced a value, whereas processing the expression `*(result m)` always produced a chunk of code. To capture this, when our partial evaluator will always produce one of the following types of results (which we call *pe-values*) when processing expressions.

- Processing an expression that only depends on static variables returns a pair $\langle S\,,\,v\rangle$ where v is the resulting value.
- Processing an expression that depends on a dynamic variable returns a pair $\langle D\,,\,e\rangle$ where e is the resulting piece of code.

Semantic values Figure 16 presents the definition of partial evaluation for expressions and assignments. We first discuss the semantic values. As noted above, the essential difference between the semantic values used in interpretation and those used in partial evaluation is that partial evaluation manipulates pe-values: tagged objects $\langle S\,,\,v\rangle, \langle D\,,\,e\rangle \in$ PE-Values[FCL].

A store σ will hold pe-values as well.

- If a variable x has a known value v, then we will have $\sigma(x) = \langle S\,,\,v\rangle$.
- If a variable x has an unknown value, then we will have $\sigma(x) = \langle D\,,\,x\rangle$.

This ensures that whenever we look x up in σ, we will either get the appropriate value for x (v), or the appropriate piece of code for x (x). For example, we would use the store below as the initial store when specializing the power program to $n \mapsto 2$:

$$[\mathtt{m} \mapsto \langle D\,,\,\mathtt{m}\rangle, \mathtt{n} \mapsto \langle S\,,\,2\rangle, \mathtt{result} \mapsto \langle S\,,\,0\rangle].$$

The other semantic domains are the same as in FCL interpretation.

Expressions The judgement giving the partial evaluation of expressions has the form $\sigma \vdash_{expr} e \Rightarrow w$. The value of value of constant c is always known, and so $\langle S\,,\,[\![c]\!]\rangle$ is returned (recall that $[\![c]\!] \in$ Values[FCL] is the constant's semantic value). Given the above strategy of representing variable values in the store, $\sigma(x)$ always produces the correct result for partial evaluating a variable.

The rules for operations are the most interesting. The first operation rule covers the case where all the argument values are known. In this case, we can simply apply the operation to the argument values and return the resulting value (marked as static).

The second rule covers the case where at least one of the argument values is unknown (that is, at least one of the pe-values has the form $\langle D\,,\,e''\rangle$). In this case, the operation cannot be evaluated, and so it must be reconstructed ($\langle D\,,\,o(e'_1 \ldots e'_n)\rangle$ is returned). However, there is still a subtle issue regarding what code to generate for the arguments e'_i. If specializing e_k yielded $\langle D\,,\,e''_k\rangle$, then clearly e''_k should be the code that is residualized for e_k. But if e_k yielded $\langle S\,,\,v''_k\rangle$, we need to return the *code* corresponding to v''_k. That is, we need to return the constant that computes to v''_k

Expressions

$$\frac{}{\sigma \vdash_{expr} c \Rightarrow \langle S\,,\, [\![c]\!] \rangle} \qquad \frac{}{\sigma \vdash_{expr} x \Rightarrow \sigma(x)}$$

$$\frac{\sigma \vdash_{expr} e_i \Rightarrow \langle S\,,\, v_i \rangle \qquad [\![o]\!](v_1 \ldots v_n) = v}{\sigma \vdash_{expr} o(e_1 \ldots e_n) \Rightarrow \langle S\,,\, v \rangle}$$

$$\frac{\sigma \vdash_{expr} e_i \Rightarrow w_i \qquad w_i \Uparrow e_i' \qquad \exists j \in \{1..n\}\,.\, w_j = \langle D\,,\, e \rangle}{\sigma \vdash_{expr} o(e_1 \ldots e_n) \Rightarrow \langle D\,,\, o(e_1' \ldots e_n') \rangle}$$

Lifting

$$\frac{}{\langle S\,,\, v \rangle \Uparrow [\![v]\!]^{-1}} \qquad \frac{}{\langle D\,,\, e \rangle \Uparrow e}$$

Assignments

$$\frac{\sigma \vdash_{expr} e \Rightarrow \langle S\,,\, v \rangle}{\sigma \vdash_{assign} x := e; \Rightarrow \langle\, \sigma[x \mapsto \langle S\,,\, v \rangle]\,,\, []\, \rangle}$$

$$\frac{\sigma \vdash_{expr} e \Rightarrow \langle D\,,\, e' \rangle}{\sigma \vdash_{assign} x := e; \Rightarrow \langle\, \sigma[x \mapsto \langle D\,,\, x \rangle]\,,\, [x := e';]\, \rangle}$$

$$\frac{}{\sigma \vdash_{assigns} \cdot \Rightarrow \langle\, \sigma\,,\, []\, \rangle}$$

$$\frac{\sigma \vdash_{assign} a \Rightarrow \langle\, \sigma'\,,\, al' \,\rangle \qquad \sigma' \vdash_{assigns} al \Rightarrow \langle\, \sigma''\,,\, al'' \,\rangle}{\sigma \vdash_{assigns} a\ al \Rightarrow \langle\, \sigma''\,,\, al' +\!\!+ al'' \,\rangle}$$

Semantic Values

$$\begin{array}{lll}
v & \in \text{Values[FCL]} & = \{0, 1, 2, \ldots\} \\
w & \in \text{PE-Values[FCL]} & = (\{S\} \times \text{Values[FCL]}) \cup (\{D\} \times \text{Expressions[FCL]}) \\
l & \in \text{Labels[FCL]} & = \text{Block-Labels[FCL]} \cup \{\mathsf{halt}\} \\
\sigma & \in \text{Stores[FCL]} & = \text{Variables[FCL]} \rightharpoonup \text{PE-Values[FCL]} \\
\Gamma & \in \text{Block-Maps[FCL]} & = \text{Block-Labels[FCL]} \rightharpoonup \text{Blocks[FCL]}
\end{array}$$

Fig. 16. Online partial evaluation for FCL programs (part 1)

Jumps

$$\frac{}{\sigma \vdash_{jump} \textbf{goto}\ l; \Rightarrow \langle\, \{l\}\,,\ \textbf{goto}\ (l,\sigma);\rangle}$$

$$\frac{\sigma \vdash_{expr} e \Rightarrow w \qquad w \Uparrow e'}{\sigma \vdash_{jump} \textbf{return}\ e; \Rightarrow \langle\, \{\mathsf{halt}\}\,,\ \textbf{return}\ e';\rangle}$$

$$\frac{\sigma \vdash_{expr} e \Rightarrow \langle S\,,\ v\rangle \qquad \mathsf{is\text{-}true?}(v)}{\sigma \vdash_{jump} \textbf{if}\ e\ \textbf{then}\ l_1\ \textbf{else}\ l_2; \Rightarrow \langle\, \{l_1\}\,,\ \textbf{goto}\ (l_1,\sigma);\rangle}$$

$$\frac{\sigma \vdash_{expr} e \Rightarrow \langle S\,,\ v\rangle \qquad \mathsf{is\text{-}false?}(v)}{\sigma \vdash_{jump} \textbf{if}\ e\ \textbf{then}\ l_1\ \textbf{else}\ l_2; \Rightarrow \langle\, \{l_2\}\,,\ \textbf{goto}\ (l_2,\sigma);\rangle}$$

$$\frac{\sigma \vdash_{expr} e \Rightarrow \langle D\,,\ e'\rangle}{\sigma \vdash_{jump} \textbf{if}\ e\ \textbf{then}\ l_1\ \textbf{else}\ l_2; \Rightarrow \langle\, \{l_1, l_2\}\,,\ \textbf{if}\ e'\ \textbf{then}\ (l_1,\sigma)\ \textbf{else}\ (l_2,\sigma);\rangle}$$

Blocks

$$\frac{\begin{array}{c}\sigma \vdash_{assigns} al \Rightarrow \langle\, \sigma_1\,,\ al_1\,\rangle \\ \sigma_1 \vdash_{jump} j \Rightarrow \langle\, \{l_{2_i} \mid i \in \{1, \ldots n\}\}\,,\ j_2\,\rangle\end{array}}{\sigma \vdash_{block} l:\ al\ j \Rightarrow \langle\, \{(l_{2_i}, \sigma_1) \mid i \in \{1, \ldots n\}\}\,,\ (l,\sigma):\ al_1\ j_2\,\rangle} \quad \text{where } n \in \{1,2\}$$

Transitions

$$\frac{\sigma \vdash_{block} \Gamma(l) \Rightarrow \langle\, \{(l'_i, \sigma'_i) \mid i \in \{1, \ldots n\}\}\,,\ b'\,\rangle}{\vdash_\Gamma\ \langle (l,\sigma) :: \mathcal{P}\,,\ \mathcal{S}\,,\ \mathcal{R}\rangle \longmapsto \langle \mathcal{P}_{new} +\!\!+ \mathcal{P}\,,\ \mathcal{S} \cup (l,\sigma)\,,\ b' :: \mathcal{R}\rangle} \quad \text{if } (l,\sigma) \notin \mathcal{S}$$
$$\text{where } \mathcal{P}_{new} = \mathsf{remove\text{-}halts}([(l'_1, \sigma'_1), \ldots, (l'_n, \sigma'_n)])$$

$$\vdash_\Gamma\ \langle (l,\sigma) :: \mathcal{P}\,,\ \mathcal{S}\,,\ \mathcal{R}\rangle \longmapsto \langle \mathcal{P}\,,\ \mathcal{S}\,,\ \mathcal{R}\rangle \qquad \text{if } (l,\sigma) \in \mathcal{S}$$

Fig. 17. Online partial evaluation for FCL programs (part 2)

The process of converting a semantic value to a piece of code for the purpose of residualization is called *lifting* (symbolized by $\Uparrow$). We use two lifting rules that will return a piece of code for any given pe-value. If the pe-value is static ($\langle S\,,\,v\rangle$), the constant corresponding to v is returned. The appropriate constant is given using the inverse of $[\![\cdot]\!]$ (which we write as $[\![\cdot]\!]^{-1}$). The inverse function exists since $[\![\cdot]\!]$ is required to be injective. If the pe-value is dynamic ($\langle D\,,\,e\rangle$), then the D tag is simply stripped off and the code e is returned.

As examples, the following derivations show the partial evaluation of power program expressions `*(result m)` and `-(n 1)` using the store

$$\sigma \;=\; [\mathtt{m} \mapsto \langle D\,,\,\mathtt{m}\rangle, \mathtt{n} \mapsto \langle S\,,\,2\rangle, \mathtt{result} \mapsto \langle S\,,\,1\rangle].$$

`*(result m)`:

$$\nabla_1 \;=\; \frac{\overline{\sigma \vdash_{expr} \mathtt{result} \Rightarrow \langle S\,,\,1\rangle} \quad \overline{\sigma \vdash_{expr} \mathtt{m} \Rightarrow \langle D\,,\,\mathtt{m}\rangle} \quad \overline{\langle S\,,\,1\rangle \Uparrow \mathtt{1}} \quad \overline{\langle D\,,\,\mathtt{m}\rangle \Uparrow \mathtt{m}}}{\sigma \vdash_{expr} \mathtt{*(result\ m)} \Rightarrow \langle D\,,\,\mathtt{*(1\ m)}\rangle}$$

`-(n 1)`:

$$\nabla_2 \;=\; \frac{\overline{\sigma' \vdash_{expr} \mathtt{n} \Rightarrow \langle S\,,\,2\rangle} \quad \overline{\sigma' \vdash_{expr} \mathtt{1} \Rightarrow \langle S\,,\,1\rangle} \quad [\![\mathtt{-}]\!](2\ 1)}{\sigma' \vdash_{expr} \mathtt{-(n\ 1)} \Rightarrow \langle S\,,\,1\rangle}$$

where $\sigma' = \sigma[\mathtt{result} \mapsto \langle D\,,\,\mathtt{result}\rangle]$.

Assignments The strategy for handling an assignment is simple:

- if the right-hand side of an assignment x `:=` e; is a static value v, then the assignment is computed (x is bound to $\langle S\,,\,v\rangle$ in the updated store), and
- if the right-hand side of the assignment is dynamic (returns $\langle D\,,\,e'\rangle$), then the residual assignment x `:=` e'; is generated, and x is bound to $\langle D\,,\,x\rangle$ in the updated store.

Residual assignments are returned in lists so that they can easily be appended to the result of processing a list of assignments. Thus, in the former case, an empty list is returned. In the latter case, a singleton list is returned. There are two rules for processing assignment lists. The first simply returns the same store and an empty residual assignment list. The second appends the result of processing the first assignment to the result of processing the rest of the assignment list ($+\!\!+$ is the append operation).

Here are some example derivations. For `result := *(result m);` we have

$$\nabla_3 \;=\; \frac{\begin{array}{c}\nabla_1 \\ \sigma \vdash_{expr} \mathtt{*(result\ m)} \Rightarrow \langle D\,,\,\mathtt{*(1\ m)}\rangle\end{array}}{\sigma \vdash_{expr} \mathtt{result := *(result\ m);} \Rightarrow \langle\, \sigma'\,,\,[\mathtt{result := *(1\ m);}]\,\rangle}$$

where $\sigma' = \sigma[\texttt{result} \mapsto \langle D\,,\,\texttt{result}\rangle]$. For $\texttt{n} := \texttt{-(n 1)};$ we have

$$\nabla_4 \;=\; \dfrac{\dfrac{\nabla_2}{\sigma' \vdash_{expr} \texttt{-(n 1)} \Rightarrow \langle S\,,\,1\rangle}}{\sigma' \vdash_{expr} \texttt{n} := \texttt{-(n 1)}; \Rightarrow \langle\, \sigma''\,,\,[]\,\rangle}$$

where $\sigma'' = \sigma'[\texttt{n} \mapsto \langle S\,,\,1\rangle]$. Now, using the following abbreviations

$$\begin{aligned} a_1 &= \texttt{result} := \texttt{*(result m)};\\ a_1' &= \texttt{result} := \texttt{*(1 m)};\\ a_2 &= \texttt{n} := \texttt{-(n 1)}; \end{aligned}$$

we have the following derivation ∇_5.

$$\dfrac{\sigma \vdash_{assign} a_1 \Rightarrow \langle\, \sigma'\,,\,[a_1']\,\rangle \qquad \dfrac{\sigma' \vdash_{assign} a_2 \Rightarrow \langle\, \sigma''\,,\,[]\,\rangle \qquad \dfrac{}{\sigma'' \vdash_{assigns} \cdot \Rightarrow \langle\, \sigma''\,,\,[]\,\rangle}}{\sigma' \vdash_{assigns} a_2 \Rightarrow \langle\, \sigma''\,,\,[]\,\rangle}}{\sigma \vdash_{assigns} a_1\, a_2 \Rightarrow \langle\, \sigma''\,,\,[a_1']\,\rangle}$$

Jumps Figure 17 presents the rest of the definition of online partial evaluation. This figure defines partial evaluation *without* transition compression (excluding transition compression makes the definition easier to understand). Transition compression will be defined in the following section.

The specialization semantics of jumps is given by the following judgement form

$$\sigma \vdash_{jump} j \Rightarrow \langle\, S\,,\,j'\,\rangle$$

where $S \subseteq$ Labels[FCL] $\times$ Stores[FCL] is a set of destination states, and j' is the jump to be placed in the residual program.

Specializing **goto** l; with store σ means that the residual program should **goto** the version of the block labeled l that is specialized with respect to σ (*i.e.*, the residual block labeled (l, σ). Specializing **return** x; simply copies that jump to the residual program. Note that it doesn't make sense to "execute" the **return** x; jump during partial evaluation even if x has a static value. Doing so would terminate the specialization process with a value. If we have passed through a dynamic conditional, there maybe more than one possible return value, and we cannot be sure that the current one is the correct one. More fundamentally, partial evaluation should produce a residual *program* that computes a value — not the value itself.

Proper treatment of conditional jumps is one of the most interesting aspects of partial evaluation. If the test expression of an **if** is static, then the **if** can be computed at specialization time. If the test result is true, then the label l_1 for the true branch is returned, and in the residual program we jump directly *via* **goto** (l_1, σ); to the version of block l_1 that is specialized with respect to σ (similarly, if the test result is false).

If the test expression is dynamic, then the **if** cannot be computed at specialization time — it must placed in the residual program. The residual **if** will cause

a jump to the version of block l_1 and is specialized with respect to σ if the test is true, otherwise there will be a jump to the specialized block (l_2, σ). A set of labels $\{l_1, l_2\}$ is returned corresponding to branch in the computation tree of the residual program.

Below is the derivation for the jump in our example block from the power program.

$$\nabla_6 \;=\; \frac{}{\sigma'' \vdash_{jump} \textbf{goto}\ \texttt{test}; \Rightarrow \langle\, \{\texttt{test}\}\,,\ \textbf{goto}\ (\texttt{test}, \sigma'');\, \rangle}$$

Blocks For a block $l :\ al\ j$, we just

1. call the specializer on the assignment list al (obtaining a new store σ' and residual assignments al'
2. call the specializer on the jump j with store σ' and obtain a set of destination labels and a residual jump j.

The set of residual labels is going to contain one label if the jump is a **goto**, **return**, or a static **if**. It the jump is a dynamic **if**, it will contain two labels.

Using the abbreviations below (along with the ones previously introduced),

$$\begin{aligned} al &= al_1\, al_2 \\ al' &= a'_1 \\ j &= \textbf{goto}\ \texttt{test}; \\ j' &= \textbf{goto}\ (\texttt{test}, \sigma''); \end{aligned}$$

we have the derivation

$$\nabla_7 \;=\; \frac{\nabla_5 \qquad\qquad \nabla_6}{\sigma \vdash_{assigns} al \Rightarrow \langle\, \sigma''\,,\ al'\,\rangle \qquad \sigma'' \vdash_{jump} j \Rightarrow \langle\, \{\texttt{test}\}\,,\ j'\,\rangle}{\sigma \vdash_{block} \texttt{test}:\ al\ j \Rightarrow \langle\, \{(\texttt{test}, \sigma'')\}\,,\ (\texttt{test}, \sigma'') :\ al'\ j'\,\rangle}$$

Transitions The transition rules formalize the **while** loop of Figure 15. Recall that this loop repeatedly transforms a pending set $\mathcal{P}$, a seen-before set $\mathcal{S}$, and a list of residual blocks $\mathcal{R}$. Figure 17 implements $\mathcal{P}$ with a stack used in a LIFO manner (*i.e.*, as a stack). This gives a depth-first generation of the trace that forms the structure of the residual program. In fact, one can safely processing the pending set in any order (*e.g.*, breadth-first), but it is typically done in a depth-first manner.

The first rule for transitions covers the case where the next item on the pending list has not been seen before. In this case, the state (l, σ) is removed from the pending list, and the specializer is called to process the block with label l. Given a list of states L, $\mathsf{remove\text{-}halts}(L)$ filters out states (l_j, σ_j) in L where $l_j = \mathsf{halt}$. Thus, non-halt destination states that result from specializing are added to the pending list. In addition, the processed state (l, σ) is added to the seen before set $\mathcal{S}$, and the residual block b' is added to the residual blocks $\mathcal{R}$.

Blocks

$$\frac{\begin{array}{c}\sigma \vdash_{assigns} al \Rightarrow \langle\, \sigma_1\,,\ al_1\,\rangle \\ \sigma_1 \vdash_{jump} j \Rightarrow \langle\, \{l_{2_i}\ \mid\ i \in \{1,\ \ldots\ n\}\}\,,\ j_2\,\rangle\end{array}}{\sigma \vdash_{block} l:\ al\ j \Rightarrow \langle\, \{(l_{2_i}, \sigma_1)\ \mid\ i \in \{1,\ \ldots\ n\}\}\,,\ (l, \sigma):\ al_1\ j_2\,\rangle} \quad \text{where } j_2 \neq \textbf{goto}\ l';$$

$$\frac{\begin{array}{c}\sigma \vdash_{assigns} al \Rightarrow \langle\, \sigma_1\,,\ al_1\,\rangle \\ \sigma_1 \vdash_{jump} j \Rightarrow \langle\, \{l_2\}\,,\ \textbf{goto}\ (l_2, \sigma_2);\rangle \\ \sigma_2 \vdash_{block} \Gamma(l_2) \Rightarrow \langle\, \{(l_{3_i}, \sigma_{3_i})\ \mid\ i \in \{1,\ \ldots\ n\}\}\,,\ l_2':\ al_2\ j_2\,\rangle\end{array}}{\sigma \vdash_{block} l:\ al\ j \Rightarrow \langle\, \{(l_{3_i}, \sigma_{3_i})\ \mid\ i \in \{1,\ \ldots\ n\}\}\,,\ (l, \sigma):\ al_1\ +\!\!+\ al_2\ j_2\,\rangle}$$

Fig. 18. Online partial evaluation with on-the-fly transition compression

The second rule for transitions covers the case where the next item on the pending list $\mathcal{P}$ has been seen before. In this case, the item is simply removed from $\mathcal{P}$ and processing continues. Notice that there will not be a transition when $\mathcal{P}$ is empty. Thus, the rules of Figure 17 exactly formalize the algorithm of Figure 15.

Definition 4 (online partial evaluation). Let $p = (x_1 \ldots x_n)\ (l)\ b^+$ be a well-formed program. Let the *static parameters* $(x_1^s \ldots x_m^s)$ be a subset (when the list is viewed as a set) of p's parameters. Let the *dynamic parameters* $(x_1^d \ldots x_o^d)$ be the remainder of p's parameters. Let $(x_1^i \ldots x_q^i)$ be p's internal variables (those not occuring as parameters). Let Γ be the blockmap for p.

Define online partial evaluation $[\![\cdot]\!]_{\text{onpe}}$ as follows:

$$[\![p]\!]_{\text{onpe}}(x_1^s, \ldots, x_m^s)(v_1, \ldots, v_m) \stackrel{\text{def}}{=}$$

$$\begin{cases} p_{res} & \text{if } \vdash_\Gamma\ \langle [(l, \sigma_{init})]\,,\ \emptyset\,,\ []\rangle\ \rightarrow^*\ \langle []\,,\ \mathcal{S}\,,\ \mathcal{R}\rangle \\ \text{undefined} & \textit{otherwise.} \end{cases}$$

where

$$\sigma_{init}\ =\ \begin{bmatrix} x_1^s \mapsto \langle S\,,\ v_1\rangle, \ \ldots\ , x_m^s \mapsto \langle S\,,\ v_m\rangle \\ x_1^d \mapsto \langle D\,,\ x_1^d\rangle, \ \ldots\ , x_o^d \mapsto \langle D\,,\ x_o^d\rangle \\ x_1^i \mapsto \langle S\,,\ 0\rangle, \ \ldots\ , x_q^i \mapsto \langle S\,,\ 0\rangle \end{bmatrix}$$

and $v_j \in$ Values[FCL], and

$$p_{res}\ =\ (x_1^d\ \ldots\ x_o^d)\ ((l, \sigma_{init}))\ \mathcal{R}.$$

Including transition compression A simple approach to transition compression is to compress transitions for all static **goto**'s and **if**'s on the fly during specialization [15, Chapter 4]. Incorporating this approach into our specialization semantics is straightforward: we only need to change the treatment of blocks. Figure 18 shows that we simply look to see if the generated residual jump is a

goto (which is what results when processing a **goto** or a static **if**). If it is, we immediately process the destination block and merge the results. This process is continued until a **return** or dynamic **if** is encountered.

For example, in the power program of Figure 1, the only block that does not end in a **goto** or static **if** is the final block `end` (it ends with a **return**. So using the formalization in Figures 16 and 17 with the definition of transition compression in Figure 18 yields the compressed residual power program in Figure 11. Intuitively, we will start with the `init` block and simply accumulate all dynamic assignments until we reach the **return** `result`; jump.

In Section 3.1 we noted that there are various subtleties involving transition compression. Naive transition compression can

1. cause the partial evaluator to go into an infinite loop on certain programs, or
2. lead to an increase in residual code size due to the loss of code sharing.

To illustrate the first problem, consider the program at the top of Figure 19. Although the program loops infinitely for any input value, specialization without transition compression (taking `z` to be dynamic, *i.e.*, no static information is supplied) terminates. The residual program is the second program in Figure 19. Termination is achieved because the state $(\texttt{b2}, [\texttt{x} \mapsto 1, \texttt{y} \mapsto D, \texttt{z} \mapsto D])$ which is reached upon exiting block `b3` will be found in the seen-before set.

In contrast, specialization with transition compression (as defined in Figure 18) will go into an infinite loop since compressing the **goto** at the end of block `b1` means that an entry for block `b2` will never be made in the seen-before set. If fact, the seen-before set is never consulted for this program since **goto**'s are the only form of jumps encountered.

In most of the partial evaluation literature, it is considered acceptable for the specializer to loop on some given static data if the source program always loops on the same static data. Taking this view, the problem of non-termination described above is acceptable, because the transition compression strategy will only lead to looping if the source program always loops on the given static data.

To illustrate the second problem (loss of code sharing) with the approach to transition compression of Figure 18, consider the program at the top of Figure 20. Now consider specializing `x` to 3 (with `z` dynamic) using specialization with the transition compression of Figure 18 (the residual program is given in Figure 20 along with the residual program for specialization without transition compression). On the first pass through the program, the **goto**'s in blocks `b1` and `b2` will be compressed. Thus, no entries will be made for `b2` and `b3` in the seen-before set. This means that when processing the true branch of the **if** in `b3`, that state $(\texttt{b2}, [\texttt{x} \mapsto 4, \texttt{y} \mapsto D, \texttt{z} \mapsto D])$ is inserted into the pending list even though we have already (in some sense) seen it before and generated code for it. However, this code lies in the middle of block of the residual block `(b1,[3,0,D])` and there is no way to jump to it directly. Thus, we must duplicate the residual code for state $(\texttt{b2}, [\texttt{x} \mapsto 4, \texttt{y} \mapsto D, \texttt{z} \mapsto D])$. The residual code `if <>(4 y) ...` for state $(\texttt{b3}, [\texttt{x} \mapsto 4, \texttt{y} \mapsto D, \texttt{z} \mapsto D])$ is duplicated for similar reasons.

Source program

```
(z)
(b1)

  b1: x := 1;                          b3: x := 1;
      y := z;                              goto b2;
      goto b2;

  b2: x := +(x 1);
      y := +(y 1);
      goto b3;
```

Specialization without transition compression

```
(z)
((b1,[0,0,D])

  (b1,[0,0,D]): y := z;                (b3, [2,D,D]): goto (b2, [1,D,D]);
                goto (b2,[1,D,D]);

  (b2, [1,D,D]: y := +(y 1);
                goto (b3, [2,D,D]);
```

Post-processing transition compression as in Similix

```
(z)
((b1,[0,0,D])

  (b1,[0,0,D]): y := z;
                goto (b2,[1,D,D]);

  (b2, [1,D,D]: y := +(y 1);
                goto (b2, [1,D,D]);
```

Fig. 19. Illustrating possible non-termination of transition compression

Source program

```
(x z)
(b1)

  b1: x := +(x 1);                    b3: if <>(x y) then b2 else b4;
      y := z;
      goto b2;

  b2: y := +(y 1);                    b4: return y;
      goto b3;
```

Specialization without transition compression (`x`*=3)*

```
(z)
((b1,[3,0,D])

  (b1,[3,0,D]): y := z;               (b3,[4,D,D]):
                goto (b2,[4,D,D]);        if <>(4 y) then (b2,[4,D,D])
                                                  else (b4,[4,D,D]);

  (b2,[4,D,D]): y := +(y 1);          (b4,[4,D,D]):
                goto (b3, [4,D,D]);       return y;
```

Specialization with on-the-fly transition compression of Figure 18

```
(z)
((b1,[3,0,D])

  (b1,[3,0,D]): y := z;
                y := +(y 1);
                if <>(4 y) then (b2,[4,D,D])
                           else (b4,[4,D,D]);

  (b2,[4,D,D]): y := +(y 1);
                if <>(4 y) then (b2,[4,D,D])
                           else (b4,[4,D,D]);

  (b4,[4,D,D]): return y;
```

Post-processing transition compression as in Similix

```
(z)
((b1,[3,0,D])

  (b1,[3,0,D]): y := z;

  (b2,[4,D,D]): y := +(y 1);
                if <>(4 y) then (b2,[4,D,D])
                           else (b4,[4,D,D]);

  (b4,[4,D,D]): return y;
```

Fig. 20. Illustrating the loss of sharing due to transition compression

Instead of the on-the-fly transition compression given in Figure 18, one may adopt a post-processing strategy similar to the post-unfolding strategy used in Similix [2]. In this approach, one first generates a specialized block for each reachable state (as in the definition in Figures 16 and 17). Now, if a block l is the target of only one jump in the program, it is marked as unfoldable (meaning that a **goto** l; can be compressed). If a block l' is the target of two or blocks, it is marked as non-unfoldable (intuitively, compressing **goto** l'; would result in a loss of sharing). After this marking phase, the program is traversed and all **goto** transitions to unfoldable blocks are compressed. The last program of Figure 20 is the residual program that results from this approach.

This post-processing approach also avoids the problem of looping specialization induced purely by on-the-fly transition compression. All reachable cycles in the program must contain a block that is the target of at least two different blocks. This block would be non-unfoldable — thus, breaking the traversal of the cycle during transition compression. For example, post-processing the residual program (the second program) of Figure 19 yields the third program of Figure 19. Since `(b2,[1,D,D])` is the target of blocks `(b1,[0,0,D])` and `(b3,[2,D,D])`, **goto**'s to it are not compressed. However, the **goto** `(b3,[2,D,D])`; in block `(b2,[1,D,D])` can be compressed since this is the only jump to `(b3,[2,D,D])`. It is important to note that even with this post-processing compression strategy, the specializer will still loop when there are an infinite number of reachable states.

Generalization When specializing, one usually wants as much static computation as possible — leading to the general view that "static is good, dynamic is bad". However, there are times when it is advantageous to have a particular construct be dynamic. Intentionally changing a construct's classification from static to dynamic is a form of *generalization*. Generalization is often used to avoid infinite specialization.

For example, consider specializing the power program to $m \mapsto 5$ instead of $n \mapsto 2$. Trying to accumulate all reachable configurations leads to the following infinite series of states.

$$
\begin{aligned}
&(\texttt{init}, [\texttt{m} \mapsto 5, \texttt{n} \mapsto D, \texttt{result} \mapsto 0]) \\
\to\ &(\texttt{test}, [\texttt{m} \mapsto 5, \texttt{n} \mapsto D, \texttt{result} \mapsto 1]) \\
\to\ &(\texttt{loop}, [\texttt{m} \mapsto 5, \texttt{n} \mapsto D, \texttt{result} \mapsto 1]) \\
\to\ &(\texttt{test}, [\texttt{m} \mapsto 5, \texttt{n} \mapsto D, \texttt{result} \mapsto 5]) \\
\to\ &(\texttt{loop}, [\texttt{m} \mapsto 5, \texttt{n} \mapsto D, \texttt{result} \mapsto 5]) \\
\to\ &(\texttt{test}, [\texttt{m} \mapsto 5, \texttt{n} \mapsto D, \texttt{result} \mapsto 25]) \\
\to\ &(\texttt{loop}, [\texttt{m} \mapsto 5, \texttt{n} \mapsto D, \texttt{result} \mapsto 25]) \\
\to\ &(\texttt{test}, [\texttt{m} \mapsto 5, \texttt{n} \mapsto D, \texttt{result} \mapsto 125]) \\
\to\ &\ldots
\end{aligned}
$$

The infinite trace situation is caused by a combination of two factors.

1. In the power program, the variable `n` is the *induction variable* — it controls the iteration. When `n` is known, the conditional controlling the loop in the power program is static, and the loop will be unfolded at specialization time. When `n` is unknown, we have a dynamic conditional and so we cannot tell how many times to unroll the loop.
2. The variable `result` is acting as an *accumulator*. Typically, the initial values of accumulators are given as constants in the program (so they are static). Here, the values used to update the accumulator (the values of variable `m`) are also known. Since `result`'s value is always known and always increasing, we have an infinite number of reachable configurations.

This is an instance of a general problem that occurs (in any language paradigm) when specializing code that uses an accumulator. Intuitively, we see that (a) knowing the value of `result` in this case is not dramatically increasing the amount of optimization, and (b) having `result` as static is leading to infinite specialization. Generalizing `result` from static to dynamic would solve the non-termination problem. One almost always wants to generalize an accumulating variable if the induction variable is unknown.

Tell handle situations like these, partial evaluators typically allow the user to force generalization by hand annotating the program. In FCL, we can incorporate this by adding an operator **gen** to expressions as follows.

$$e ::= \ldots \mid \mathbf{gen}\, e$$

gen is specialized using the following rule.

$$\frac{\sigma \vdash_{expr} e \Rightarrow w \qquad w \Uparrow e'}{\sigma \vdash_{expr} \mathbf{gen}\, e \Rightarrow \langle D\,,\, e' \rangle}$$

In essence, **gen** hard-codes a lifting operation. Of course, **gen** has meaning only when specializing, and it should be removed when executing the program (or the interpreter should be modified to ignore it).

Now we can changing the power program to use **gen** in the `init` block.

```
init: result := gen 1;
      goto test;
```

In this modified setting, generating reachable configurations when specializing to $m \mapsto 5$ would include the following transitions.

$$\begin{aligned} & (\texttt{init}, [\texttt{m} \mapsto 5, \texttt{n} \mapsto D, \texttt{result} \mapsto 0]) \\ \rightarrow\ & (\texttt{test}, [\texttt{m} \mapsto 5, \texttt{n} \mapsto D, \texttt{result} \mapsto D]) \\ \rightarrow\ & (\texttt{loop}, [\texttt{m} \mapsto 5, \texttt{n} \mapsto D, \texttt{result} \mapsto D]) \\ \rightarrow\ & (\texttt{test}, [\texttt{m} \mapsto 5, \texttt{n} \mapsto D, \texttt{result} \mapsto D]) \end{aligned}$$

The state on the fourth line is identical to the one on the second line (we will find it in the seen-before set), and so the process terminates. Note that this sequence of states *is not* the complete trace of the power program. Since `n` is dynamic, the complete trace would be branching at each `test` node. The branches not shown above would include a state for block `end`.

3.4 Exercises

1. Show the values of the pending list $\mathcal{P}$, seen-before set $\mathcal{S}$, and the residual program for each partial evaluation transition (without on-the-fly transition compression) when specializing the power program to `n` = 2. Your solution should take the same form as the transitions for the term multiplier program at the end of Section 3.2.
2. Using the operational semantics specification of the online partial evaluator *with on-the-fly transition compression*, show the derivation for the specialization of the `done-n` block in the term multiplier program for the state shown at the bottom of the trace in Figure 14.
3. Extend the operational semantics specification of the partial evaluator to process lists and associated operations (car, cdr, cons, *etc.*).
4. **Project:** Following the operational semantics definition, program the FCL online partial evaluator in the language of your choice.
5. **Project:** Revise your implementation of the partial evaluator so that transition compression is performed as a post-processing step (by counting uses of blocks) as discussed at the end of Section 3.3.

3.5 Further reading

Some of the earliest work on online specialization of imperative languages is Ershov's work on "mixed computation" [8, 9] ([8] presents the now classic "power" example in an appendix). Meyer [21] gives techniques for online specialization of imperative programs. Marquard and Steensgaard [20] describe online partial evaluation of an imperative object-oriented language.

For functional languages, Glück and Klimov's work on supercompilation is a very clear and concise presentation of online specialization [10]. Consel and Khoo give a parameterized framework that incorporates both online and offline partial evaluation [4]. Ruf and Weise's work on Fuse [23, 24] is the most extensive work on online partial evaluation for functional languages.

4 Offline Partial Evaluation

This chapter introduces *offline partial evaluation* for FCL. There are several interesting trade-offs between online and offline partial evaluation. We will give a more detailed discussion of these trade-offs later. For now we will simply say that online partial evaluation is potentially more powerful but harder to control (*i.e.*, it is harder to ensure termination and compact residual code). There is anecdotal evidence that the offline strategy is better for handling the complex features of imperative languages such as pointers and side-effects.

4.1 Concepts

Online partial evaluation In FCL online partial evaluation, the task of distinguishing the constructs to be executed from those to be residualized was performed *online*, *i.e.*, as specialization was carried out. Partial evaluation of an expression e returned a tagged pe-value $\langle S\,,\,v\rangle$ or $\langle D\,,\,e'\rangle$. The tags S and D were checked to see if e could be executed, or if e had to be residualized.

For example, recall how the online FCL partial evaluator would have treated the following expression with the store $[\mathtt{a} \mapsto \langle D\,,\,\mathtt{a}\rangle, \mathtt{b} \mapsto \langle S\,,\,3\rangle]$.

$$*(\mathtt{a}\ +(\mathtt{b}\ 2))$$

To handle the `*` operation, the partial evaluator would first process both arguments to see if the `*` operation should be executed or residualized. This would involve processing both arguments of the `+` operation. Processing `b` will return $\langle S\,,\,3\rangle$, and processing `2` will return $\langle S\,,\,2\rangle$. Since both the tags are static, we know the `+` operation can be executed — yielding $\langle S\,,\,5\rangle$. However, since processing `a` yields $\langle D\,,\,\mathtt{a}\rangle$, the `*` operation must be residualized. This requires that the value 5 be lifted to the FCL constant `5`.

Offline partial evaluation In offline partial evaluation, the task of distinguishing constructs to be executed from those to be residualized is performed *offline*, *i.e.*, in a separate phase *before* specialization is carried out. Thus, offline partial evaluation consists of two phases.

1. *Binding-time analysis (BTA):* In this first phase, the user does not supply the actual parameter values to be used in partial evaluation, but only a *specification* of which parameters will be known (static) and which will be unknown (dynamic). Based on this information, the partial evaluator will then classify each construct in the program as *eliminable* (*i.e.*, executable at specialization time) or *residual* (*i.e.*, the construct is to be placed in the residual program).[2] This classification is usually expressed by attaching an annotation to each construct in the source program.
2. *Specialization:* In this phase, the user gives the actual parameter values for the static parameters, and the partial evaluator processes the annotated program to produce the residual program. The interesting point is that the specializer does not need to manipulate tags to check to see if which constructs can or cannot be executed. It simply follows the classification annotations already made by the binding-time analysis.

Using the example expression above, we would supply a specification of $[\mathtt{a} \mapsto D, \mathtt{b} \mapsto S]$ to the binding-time analysis. Since both `b` and `2` represent values that will be known at specialization time, the BTA can determine that the `+` operation can be executed. However, since `a` is represents an unknown value,

[2] The *eliminable/residual* terminology is from Ershov [8].

the BTA can determine that the * operation cannot be executed. Based on this analysis, the BTA produces the annotated code below.

$$\underline{*}(\underline{\mathtt{a}}\ \mathsf{lift}(+(\mathtt{b}\ 2)))$$

Residual constructs are underlined, eliminable constructs are not underlined. A special operator lift has been inserted to indicate that the result of +(b 2) needs to be lifted. Note that the lift operation is inserted in exactly the same place as where the online partial evaluator carried out lifting. Thus, the lifting operation is "compiled" into the annotated program.

Once the annotated program is produced, the partial evaluator takes in the values of the static parameters. In contrast to the online partial evaluator, the offline specializer only needs to hold the values of the static variables in the specialization store. Furthermore, there is no need to tag these static values with S since there are no dynamic values stored. We will see later on that the specializer will never attempt to update or fetch the value of a dynamic variable in a well-annotated program ("well-annotatedness" is a crucial property defined later).

Using the store $[\mathtt{b} \mapsto 3]$, the specializer simply follows the annotations to produce the residual program. The + operation is executed to produce the value 5, the lifting directive is followed to produce the code 5, the * and both residual arguments yield the residual contruct $*(\mathtt{a}\ 5)$.

Assessment Moving from online to offline partial evaluation is a staging transformation: one factors the jobs of propagating tag information (S, D) and specializing into two phases. An analogy with type-checking gives a rough idea of the advantages and disadvantages of this approach. Online partial evaluation is analogous to run-time type-checking where values are tagged with types and type-checks are performed on the fly. Offline partial evaluation is analogous to static type-checking which checks types by computing a conservative approximation of the program's flow of values.

Just like dynamic type-checking, online partial evaluation has the advantage of being less conservative and more flexible. In this strategy, decisions about the execution or residualization of constructs can be based on the actual static values supplied for specialization. In the offline strategy, at binding-time analysis time one does not have the static values in hand — only the tags S and D. This becomes an important issue in stronger forms of program specialization such as supercompilation [10, 25, 27] where it seems difficult to construct an effective binding-time analysis.

Just like static type-checking, offline partial evaluation has the advantage of being more efficient and easier to reason about. Since the type-checking is done once-and-for-all, repeated executions of the program avoid the overhead associated with type-checking. Similarly, since binding-time analysis is done once-and-for-all, repeated specialization of the program (using the same parameter specification, but with different static values) avoids the overhead associated

with inserting and inspecting binding-time tags. Finally, offline partial evaluation seems easier to reason about (easier to debug problematic specialization results) because the binding-time analysis computes a conservative classification of constructs before actually specializing the program. Thus, before specialization actually proceeds, the user can be presented with an annotated program, and the results of specialization can be predicted fairly well from this. For example, in the example annotated expression $\underline{*}(\underline{\mathtt{a}}\ \mathsf{lift}(+(\mathtt{b}\ 2)))$, we can immediately see that the * operation will appear in the residual program, whereas the + operation will not.

The first partial evaluators were online. The offline strategy was originally introduced to overcome problems associated with self-application of partial evaluators. But even as enthusiasm about self-application has waned, the offline strategy has proven to be very effective in handling complex language features.

4.2 Two-level FCL

The standard method of expressing annotated programs produced by binding-time analysis is to use a *two-level language* [22]. Using a two-level language is a general technique for expressing staged-computation [17]. In partial evaluation, the two stages are *specialization time* and residual program *run-time*. To connect with our previous terminology, eliminable constructs will be executed at specialization time; residual constructs will be executed at run-time.

Figure 21 presents a two-level version of FCL called FCL-2. In FCL-2, there are underlined and non-underlined versions of each computational construct To avoid confusion with regular FCL constructs, the FCL-2 syntax metavariables with a hat $\hat{}$.

In offline partial evaluation, we will encounter again the same three steps from online specialization (computing reachable configurations, program point specialization, transition compression). These will occur in the specialization phase that follows the binding-time analysis. In summary, the main components of offline partial evaluation are as follows.

1. binding-time analysis (inputs: source program and input parameter binding times)
   (a) propagating input parameter bindings to the whole program
   (b) annotating the source program (creating a two-level program)
2. specialization (inputs: two-level program and values of the static parameters)
   (a) computing reachable configurations
   (b) program-point specialization
   (c) transition compression

In partial evaluators for realistic languages with side-effects or higher-order functions, binding-time analysis is usually preceded by other analyses such as pointer analysis or closure analysis that compute other data and control flow properties.

Binding-time analysis is usually implemented using a iterative fixpoint algorithm or by constraint-solving methods. We will take a very simple approach and use two steps to construct an annotated program from the source program.

Syntax Domains

$\hat{p} \in$ Programs[FCL-2]	$\hat{x} \in$ Variables[FCL-2]
$\hat{b} \in$ Blocks[FCL-2]	$\hat{e} \in$ Expressions[FCL-2]
$l \in$ Block-Labels[FCL-2]	$\hat{c} \in$ Constants[FCL-2]
$\hat{a} \in$ Assignments[FCL-2]	$\hat{j} \in$ Jumps[FCL-2]
$\hat{al} \in$ Assignment-Lists[FCL-2]	$o \in$ Operations[FCL-2]

Grammar

$$
\begin{array}{rcl}
\hat{p} & ::= & (x^*)\ (l)\ \hat{b}^+ \\
\hat{b} & ::= & l : \ \hat{al}\ \hat{j} \\
\hat{a} & ::= & x := \hat{e}; \ \mid \ \underline{x} := \hat{e}; \\
\hat{al} & ::= & \hat{a}\ \hat{al} \ \mid \ \cdot \\
\hat{e} & ::= & \hat{c} \ \mid \ \hat{x} \ \mid \ o(\hat{e}^*) \ \mid \ \underline{o}(\hat{e}^*) \ \mid \\
 & & \mathsf{lift}(\hat{e}) \\
\hat{c} & ::= & \mathtt{0} \ \mid \ \mathtt{1} \ \mid \ \mathtt{2} \ \mid \ \ldots \ \mid \\
 & & \underline{\mathtt{0}} \ \mid \ \underline{\mathtt{1}} \ \mid \ \underline{\mathtt{2}} \ \mid \ \ldots \\
o & ::= & + \ \mid \ - \ \mid \ * \ \mid \ \ldots \\
\hat{j} & ::= & \textbf{goto}\ l; \ \mid \ \underline{\textbf{goto}}\ l; \ \mid \\
 & & \textbf{return}\ \hat{e}; \ \mid \ \underline{\textbf{return}}\ \hat{e}; \ \mid \\
 & & \textbf{if}\ \hat{e}\ \textbf{then}\ l_1\ \textbf{else}\ l_2; \ \mid \ \underline{\textbf{if}}\ \hat{e}\ \underline{\textbf{then}}\ l_1\ \underline{\textbf{else}}\ l_2;
\end{array}
$$

Fig. 21. Syntax of the Two-Level Flowchart Language FCL-2

- First, we use an simple iterative fixpoint algorithm that will take as input the source program and a binding-time specification for its parameters, and will then classify *all* the variables in the program as either static or dynamic. The result will be what is called a *division* Δ — a splitting of the set of program variables into two categories (static or dynamic).
- Second, using the division as input data, we will make another pass through the source program and produce an annotation for each construct.

In practice (*e.g.*, in constraint-based algorithms), computing the division and computing the annotations are performed simultaneously (the annotations are repeatedly re-adjusted as data flow information becomes more precise). We use the two-step method above for pedagogical reasons, and also to connect with the division idea which was the original way of explaining the concept of binding-time analysis [16] — that is, binding-time analysis determines the *time* (*i.e.*, stage) at which each variable will *bind* to an actual value.

4.3 A motivating example

Binding-time analysis

Computing a congruent division As in the previous chapter, let's specialize the power program of Figure 1 to the exponent value of 2. As we begin with the

binding-time analysis, in contrast to online specialization, we do not have the actual value for the exponent parameter `n`, but instead supply a specification $[\texttt{m} \mapsto D, \texttt{n} \mapsto S]$ that gives the binding-time of each parameter. From this specification we can construct an *initial division*

$$\Delta \;=\; [\texttt{m} \mapsto D, \texttt{n} \mapsto S, \texttt{result} \mapsto S]$$

that matches the parameter specification and has all other program variables classified as static. Non-parameter variables such as `result` can be considered known since all FCL variables are initialized to 0.

To be usuable for specialization, a division must be *congruent*. A division is congruent if for any transition $(l, \sigma) \rightarrow (l', \sigma')$ the values of the static variables at block l' must be computable from the static variables at block l. As Gomard and Jones state, "any variable that depends on a dynamic variable must itself be dynamic". The notion of congruence can be made more precise (see *e.g.*, Jones [14]).

The initial division above is *not* congruent since the static variable `result` of block `loop` is not computable from the static variables at block `test`; specicifically, the value to be assigned to `result` depends on the dynamic variable `m`.

To obtain a congruent division, we repeatedly traverse the program and keep updating the division until we make a traversal where nothing changes. The division must be *conservative* in that sense that if a variable is unknown at any point of the program, then it must be classified as dynamic — even if it is known at some other point. Thus, when updating the division, we are allowed to change a variable's classification from S to D but not vice versa. In essence the division will only change when a dynamic value is assigned to a variable previously classified as static. This strategy results in a congruent division called a *uniform division* since it holds for the entire program. Later, we will see how the constraints above can be relaxed to allow variables to have different classifications at different program points (*point-wise division*) or multiple classifications at the same program point (*polyvariant division*).

For now, we compute a uniform division as follows.

- *First traversal:*
  - `init` block: the division does not change here since `result` (which is static) is assigned the static value 1.
  - `test` block: the division does not change here since there are no assignments.
  - `loop` block: in this case, we cannot compute the $*(\texttt{result}\,\texttt{m})$ operation since `m` is dynamic. Therefore, `result` must be classified as dynamic. That is, we have the new division

    $$\Delta' \;=\; [\texttt{m} \mapsto D, \texttt{n} \mapsto S, \texttt{result} \mapsto D].$$

    The assignment to `n` does not change the division, since $\texttt{n} \mapsto S$ and the operation $-(\texttt{n}\;1)$ is static since both operations are static.

```
(m n)
(init)

init:   result := 1;
        goto test;

test:   if <(n 1) then end else loop;

loop:   result := *(result m);
        n := -(n 1);
        goto test;

end:    return result;
```

Fig. 22. Two-level version of the power program using the division $\Delta' = [\texttt{m} \mapsto D, \texttt{n} \mapsto S, \texttt{result} \mapsto D]$

  - `end` block: there are no assignments here so the division does not change.
- *Second traversal:*
  - `init` block: the division does not change here since `result` (which is now dynamic) is assigned the static value 1 (remember, that we only change the a variable's classification from static to dynamic).
  - `test` block: the division does not change here since there are no assignments.
  - `loop` block: as before, the expression in the assignment to `result` is dynamic. Since `result` is already dynamic, nothing changes. The assignment to `n` proceeds as before, so the division does not change.
  - `end` block: there are no assignments here so the division does not change.

  Since the second traversal does not cause any changes, the final division is Δ'.

We are guaranteed that computing a division in this manner will always terminate since (a) there are a finite number of variables, (b) there are only two possible values (S, D) that can be associated with each variable, and (c) the process is monotonic in the sense that we are only changing from S to D and not vice versa.

Producing a two-level program Given the division Δ' above, we can construct an annotated program in one traversal over the source program. Figure 22 shows the resulting program. We now explain how it was obtained.

- `init` block: since $\Delta'(\texttt{result}) = D$, the assignment to `result` must be residualized (underlined). This means that specializing the expression of the right-hand side of the assignment must also yield a residual piece of code. Therefore, we have the residual `1`. Note that we could have also written lift(1) to

obtain the same effect. For now, we will assume that transition compression is not performed. Therefore, all **goto**'s are underlined since they will all appear in the residual program.

- `test` block: all components of the test expression are static, and so the conditional jump itself can be performed at specialization time. Thus, nothing is underlined in this block.
- `loop` block: in the assignment to `result`, both `result` and `m` are dynamic, thus the `*` operation is residual, and the assignment itself is residual.
  In the assignment to `n`, $\Delta'(\mathtt{n}) = S$ and so both the left- and right-hand sides of the assignment, as well as the assignment itself are not underlined.
- `end` block: the **return** is underlined since, as in online partial evaluation, we always want to residualize **return** jumps.

Specialization We now summarize how the three-step view of specialization changes when moving from online to offline partial evaluation.

Computing reachable configurations At this point, the partial evaluator accepts the actual values for the static parameters and computes the set of reachable configurations. In contrast to the online setting, the congruence property of divisions ensures that we can completely ignore the dynamic variables since no static variable will ever depend on a dynamic one. That is, we need only use the static portion $[\mathtt{n} \mapsto \ldots]$ of a store $[\mathtt{m} \mapsto \ldots, \mathtt{n} \mapsto \ldots \mathtt{result} \mapsto \ldots]$. Therefore, our initial state will be

$$(\mathtt{init}, [\mathtt{n} \mapsto 2]).$$

Here is the trace showing the reachable configurations.

$$\begin{aligned}
&(\mathtt{init}, [\mathtt{n} \mapsto 2]) \\
\rightarrow\ &(\mathtt{test}, [\mathtt{n} \mapsto 2]) \\
\rightarrow\ &(\mathtt{loop}, [\mathtt{n} \mapsto 2]) \\
\rightarrow\ &(\mathtt{test}, [\mathtt{n} \mapsto 1]) \\
\rightarrow\ &(\mathtt{loop}, [\mathtt{n} \mapsto 1]) \\
\rightarrow\ &(\mathtt{test}, [\mathtt{n} \mapsto 0]) \\
\rightarrow\ &(\mathtt{end}, [\mathtt{n} \mapsto 0]) \\
\rightarrow\ &(\mathsf{halt}, [\mathtt{n} \mapsto 0])
\end{aligned}$$

Again, when generating the trace we simply follow the program annotations. Presently, we ignore the the underlined constructs since they contribute nothing to the static store transformations. We only process the non-underlined constructs such as the conditional in the `test` block and the assignment to `n` in `loop`.

```
(m)
((init,[2]))

  (init,[2]): result := 1;
              goto (test,[2]);

  (test,[2]): goto (loop,[2]);

  (loop,[2]): result := *(result,m);
              goto (test,[1]);

  (test,[1]): goto (loop,[1]);

  (loop,[1]): result := *(result,m);
              goto (test,[0]);

  (test,[0]): goto (end,[0]);

  (end,[0]):  return result;
```

Fig. 23. The FCL power program specialized to $\mathtt{n} = 2$ using offline partial evaluation.

Program point specialization As with online partial evaluation, for each reachable state (l, σ_s), we create a version of block l that is specialized to σ_s. The difference is that in the offline case, we don't check S and D tags, we simply follow the annotations. For example, in the `loop` block

```
loop:  result := *(result m);
       n := -(n 1);
       goto test;
```

the components of the first assignment statement are all underlined, so they are copied to the residual program. All the components of the second component are non-underlined, so they are executed and `n` receives a new value in the static store. Finally, the **goto** is copied (minus the underline) to the residual program with the label changed to the appropriate specialized version of `test`. Figure 23 gives the residual program.

Transition compression Transition compression for FCL is orthogonal to the use of an online or offline strategy. We proceed the same as in the online case; the result is given in Figure 24.

Assessment Let's compare the results of offline partial evaluation in Figure 23 to the result of online partial evaluation in Figure 9. In both cases, there are

```
(m)
((init,[2]))

  (init,[2]): result := 1;
              result := *(result,m);
              result := *(result,m);
              return result;
```

Fig. 24. The compressed FCL power program specialized to $\mathtt{m} = 2$.

seven reachable configurations, and thus seven residual blocks. The only real difference between the two (ignoring the irrelevant difference between labels) occurs in the `init` block and the first `loop` block. In the online case, we compute the assignment $\mathtt{result} := 1$; since 1 is static. However, in our division Δ' for the offline case, we have $\mathtt{result} \mapsto D$ because of the assignment that comes later in the `loop` block. Therefore, in contrast to the online case, the assignment to `result` in `init` is *not* executed, but is copied to the residual program.

The difference between the first version of the `loop` block is caused by the difference between the two `init` blocks. In the online case, in the `loop` block we have

$$\mathtt{result} := *(1\ \mathtt{m});$$

but in the offline case we have

$$\mathtt{result} := *(\mathtt{result}\ \mathtt{m});.$$

In the online case, when we reach `loop` for the first time we have $\mathtt{result} \mapsto 1$ because we computed the assignment in `init`. Therefore, the *value* of `result` gets lifted and residualized in the `*` operation. In the offline case, `result` is always dynamic, and so the *variable* `result` is residualized.

Even though these difference are small, they substantiate the claim that offline partial evaluation is more conservative than online. We performed one less assignment in the offline case than we did in the online case. In addition, even though in this example the number of reachable states was the same for both online and offline, online will generally have more. Since online variables do not receive a single S/D classification as in the offline case, more variations are possible.[3]

4.4 Binding-time analysis

Well-formedness of annotated programs The binding-time analysis phase must produce annotations that correctly instruct the specializer how to specialize

[3] Later we will see more general forms of binding-time analyses that allow multiple classifications.

a program. However, the two-level language in Figure 21 is so general that it is possible to produce annotations that incorrectly guide the specializer. For example, the two-level expression

$$*(\underline{\mathtt{a}}\ \underline{+}(\mathtt{b}\ 2))$$

is problematic because the $\underline{+}$ is marked as residual (and therefore will not produce a value) but the nonunderlined $*$ tells the specializer that $*$ can be evaluated. In short, when evaluating $*$, the specializer expects to receive values as arguments but gets a piece of code instead.

The opposite scenario is also problematic. In the expression,

$$\underline{*}(\underline{\mathtt{a}}\ +(\mathtt{b}\ 2))$$

the specializer expects to receive pieces of code as arguments when processing $\underline{*}$, but the result of $+(\mathtt{b}\ 2)$ will be a value since $+$ is non-underlined.

In summary, the binding-time analysis must produce an annotated program that is well-formed in the sense that it does not cause the specializer to "go wrong". Figure 25 gives a rule system that defines this notion of well-formedness.

The rules for expressions have the form

$$\Delta \vdash^{bt}_{expr}\ e\ [\hat{e} : t].$$

This judgment expresses that under division Δ, the two-level expression $\hat{e}$ is a well-annotated version of the source expression e with binding-time type $t \in \{S, D\}$. Having $t = S$ indicates that specializing $\hat{e}$ will produce a value; $t = D$ implies that specializing e will produce a piece of residual code.

We briefly summarize the intuition behind the expression rules.

- A constant c can be annotated as eliminable (with binding-time type S) or residual (with binding-time type D).
- The binding-time type for a variable x is found by in the division (*i.e.*, the type is $\Delta(x)$).
- For an operation to be eliminable, all its arguments must have type S (*i.e.*, they must also be eliminable an produce values at specialization time). When an operation is marked residual, all its arguments also be residual. That is, they must have type D.
- The lift construct is an explicit coercion from from a static to a dynamic binding-time type. In essence, this compiles into the program at binding-time analysis time the decision to lift or not to lift that was made at specialization time in the online partial evaluator of Section 3.3.

For example, using the division $\Delta = [\mathtt{a} \mapsto D, \mathtt{b} \mapsto S]$ we have the following derivations using the well-formedness rules.

$$\nabla_1 = \frac{\dfrac{}{\Delta \vdash^{bt}_{expr}\ \mathtt{b}\ [\mathtt{b} : S]} \qquad \dfrac{}{\Delta \vdash^{bt}_{expr}\ 2\ [2 : S]}}{\Delta \vdash^{bt}_{expr}\ +(\mathtt{b}\ 2)\ [+(\mathtt{b}\ 2) : S]}$$

Expressions

$$\frac{}{\Delta \vdash^{bt}_{expr}\ c\ [c : S]} \qquad \frac{}{\Delta \vdash^{bt}_{expr}\ c\ [\underline{c} : D]}$$

$$\frac{}{\Delta \vdash^{bt}_{expr}\ x\ [x : S]}\ \text{if } \Delta(x) = S \qquad \frac{}{\Delta \vdash^{bt}_{expr}\ x\ [\underline{x} : D]}\ \text{if } \Delta(x) = D$$

$$\frac{\Delta \vdash^{bt}_{expr}\ e_i\ [\hat{e}_i : S]}{\Delta \vdash^{bt}_{expr}\ o(e_1 \ldots e_n)\ [o(\hat{e}_1 \ldots \hat{e}_n) : S]} \qquad \frac{\Delta \vdash^{bt}_{expr}\ e_i\ [\hat{e}_i : D]}{\Delta \vdash^{bt}_{expr}\ o(e_1 \ldots e_n)\ [\underline{o}(\hat{e}_1 \ldots \hat{e}_n) : D]}$$

Lifting

$$\frac{\Delta \vdash^{bt}_{expr}\ e\ [\hat{e} : S]}{\Delta \vdash^{bt}_{expr}\ e\ [\mathsf{lift}(\hat{e}) : D]}$$

Assignments

$$\frac{\Delta \vdash^{bt}_{expr}\ e\ [\hat{e} : S] \qquad \Delta(x) = S}{\Delta \vdash^{bt}_{assign}\ x := e; [x := \hat{e};]} \qquad \frac{\Delta \vdash^{bt}_{expr}\ e\ [\hat{e} : D] \qquad \Delta(x) = D}{\Delta \vdash^{bt}_{assign}\ x := e; [\underline{x} \underline{:=} \hat{e};]}$$

$$\frac{}{\Delta \vdash^{bt}_{assigns}\ \cdot\ [\cdot]} \qquad \frac{\Delta \vdash^{bt}_{assign}\ a\ [\hat{a}] \qquad \Delta \vdash^{bt}_{assigns}\ al\ [\hat{al}]}{\Delta \vdash^{bt}_{assigns}\ a\ al\ [\hat{a}\ \hat{al}]}$$

Jumps

$$\frac{}{\Delta \vdash^{bt}_{jump}\ \textbf{goto}\ l; [\underline{\textbf{goto}}\ l;]} \qquad \frac{\Delta \vdash^{bt}_{expr}\ e\ [\hat{e} : D]}{\Delta \vdash^{bt}_{jump}\ \textbf{return}\ e; [\underline{\textbf{return}}\ \hat{e};]}$$

$$\frac{\Delta \vdash^{bt}_{expr}\ e\ [\hat{e} : S]}{\Delta \vdash^{bt}_{jump}\ \textbf{if}\ e\ \textbf{then}\ l_1\ \textbf{else}\ l_2; [\textbf{if}\ \hat{e}\ \textbf{then}\ l_1\ \textbf{else}\ l_2;]}$$

$$\frac{\Delta \vdash^{bt}_{expr}\ e\ [\hat{e} : D]}{\Delta \vdash^{bt}_{jump}\ \textbf{if}\ e\ \textbf{then}\ l_1\ \textbf{else}\ l_2; [\underline{\textbf{if}}\ \hat{e}\ \underline{\textbf{then}}\ l_1\ \underline{\textbf{else}}\ l_2;]}$$

Blocks

$$\frac{\Delta \vdash^{bt}_{assigns}\ al\ [\hat{al}] \qquad \Delta \vdash^{bt}_{jump}\ j\ [\hat{j}]}{\Delta \vdash^{bt}_{block}\ l :\ al\ j\ [l :\ \hat{al}\ \hat{j}]}$$

Semantic Values

$$t \in \text{BT-Types[FCL]} = \{S, D\}$$
$$\Delta \in \text{Divisions[FCL]} = \text{Variables[FCL]} \rightharpoonup \text{BT-Types[FCL]}$$

Fig. 25. Well-formedness conditions for the two-level language FCL-2

$$\nabla_2 \;=\; \dfrac{\dfrac{}{\Delta \vdash^{bt}_{expr} \mathsf{a}\ [\underline{\mathsf{a}} : D]} \qquad \dfrac{\dfrac{\nabla_1}{\Delta \vdash^{bt}_{expr} +(\mathsf{b}\ 2)\ [+(\mathsf{b}\ 2) : S]}}{\Delta \vdash^{bt}_{expr} +(\mathsf{b}\ 2)\ [\mathsf{lift}(+(\mathsf{b}\ 2)) : D]}}{\Delta \vdash^{bt}_{expr} *(\mathsf{a}\ +(\mathsf{b}\ 2))\ [\underline{*}(\underline{\mathsf{a}}\ \mathsf{lift}(+(\mathsf{b}\ 2))) : D]}$$

Note that the well-formedness rules do not allow the problematic annotation $*(\underline{\mathsf{a}}\ \underline{+}(\mathsf{b}\ 2))$ to be derived, because the non-underlined $*$ requires the rule for eliminable operations, and this rule requires that all arguments have binding-time type S.

The other problematic annotation $\underline{*}(\underline{\mathsf{a}}\ +(\mathsf{b}\ 2))$ is disallowed because $\underline{*}$ requires the rule for residual operations, and this rule requires that all arguments have binding-time type D. However, the rules give $+(\mathsf{b}\ 2)$ the type S. Before $+(\mathsf{b}\ 2)$ can be used as an argument to $\underline{*}$, it must be coerced to type D using lift as in the derivation above.

The rules for assignments encode the strategy used for online partial evaluation. If the assignment expression has type D, then the assignment is residualized. If the expression has type S, then it is eliminated. Note that we require that the mode of the assignment match the variable's type in the division. That is, a residual assignment statement must assign to a dynamic variable, and an eliminable assignment must assign to a static variable.

For jumps: **goto**'s and **return**'s are always annotated as residual. We emphasized in the previous chapter that **return** must always be residualized so this rule is no surprise. However, the decision to always residualize **goto**'s is technical design decision. Including transition compression in specialization might prompt one to mark **goto**'s as eliminable (since they will always be removed in compression). However, in our presentation of transition compression, **goto**'s are removed not in jump specialization but afterwards in block specialization. Thus, we always residualize **goto**'s during jump specialization (but they may be removed later).

On the other hand, not *all* control transfers are residualized: an **if** is eliminable when its test expression is eliminable, otherwise it is residual.

An algorithm for attaching annotations The well-formedness rules yield constraints on annotated programs that a correct binding-time analysis algorithm must satisfy. Specifically, if Δ is a congruent division for program p, and if e is some expression in p, then a correct binding-time analysis must produce an annotated expression $\hat{e}$ such that $\Delta \vdash^{bt}_{expr} e\ [\hat{e} : t]$ for some t (similarly for assignments, jumps, and blocks).

The reader may have noticed that, for a given Δ and e, there may be more than one two-level term $\hat{e}$ such that $\Delta \vdash^{bt}_{expr} e\ [\hat{e} : t]$. For example, using the

division and expression from the previous section, we have

$$(1)\ \Delta \vdash^{bt}_{expr} *(\mathsf{a}\ +(\mathsf{b}\ 2))\ [\underline{*}(\underline{\mathsf{a}}\ \mathsf{lift}(+(\mathsf{b}\ 2))) : D]$$
$$(2)\ \Delta \vdash^{bt}_{expr} *(\mathsf{a}\ +(\mathsf{b}\ 2))\ [\underline{*}(\underline{\mathsf{a}}\ \underline{+}(\mathsf{lift}(\mathsf{b})\ \mathsf{lift}(2))) : D]$$
$$(3)\ \Delta \vdash^{bt}_{expr} *(\mathsf{a}\ +(\mathsf{b}\ 2))\ [\underline{*}(\underline{\mathsf{a}}\ \underline{+}(\mathsf{lift}(\mathsf{b})\ \underline{2})) : D].$$

Clearly, (1) is better (allows more specialization) than (2) or (3) because it has the + operation as eliminable whereas the others do not. The only difference between (2) and (3) is that the 2 is residualized indirectly *via* lift in (2) but directly in (3). In some sense, underlined constants such as $\underline{2}$ are redundant since the same effect can be achieved using non-underlined constants and lift only. One might argue that residual constants should be included since they avoid the computing of lifting. In any case, whether or not to include residual constants is largely a matter of taste. Most presentations in the literature do include them, so we include them here.

The examples above illustrate that the binding-time analysis does have some flexibility in producing correct output. In general, the annotation with as few residual constructs as possible should be produced. In some cases, it may be desirable to deliberately annotate some constructs as residual that could actually be marked as eliminable. This is sometimes done to avoid non-terminating specialization (analogous to the discussion of generalization in Section 3.3.

Another motivation for increasing the number of residual constructs might be to avoid code blow-up due to over-specialization. Just as we added the **gen** construct for the online partial evaluator (Section 3.3), many offline partial evaluators include an analogous construct that lets the user force a particular fragment of code to be residual.

For now, we will simply assume that we want annotations with as few residual constructs as possible. Given a congruent division, it is relatively clear how to do this, *i.e.*, how to map a program p to an annotated version $\hat{p}$. For completeness, Figure 26 gives a rule-based system for attaching annotations given a congruent division Δ. The rules describe a one-pass traversal of the program.

The rules for expressions have the form

$$\Delta \vdash^{ann}_{expr} e \triangleright \langle \hat{e}, t \rangle.$$

Given the division Δ, e should be annotated as $\hat{e}$ and its type is t.

Recall that one of the reasons that we can get multiple correct annotations for a single expression is that lifting can occur in several different places (see annotations (1) and (2) above). The purpose of lifting is to allow static computation in dynamic contexts, but in annotation (2), lifting is used in what is intuitively a static context. The primary property of the annotating algorithm of Figure 26 as opposed to the well-formedness rules of Figure 25 is that the annotating algorithm has a specific judgement for lifting $\cdot \uparrow \cdot$, and this judgement is only used when there is a truly dynamic context. Note that the rules given for lifting never insert a residual constant (*e.g.*, $\underline{1}$). Instead, a lift of the constant

Expressions

$$\frac{}{\Delta \vdash^{ann}_{expr} \; c \; \triangleright \; \langle\, c,\, S \,\rangle}$$

$$\frac{}{\Delta \vdash^{ann}_{expr} \; x \; \triangleright \; \langle\, x\,,\, S \,\rangle} \quad \text{if } \Delta(x) = S \qquad \frac{}{\Delta \vdash^{ann}_{expr} \; x \; \triangleright \; \langle\, \underline{x}\,,\, D \,\rangle} \quad \text{if } \Delta(x) = D$$

$$\frac{\Delta \vdash^{ann}_{expr} \; e_i \; \triangleright \; \langle\, \hat{e}_i\,,\, S \,\rangle}{\Delta \vdash^{ann}_{expr} \; o(e_1 \ldots e_n) \; \triangleright \; \langle\, o(\hat{e}_1 \ldots \hat{e}_n)\,,\, S \,\rangle}$$

$$\frac{\Delta \vdash^{ann}_{expr} \; e_i \; \triangleright \; \langle\, \hat{e}_i\,,\, t_i \,\rangle \qquad \exists j \in \{1, \ldots, n\} \,.\, t_j \;=\; D \qquad \langle\, \hat{e}_i\,,\, t_i \,\rangle \;\uparrow\; \hat{e}'_i}{\Delta \vdash^{ann}_{expr} \; o(e_1 \ldots e_n) \; \triangleright \; \langle\, \underline{o}(\hat{e}'_1 \ldots \hat{e}'_n)\,,\, D \,\rangle}$$

Lifting

$$\frac{}{\langle\, \hat{e}\,,\, S \,\rangle \;\uparrow\; \mathsf{lift}(\hat{e})} \qquad \frac{}{\langle\, \hat{e}\,,\, D \,\rangle \;\uparrow\; \hat{e}}$$

Assignments

$$\frac{\Delta(x) \;=\; S \qquad \Delta \vdash^{ann}_{expr} \; e \; \triangleright \; \langle\, \hat{e}\,,\, S \,\rangle}{\Delta \vdash^{ann}_{assign} \; x := e; \; \triangleright \; x := \hat{e};}$$

$$\frac{\Delta(x) \;=\; D \qquad \Delta \vdash^{ann}_{expr} \; e \; \triangleright \; \langle\, \hat{e}\,,\, D \,\rangle \qquad \langle\, \hat{e}\,,\, t \,\rangle \;\uparrow\; \hat{e}'}{\Delta \vdash^{ann}_{assign} \; x := e; \; \triangleright \; \underline{x} \,\underline{:=}\, \hat{e}';}$$

$$\frac{}{\Delta \vdash^{ann}_{assigns} \; \cdot \; \triangleright \; \cdot} \qquad \frac{\Delta \vdash^{ann}_{assigns} \; a \; \triangleright \; \hat{a} \qquad \Delta \vdash^{ann}_{assigns} \; al \; \triangleright \; \hat{al}}{\Delta \vdash^{ann}_{assigns} \; a \; al \; \triangleright \; \hat{a} \; \hat{al}}$$

Jumps

$$\frac{}{\Delta \vdash^{ann}_{jump} \; \mathbf{goto}\, l; \; \triangleright \; \underline{\mathbf{goto}}\, l;} \qquad \frac{\Delta \vdash^{ann}_{expr} \; e \; \triangleright \; \langle\, \hat{e}\,,\, t \,\rangle \qquad \langle\, \hat{e}\,,\, t \,\rangle \;\uparrow\; \hat{e}'}{\Delta \vdash^{ann}_{jump} \; \mathbf{return}\, e; \; \triangleright \; \underline{\mathbf{return}}\, \hat{e}';}$$

$$\frac{\Delta \vdash^{ann}_{expr} \; e \; \triangleright \; \langle\, \hat{e}\,,\, S \,\rangle}{\Delta \vdash^{ann}_{jump} \; \mathbf{if}\, e \,\mathbf{then}\, l_1 \,\mathbf{else}\, l_2; \; \triangleright \; \mathbf{if}\, \hat{e} \,\mathbf{then}\, l_1 \,\mathbf{else}\, l_2;}$$

$$\frac{\Delta \vdash^{ann}_{expr} \; e \; \triangleright \; \langle\, \hat{e}\,,\, D \,\rangle}{\Delta \vdash^{ann}_{jump} \; \mathbf{if}\, e \,\mathbf{then}\, l_1 \,\mathbf{else}\, l_2; \; \triangleright \; \underline{\mathbf{if}}\, \hat{e} \,\underline{\mathbf{then}}\, l_1 \,\underline{\mathbf{else}}\, l_2;}$$

Blocks

$$\frac{\Delta \vdash^{ann}_{assigns} \; al \; \triangleright \; \hat{al} \qquad \Delta \vdash^{ann}_{jump} \; j \; \triangleright \; \hat{j}}{\Delta \vdash^{ann}_{block} \; l : \; al \; j \; \triangleright \; l : \; \hat{al} \; \hat{j}}$$

Fig. 26. Attaching annotations using the two-level language FCL-2

will be inserted (*e.g.*, $\mathsf{lift}(1)$). To generate lifted constants, we can add the rule

$$\langle\, n\,,\, S\,\rangle \;\uparrow\; \underline{\mathtt{n}}$$

and restrict n from occuring in existing rule for static expressions (which will force the added rule to be used).

Here are some example derivations using the division $\Delta \;=\; [\mathtt{a} \mapsto D, \mathtt{b} \mapsto S]$.

$$\nabla_1 \;=\; \frac{\overline{\Delta \vdash^{ann}_{expr} \;\mathtt{b} \;\triangleright\; \langle\, \mathtt{b}\,,\, S\,\rangle} \qquad \overline{\Delta \vdash^{ann}_{expr} \;2 \;\triangleright\; \langle\, 2\,,\, S\,\rangle}}{\Delta \vdash^{ann}_{expr} \;+(\mathtt{b}\;2) \;\triangleright\; \langle\, +(\mathtt{b}\;2)\,,\, S\,\rangle}$$

$$\frac{\overline{\Delta \vdash^{ann}_{expr} \;\mathtt{a} \;\triangleright\; \langle\, \underline{\mathtt{a}}\,,\, D\,\rangle} \qquad \overset{\nabla_1}{\Delta \vdash^{ann}_{expr} \;+(\mathtt{b}\;2) \;\triangleright\; +(\mathtt{b}\;2)S} \qquad \langle\, +(\mathtt{b}\;2)\,,\, S\,\rangle \;\uparrow\; \mathsf{lift}(+(\mathtt{b}\;2))}{\Delta \vdash^{ann}_{expr} \;*(\mathtt{a}\;+(\mathtt{b}\;2)) \;\triangleright\; \langle\, \underline{*}(\underline{\mathtt{a}}\;\mathsf{lift}(+(\mathtt{b}\;2)))\,,\, D\,\rangle}$$

The annotating algorithm is correct in the sense that any annotation produced will be well-formed. That is, if Δ is a congruent division for p, then for any expression e in p, $\Delta \vdash^{ann}_{expr} \; e \;\triangleright\; \langle\, \hat{e}\,,\, t\,\rangle$ implies $\Delta \vdash^{bt}_{expr} \; e\;[\hat{e} : t]$ (similarly for the other syntactic categories).

4.5 Specialization

Figures 27 and 28 present the definition of offline specialization. Given the motivation for two-level terms in the previous sections, the intuition should be fairly clear. The rules are similar to the rules for online specialization (Figures 16 and 17), but instead of checking binding-time tags, the specializer simply follows the term annotations. Since the specializer does manipulate tagged values, the definition of PE-Values[FCL] is now a simple union of values and expressions instead of a tagged union with tags $\{S, D\}$.

4.6 Other notions of division

Up to this point we have only considered congruent divisions which give a single static/dynamic variable classification for the entire program. The notion of division can be generalized to allow a different classification for each block, or multiple classifications for the same block.

Pointwise divisions Often a variable can be static in one part of the program and dynamic in another. But using a uniform division forces the variable to be classified as dynamic throughout the entire program. A *pointwise* division allows a different classification for each block.

Consider the following program where `x` is static and `y` is dynamic.

Expressions

$$\frac{}{\sigma \vdash^{spec}_{expr} c \Rightarrow [\![c]\!]} \qquad \frac{}{\sigma \vdash^{spec}_{expr} \underline{c} \Rightarrow c}$$

$$\frac{}{\sigma \vdash^{spec}_{expr} x \Rightarrow \sigma(x)} \qquad \frac{}{\sigma \vdash^{spec}_{expr} \underline{x} \Rightarrow x}$$

$$\frac{\sigma \vdash^{spec}_{expr} \hat{e}_i \Rightarrow v_i \qquad [\![o]\!](v_1 \ldots v_n) = v}{\sigma \vdash^{spec}_{expr} o(\hat{e}_1 \ldots \hat{e}_n) \Rightarrow v}$$

$$\frac{\sigma \vdash^{spec}_{expr} \hat{e}_i \Rightarrow e_i}{\sigma \vdash^{spec}_{expr} \underline{o}(\hat{e}_1 \ldots \hat{e}_n) \Rightarrow o(e_1 \ldots e_n)}$$

Lifting

$$\frac{\sigma \vdash^{spec}_{expr} \hat{e} \Rightarrow v}{\sigma \vdash^{spec}_{expr} \mathsf{lift}(\hat{e}) \Rightarrow [\![v]\!]^{-1}}$$

Assignments

$$\frac{\sigma \vdash^{spec}_{expr} \hat{e} \Rightarrow v}{\sigma \vdash^{spec}_{assign} x := \hat{e}; \Rightarrow \langle\, \sigma[x \mapsto v]\,,\, [\,]\,\rangle} \qquad \frac{\sigma \vdash^{spec}_{expr} \hat{e} \Rightarrow e}{\sigma \vdash^{spec}_{assign} \underline{x} \underline{:=} \hat{e}; \Rightarrow \langle\, \sigma\,,\, [x := e;]\,\rangle}$$

$$\frac{}{\sigma \vdash^{spec}_{assigns} \cdot \Rightarrow \langle\, \sigma\,,\, [\,]\,\rangle}$$

$$\frac{\sigma \vdash^{spec}_{assign} \hat{a} \Rightarrow \langle\, \sigma'\,,\, al'\,\rangle \qquad \sigma' \vdash^{spec}_{assigns} \hat{al} \Rightarrow \langle\, \sigma''\,,\, al''\,\rangle}{\sigma \vdash^{spec}_{assigns} \hat{a}\ \hat{al} \Rightarrow \langle\, \sigma''\,,\, al' +\!\!+ al''\,\rangle}$$

Semantic Values

$$\begin{array}{lll}
v & \in \text{Values[FCL]} & = \{0, 1, 2, \ldots\} \\
w & \in \text{PE-Values[FCL]} & = \text{Values[FCL]} \cup \text{Expressions[FCL]} \\
l & \in \text{Labels[FCL]} & = \text{Block-Labels[FCL]} \cup \{\mathsf{halt}\} \\
\sigma & \in \text{Stores[FCL]} & = \text{Variables[FCL]} \rightharpoonup \text{Values[FCL]} \\
\Gamma & \in \text{Block-Maps[FCL]} & = \text{Block-Labels[FCL]} \rightharpoonup \text{Blocks[FCL]}
\end{array}$$

Fig. 27. Specialization for FCL-2 programs (part 1)

Jumps

$$\frac{}{\sigma \vdash^{spec}_{jump} \mathbf{goto}\ l; \Rightarrow \langle\, \{l\}\,,\ \mathbf{goto}\ (l,\sigma); \rangle} \qquad \frac{}{\sigma \vdash^{spec}_{jump} \underline{\mathbf{goto}}\ l; \Rightarrow \langle\, \{l\}\,,\ \mathbf{goto}\ (l,\sigma); \rangle}$$

$$\frac{\sigma \vdash^{spec}_{expr} \hat{e} \Rightarrow e}{\sigma \vdash^{spec}_{jump} \underline{\mathbf{return}}\ \hat{e}; \Rightarrow \langle\, \{\mathsf{halt}\}\,,\ \mathbf{return}\ e; \rangle}$$

$$\frac{\sigma \vdash^{spec}_{expr} \hat{e} \Rightarrow v \qquad \mathit{true\text{-}value}(v)}{\sigma \vdash^{spec}_{jump} \mathbf{if}\ \hat{e}\ \mathbf{then}\ l_1\ \mathbf{else}\ l_2; \Rightarrow \langle\, \{l_1\}\,,\ \mathbf{goto}\ (l_1,\sigma); \rangle}$$

$$\frac{\sigma \vdash^{spec}_{expr} \hat{e} \Rightarrow v \qquad \mathit{false\text{-}value}(v)}{\sigma \vdash^{spec}_{jump} \mathbf{if}\ \hat{e}\ \mathbf{then}\ l_1\ \mathbf{else}\ l_2; \Rightarrow \langle\, \{l_2\}\,,\ \mathbf{goto}\ (l_2,\sigma); \rangle}$$

$$\frac{\sigma \vdash^{spec}_{expr} \hat{e} \Rightarrow e}{\sigma \vdash^{spec}_{jump} \underline{\mathbf{if}}\ \hat{e}\ \underline{\mathbf{then}}\ l_1\ \underline{\mathbf{else}}\ l_2; \Rightarrow \langle\, \{l_1, l_2\}\,,\ \mathbf{if}\ e\ \mathbf{then}\ (l_1,\sigma)\ \mathbf{else}\ (l_2,\sigma); \rangle}$$

Blocks

$$\frac{\begin{array}{c}\sigma \vdash_{assigns} al \Rightarrow \langle\, \sigma_1\,,\ al_1\, \rangle \\ \sigma_1 \vdash_{jump} j \Rightarrow \langle\, \{l_{2_i} \mid i \in \{1, \ldots n\}\}\,,\ j_2\, \rangle\end{array}}{\sigma \vdash_{block} l :\ al\ j \Rightarrow \langle\, \{(l_{2_i}, \sigma_1) \mid i \in \{1, \ldots n\}\}\,,\ (l,\sigma) :\ al_1\ j_2\, \rangle} \quad \text{where } j_2 \neq \mathbf{goto}\ l';$$

$$\frac{\begin{array}{c}\sigma \vdash_{assigns} al \Rightarrow \langle\, \sigma_1\,,\ al_1\, \rangle \\ \sigma_1 \vdash_{jump} j \Rightarrow \langle\, \{l_2\}\,,\ \mathbf{goto}\ (l_2,\sigma_2); \rangle \\ \sigma_2 \vdash_{block} \Gamma(l_2) \Rightarrow \langle\, \{(l_{3_i}, \sigma_{3_i}) \mid i \in \{1, \ldots n\}\}\,,\ l_2' :\ al_2\ j_2\, \rangle\end{array}}{\sigma \vdash_{block} l :\ al\ j \Rightarrow \langle\, \{(l_{3_i}, \sigma_{3_i}) \mid i \in \{1, \ldots n\}\}\,,\ (l,\sigma) :\ al_1 +\!\!+ al_2\ j_2\, \rangle}$$

Transitions

$$\frac{\sigma \vdash_{block} \Gamma(l) \Rightarrow \langle\, \{(l_i', \sigma_i') \mid i \in \{1, \ldots n\}\}\,,\ b'\, \rangle}{\begin{array}{c}\vdash_\Gamma\ \langle (l,\sigma) :: \mathcal{P}\,,\ \mathcal{S}\,,\ \mathcal{R}\rangle \longmapsto \langle \mathcal{P}_{new} +\!\!+ \mathcal{P}\,,\ \mathcal{S} \cup (l,\sigma)\,,\ b' :: \mathcal{R}\rangle \\ \text{where } \mathcal{P}_{new} = \mathsf{remove\text{-}halts}([(l_1', \sigma_1'), \ldots, (l_n', \sigma_n')])\end{array}} \qquad \text{if } (l,\sigma) \notin \mathcal{S}$$

$$\vdash_\Gamma\ \langle (l,\sigma) :: \mathcal{P}\,,\ \mathcal{S}\,,\ \mathcal{R}\rangle \longmapsto \langle \mathcal{P}\,,\ \mathcal{S}\,,\ \mathcal{R}\rangle \qquad \text{if } (l,\sigma) \in \mathcal{S}$$

Fig. 28. Specialization for FCL-2 programs (part 2)

```
(x y)
(b1)

b1: m := +(x 4);
    n := *(y m);
    goto b2;

b2: n := 2;
    m := *(3 y);
    n := +(n x);
    return n;
```

Using a uniform division, we would be forced to make both `m` and `n` dynamic because they both depend on `y` at some point. However, `m` only depends on `x` in block `b1` and can be considered static for that block. The first assignment in `b2` gives `n` a static value After that, `n`'s value only depends on the static variable `x`.

Using a pointwise division, we can have $[\mathtt{m} \mapsto S, \mathtt{n} \mapsto D]$ in block `b1` and $[\mathtt{m} \mapsto D, \mathtt{n} \mapsto S]$ in block `b2`. Formally, a pointwise division Δ^{pw} is a function from program labels to division tuples.

$$\Delta^{pw} \in \text{Point-Divisions[FCL]} = \text{Block-Labels[FCL]} \rightharpoonup \text{Divisions[FCL]}$$

A valid pointwise division for the program above would be

$$\begin{aligned} \mathtt{b1} &\mapsto [\mathtt{x} \mapsto S, \mathtt{y} \mapsto D, \mathtt{m} \mapsto S, \mathtt{n} \mapsto D] \\ \mathtt{b2} &\mapsto [\mathtt{x} \mapsto S, \mathtt{y} \mapsto D, \mathtt{m} \mapsto D, \mathtt{n} \mapsto S] \end{aligned}$$

Polyvariant divisions There are also cases where it is useful to have multiple classifications for each block. Such divisions are called *polyvariant divisions*. Generally, this is desirable when there are multiple incoming paths to a particular block.

For example, consider the following program where `x` is static, `y` is dynamic, and there are multiple paths coming into block `b4`.

```
(x y)
(b1)

b1: if =(x y) then b2 else b3;

b2: m := 2;
    goto b4;

b3: m := y;
    goto b4;

b4: m := +(x m);
    return m;
```

Along the path from `b2`, `m` is static; along the path from `b3`, `m` is dynamic. Rather than forcing `m` to be dynamic throughout all the program (as with a uniform division), or forcing only one division for block `b4` (as with a pointwise division), we may prefer to have two *variants* of a division for block `b4` — one where `m` is static, and another where `m` is dynamic.

Formally, a polyvariant division maps each program label to a set of divisions.

$$\Delta^{pl} \in \text{Poly-Divisions[FCL]} = \text{Block-Labels[FCL]} \rightharpoonup \mathcal{P}(\text{Divisions[FCL]})$$

A valid polyvariant division for the program above would be

$$\begin{array}{l}
\texttt{b1} \mapsto \{[\texttt{x} \mapsto S, \texttt{y} \mapsto D, \texttt{m} \mapsto S]\} \\
\texttt{b2} \mapsto \{[\texttt{x} \mapsto S, \texttt{y} \mapsto D, \texttt{m} \mapsto S]\} \\
\texttt{b3} \mapsto \{[\texttt{x} \mapsto S, \texttt{y} \mapsto D, \texttt{m} \mapsto D]\} \\
\texttt{b4} \mapsto \{[\texttt{x} \mapsto S, \texttt{y} \mapsto D, \texttt{m} \mapsto S], [\texttt{x} \mapsto S, \texttt{y} \mapsto D, \texttt{m} \mapsto D]\}
\end{array}$$

Binding-time analyses that are not polyvariant are called *monovariant* (*e.g.*, those that compute uniform and pointwise divisions). Note that performing a pointwise or polyvariant binding-time analysis is significantly more complicated than performing computing a uniform division since one must take into account the control-flow of the program. For instance, in the program example for the polyvariant division above, we needed to know that blocks `b2` and `b3` have block `b4` as a successor.

4.7 Exercises

1. Outline the steps involved in computing a uniform congruent division for the term multiplier program of Figure 13 where `n` is static and `m` and `term` are dynamic. Your solution should take the same form as the summary of computing the division for the power program in Section 4.3.
2. Using the division computed in the exercise above, give the annotated version of the term multiplier program that would result from using the annotation rules of Figure 26.
3. Continuing from the exercise above and using the rules of Figure 26, show the derivation for the annotation of the `loop` block of the term multiplier program.
4. Given the annotated program for the term multiplier program above, show the values of the pending list $\mathcal{P}$, seen-before set $\mathcal{S}$ and the residual program for each offline partial evaluation transition (without on-the-fly transition compression) when specializing the program to $\texttt{n} = 2$. Contrast your solution to the analogous transitions in the online specialization of the term multiplier program at the end of Section 3.2.
5. Extend the operational semantics specification of the offline partial evaluator (as well as the FCL-2 well-formedness rules of Figure 25 and annotation algorithm of Figure 26) to process lists and associated operations (car, cdr, cons, *etc.*).

6. Extend the operational semantics specification of the offline partial evaluator (as well as the FCL-2 well-formedness rules of Figure 25 and annotation algorithm of Figure 26) to handle a **gen** construct analogous to the one presented in Section 3.3.
7. **Project:** Implement the algorithm for computing a congruent uniform division, the annotation algorithm, and the offline specializer in the language of your choice.
8. **Project:** Modify the partial evaluator so that pointwise or polyvariant divisions are computed in the binding-time analysis (some subtle issues arise here).

4.8 Further reading

For additional perspective, the reader will find it worth returning to Gomard and Jones's original presentation of offline partial evaluation for FCL in [15]. The best documented work on offline partial evaluation for imperative languages is Andersen's work on C-Mix [1] (see also Andersen's chapter in [15]). Glück *et al.*[18] describe F-Spec — an offline partial evaluator for FORTRAN. Consel's group's Tempo system incorporates both compile-time and run-time specialization [3, 5] and many interesting applications have been made with it. There is a huge body of work on offline partial evaluation of functional languages (see [15]). For offline specialization of functional language with imperative features, see [7, 12, 13, 19, 26].

References

1. Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, Computer Science Department, University of Copenhagen, Copenhagen, Denmark, 1994. DIKU Report 94-19.
2. 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.
3. Charles Consel, Luke Hornof, François Noël, Jacques Noyé, and Nicolae Volanschi. A uniform approach for compile-time and run-time specialization. In *Proceedings of the 1996 International Seminar on Partial Evaluation*, number 1110 in Lecture Notes in Computer Science, pages 54–72, Dagstuhl Castle, Germany, February 1996.
4. Charles Consel and Siau Cheng Khoo. Parameterized partial evaluation. *ACM Transactions on Programming Languages and Systems*, 15(3):463–493, 1993.
5. Charles Consel and François Noël. A general approach for run-time specialization and its application to C. In *Proceedings of the Twenty-third Annual ACM Symposium on Principles of Programming Languages*, pages 145–156, St. Petersburg, FLA USA, January 1996. ACM Press.
6. Olivier Danvy, Robert Glück, and Peter Thiemann, editors. *1998 Symposium on Partial Evaluation*, 1998.

7. Dirk Dussart, John Hughes, and Peter Thiemann. Type specialisation for imperative languages. In *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming*, pages 204–216, Amsterdam, The Netherlands, June 1997. ACM Press.
8. Andrei P. Ershov. On the partial computation principle. *Information Processing Letters*, 6(2):41–67, 1977.
9. Andrei P. Ershov. Mixed computation: Potential applications and problems for study. *Theoretical Computer Science*, 18:41–67, 1982.
10. Robert Glück and Andrei Klimov. Occam's razor in metacomputation: the notion of a perfect process tree. In Patrick Cousot, Moreno Falaschi, Gilberto Filè, and Antoine Rauzy, editors, *Proceedings of the Third International Workshop on Static Analysis WSA'93*, volume 724 of *Lecture Notes in Computer Science*, pages 112–123, Padova, Italy, September 1993.
11. Carsten K. Gomard and Neil D. Jones. Compiler generation by partial evaluation. In G. X. Ritter, editor, *Information Processing '89. Proceedings of the IFIP 11th World Computer Congress*, pages 1139–1144. IFIP, North-Holland, 1989.
12. John Hatcliff. Foundations of partial evaluation of functional programs with computational effects. *ACM Computing Surveys*, 1998. (in press).
13. John Hatcliff and Olivier Danvy. A computational formalization for partial evaluation. *Mathematical Structures in Computer Science*, 7:507–541, 1997. Special issue devoted to selected papers from the *Workshop on Logic, Domains, and Programming Languages*. Darmstadt, Germany. May, 1995.
14. Neil D. Jones. The essence of program transformation by partial evaluation and driving. In Masahiko Sato Neil D. Jones, Masami Hagiya, editor, *Logic, Language and Computation, a Festschrift in honor of Satoru Takasu*, pages 206–224. Springer-Verlag, April 1994.
15. Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall International, 1993.
16. Neil D. Jones, Peter Sestoft, and Harald Søndergaard. MIX: A self-applicable partial evaluator for experiments in compiler generation. *LISP and Symbolic Computation*, 2(1):9–50, 1989.
17. Ulrik Jørring and William L. Scherlis. Compilers and staging transformations. In Mark Scott Johnson and Ravi Sethi, editors, *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Programming Languages*, pages 86–96, St. Petersburg, Florida, January 1986.
18. Paul Kleinrubatscher, Albert Kriegshaber, Robert Zöchling, and Robert Glück. Fortran program specialization. *SIGPLAN Notices*, 30(4):61–70, 1995.
19. Julia L. Lawall and Peter Thiemann. Sound specialization in the presence of computational effects. In *Proceedings of Theoretical Aspects of Computer Software*, Lecture Notes in Computer Science, September 1997. (to appear).
20. M. Marquard and B. Steensgaard. Partial evaluation of an object-oriented imperative language. Master's thesis, University of Copenhagen, Copenhagen, Denmark, 1992.
21. U. Meyer. Techniques for partial evaluation of imperative languages. In Paul Hudak and Neil D. Jones, editors, *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, SIGPLAN Notices, Vol. 26, No 9, pages 94–105, New Haven, Connecticut, June 1991. ACM Press.
22. Flemming Nielson and Hanne Riis Nielson. *Two-Level Functional Languages*, volume 34 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1992.

23. Erik Ruf. *Topics in online partial evaluation.* PhD thesis, Stanford, Palo Alto, California, February 1993.
24. Erik Ruf and Daniel Weise. On the specialization of online program specializers. *Journal of Functional Programming*, 3, 1993.
25. Morten Heine Sørensen, Robert Glück, and Neil Jones. A positive supercompiler. *Journal of Functional Programming*, 6(6):811–838, 1996.
26. Peter Thiemann. A generic framework for partial evaluation of programs with computational effects. In *Proceedings of the Seventh European Symposium on Programming*, 1998.
27. Valentin F. Turchin. The concept of a supercompiler. *ACM Transactions on Programming Languages and Systems*, 8(3):292–325, 1986.
28. Glynn Winskel. *The Formal Semantics of Programming Languages: An Introduction.* MIT Press, 1993.