MiXIMUM
A simpler and more liberal type specializer

Henning Makholm
henning@makholm.net

March 5, 2000

Contents

List of Figures

Chapter 1

Introduction

1.1 Optimal program specializers

A **program specializer** is a software system that given a program p and (commonly) some of its input d_1 produces a new program $\mathrm{p}_{\mathrm{res}}$ whose behavior on the remaining input is identical to that if the original program.

We can also express that in algebraic guise:

$$\forall \mathrm{d}_1, \mathrm{d}_2 : \quad [\![\mathrm{p}]\!](\mathrm{d}_1, \mathrm{d}_2) = [\![\, [\![\mathit{spec}]\!](\mathrm{p}, \mathrm{d}_1)]\!](\mathrm{d}_2) \tag{1.1}$$

where $[\![\cdot]\!]$ is the mapping that takes a program text to its meaning as a function from input to output.

The program p that is given to the specializer is called the **subject program**. The resulting program, called $\mathrm{p}_{\mathrm{res}}$ in the verbal description and $[\![\mathit{spec}]\!](\mathrm{p}, \mathrm{d}_1)$ in the equation, is called the **residual program** d_1 is called the **static input** and d_2 the **dynamic input**. The idea is that $\mathrm{p}_{\mathrm{res}}$ is especially useful to have if you have to run p a lot of times with the same d_1 but different d_2.

It is not hard to construct a program specializer with the stated property. Basically, one can just hard-code d_1 into p and otherwise keep p unchanged. The real challenge is to employ the knowledge of d_1 to produce a $\mathrm{p}_{\mathrm{res}}$ that runs (significantly) faster than p.

During the last few decades, specializers have been developed which are remarkably succesful in achieving that goal, mostly so in situations where d_1 can be viewed as a program (in a more or less complex programming language) which is interpreted by p. In that case $\mathrm{p}_{\mathrm{res}}$ is a program that behaves like d_1 but is written in the same programming language as p is. That is, the net effect of the specialization has been to *compile* the program d_1 into another language, with the interpreter p acting as a catalyst. This effect was first noted by Futamura [1971] who stated what is now known as the *First Futamura Projection*[1]:

$$[\![\mathit{spec}]\!](\mathit{int}, \mathit{source}) = \mathit{target} \tag{1.2}$$

Some contemporary specializers realize this so well that running $[\![\mathit{target}]\!](\mathrm{d})$ is an order of magnitude (or more) faster than running $[\![\mathit{int}]\!](\mathit{source}, \mathrm{d})$. In many cases the efficiency of *target* is as good as if it had been manually translated from

[1] There are two other Futamura Projections as well, but this paper is not concerned with them.

source by a human programmer who knows how the source language works (notice that *spec* does not know that—that knowledge is only implicit in *int* which is data for *spec*). This effect has been expressed by the catch phrase that the specializer "removes an entire layer of interpretation".

The ability to remove entire layers of interpretation was formalised by Jones et al. [1987, problem 3.8]: a program specializer *spec* was called "strong enough" iff for some self-interpreter[2] *sint* and for all correct programs p,

$$[\![\mathit{spec}]\!](\mathit{sint}, \mathtt{p}) =_\alpha \mathtt{p} \tag{1.3}$$

(where $=_\alpha$ denotes textual identity modulo trivial changes such as renaming of variables or reordering of function definitions). The intuition is that if *spec* can do a good job with *sint*, which cannot be entirely trivial because the language it interprets is strong enough to permit a self-interpreter to be written, it can probably do good jobs with other interpreters, too.

Later [Jones et al. 1993, Section 6.4] this notion evolved into the modern concept of an **optimal** program specializer, which merely requires that $[\![\mathit{spec}]\!](\mathit{sint}, \mathtt{p})$ is *at least* as efficient as p on all inputs. An optimal specializer is allowed to be "more than strong enough"—for example, it may apply more local optimizations to the residual program than are necessary to obtain the α-equivalence in (1.3).

The notion of optimal specialization is not completely foolproof, because it is possible for the specializer to "cheat" by simply outputting its second input if it recognizes its first input as the self-interpreter used in the test, and otherwise using a general but inefficient specialization method. That possibility nonwithstanding, the notion has proved very productive as an engineering standard.

1.2 John Hughes' type specializer

In the mid-1990's, optimal specializers existed for several untyped (or dynamically typed) programming languages, including various Lisp dialects and the untyped lambda calculus. Optimal specialization for languages with nontrivial type systems had proved to be a hard problem, however.

The problem is that when there is no single type (or finite set of types) that decribes all of the values any program may construct, the self-interpreter needs to represent the values manipulated by the interpreted program in an encoded form. And the specialization techniques in use at that date were not powerful enough to eliminate that encoding, so the residual programs would still contain code to interpret and maintain the encoding used by the self-interpreter.

Then Hughes [1996a,b] presented his *type specializer* which optimally[3] specialized the simply-typed lambda calculus extended with integer arithmetic and sum and product types.

[2] That is an interpreter that interprets the language in which it is itself written

[3] Actually, Hughes did not demonstrate that the type specializer was optimal, because the interpreter he specialized interpreted only a subset of the language it was written in. He did manage, however, to convince many people—this author included—that the type specializer *could* be optimal if the required self-interpreter was constructed and the termination properties of the type specializer itself was better understood.

The type specializer departs radically from the predominant technique for program specialization which could be called **partial interpretation**[4] and consists roughly of tracing possible execution paths through the subject program, keeping track of the variable values that are known or can be precomputed and producing specialized versions of the subject program containing computations of those values that cannot be precomputed. In contrast to this, the type specializer works by performing *type inference* for a non-standard type system. This is one of the factors that give the type specializer its name—the other is that it is able to produce residual programs that use arbitrarily complex types; that (quasi-mythical) capability had already been called "type specialization" for some time.

For all its virtues, the type specializer also has some deficiencies:

- The type specializer sometimes *rejects static inputs* when the subject program uses them in ways the specializer cannot handle.

  Hughes [1996a, Section 8.8] argues that when specializing an interpreter this only happens if the source program (*i.e.*, the static input to the specializer) is ill-typed, thus it is reasonable that the specializer refuses to compile it into the target language. However, specializing interpreters is not the only possible use of a program specializer, and I find it is in general unacceptable for a specializer to reject its input as "ill-formed" (unless the subject program in itself is not a valid program, of course).

- It is *hard to understand* what the type specializer does.

  One difficulty is that the intuitive relation between the non-standard types and the specialization process is complex. The main step of the type specializer is to infer a type derivation consisting of judgements of the form

  $$\Gamma \vdash e : \tau \hookrightarrow e' : \tau'$$

  Here, e is a piece of the subject program. e' is a corresponding piece of an intermediate program which needs an essential post-processing phase to form the residual program. τ and τ' are types from two different non-standard type systems; Hughes calls τ a "two-level type", which is reasonable, and τ' a "residual type", which is not because it is neither (in any obvious way) the type of e' nor of the corresponding piece of the residual program. The value of e' is not quite the same as the value of e (though the value of e' together with τ' allows the value of e to be reconstructed), and neither is it quite the same as that of the corresponding piece of the residual program. And we haven't even begun to look at Γ.

  Another difficulty is understanding how the type specializer manages to be optimal. It works with a fairly complex language, which means that it is hard to see which of the type specializer's features are essential for

[4] This terminology is not standard. Traditionally, there has been no need to distinguish strictly between the extensional specification of a specializer (which is equation (1.1)) and its internal workings, because there has only been one basic technique: the one described by [Jones et al. 1993]. However, given that Hughes as well as Danvy [1996] have proposed specialization techniques that are very different from those used previously, I think it is important to keep the "what"s separate from the "how"s. It would be nice and intuitive to call the traditional technique "partial evaluation", but that term is already firmly established as being completely synonymous with "program specialization". Instead I have coined "partial interpretation" to describe the traditional technique.

optimal specialization of strongly typed languages in general and which are just necessary for supporting non-essential features such as first-class function values.

- The *implementation* of the type specializer is complex. The structure of the type derivation that must be constructed in the main phase does not reflect the data flows it encodes, so at each level, the subderivations need to be constructed in parallel. Consequently, the actual control flow inside Hughes' implementation is not at all easy to follow. It uses a special-purpose monad that collects "demons"—bits of code that constructs parts of the derivations—and only executes them when necessary type information has been produced in other parts of the derivation.

  Furthermore, due to Hughes' "polyvariant sums and products", the derivation rules are nondeterministic—there can be several different possible type derivations for the same combination of subject program and static input. Hughes uses a backtracking heuristic for navigating through this nondeterminism but reports that the backtracking can have nonobvious effects on the specializer's running time and may even lead to non-termination [Hughes 1996a, Section 8.4].

1.3 This paper

The original goal of the project that resulted in this paper was to try to understand Hughes' work better by creating a light-weight version of the type specializer, guided by the question: "which of all these features are actually *necessary* to obtain optimal specialization of a typed language?"

In the beginning, my thesis was that the critical idea in Hughes' work was the idea of using a type-like notation for encoding static values in the specializer, but that it was not important that these types was the result of a type *inference* algorithm.

My plan was to reexpress the specialization phase of the type specializer as something that resembled abstract interpretation more than type inference. I was making reasonably good progress when, a couple of weeks before the deadline, I realized that the computation steps that technical considerations had forced me to move from the specialization phase to a postphase were actually those that solved the hard problems. A quick experiment confirmed that the quite complex specializer I worked on by then could be replaced with a simple partial interpreter with no novel features—and my system could still specialize its typed target language optimally.

Thus, my initial thesis has been disproved. There are excellent compensations, however. The specializer I worked on previously was based on abstract interpretation over a domain that turned out not to have any nice limit properties, so I had to use quite exotic heuristics to find nontrivial fixpoints at all. It was also inherently on-line, and as I had no new innovative proposals to offer for how to control generalization in online specializers, it could only specialize a self-interpreter with respect to the simplest of programs without exhibiting non-termination. Yet it still threatened to become as least as complex as Hughes' one.

Instead of all that, I can now present a specializer, MiXIMUM[5], which is constructed from well-understood standard components. Its only not necessarily terminating component is a good old-fashioned partial interpreter, which means that well-known techinques for controlling the termination of partial interpreters can be applied. In particular, the accumulated experience of how to avoid infinite specialization by annotating the subject program intelligently before specialization, is readily applicable.

Yet MiXIMUM is optimal according to Jones' criterion, even for a typed language. This is possible thanks to an extensive post-processing of the specialized program. The post-processes are direct descendants of Hughes' system. Two of them are built around a type-inference step on the input program—so in the end Hughes was right after all: Type inference *is* the key to optimal specialization of typed languages.

MiXIMUM also never rejects static input as "ill-formed". If p is not a well-typed program when executing $[\![spec]\!](sint, \mathrm{p})$, the specializer leaves just enough tagging and untagging operations in the residual program that it is type-correct and computes at run-time whether the original program would commit a type error. In effect, the specializer can use an interpreter to compile an untyped language *efficiently* into a typed one—without inserting gratuitous run-time checks in programs (or program parts) that do not need them. This is a well-known implementation trick for dynamically typed languages ("soft typing"), but as far as I know, MiXIMUM is the first system that does it automatically when compiling by specializing an interpreter.

Because the insights that enabled me to do this are quite recent, I'll have to ask the reader to bear with the presentation being somewhat rough and unpolished at places.

[5] The choice of this particular name is fairly arbitrary, but it is on purpose that I give my specializer a name at all. After writing this paper and needing to refer to "Hughes' type specializer" many times, I have reached the conclusion that even primitive and unstable prototype systems ought to have names, so that one can refer to them easily in a discussion.

Chapter 2

Language

We use the PEL language defined by Welinder [1998]. It is a minimal strict functional language which manipulates first-order terms built from this grammar:

$$\begin{array}{rcl} \langle \mathit{Val} \rangle \ni v & ::= & (v,v) \\ & | & \mathsf{L}\,v \mid \mathsf{R}\,v \\ & | & 0 \mid 1 \mid 2 \mid \ldots \end{array}$$

The language is attractive for our purposes because its simplicity and because we can use Welinder's PEL self-interpreter, which has been formally proven correct, in optimality experiments. Using an existing self-interpreter vindicates us of the possible critique of having programmed the self-interpreter in unnatural ways to compensate for idiosyncrasies in the specializer.

The syntax of PEL is given in Figure 2.1. The semantics in Figure 2.2 is nonsurprising: a program is a set of mutually recursive functions, the first of which defines the meaning of the program. Pairs and sums are constructed and destructed in the usual way, integer addition and multiplication work as expected, subtraction rounds negative results up to 0, integer comparison encodes false as $\mathsf{L}\,0$ and true as $\mathsf{R}\,0$ (this allows case expressions to be used for if–then–else). For full details see Welinder [1998, Section 3.3].

We assume that there is a distinguished function name f_0 which is the name of the first function (*i.e.*, the main function) of every PEL program.

2.1 Evaluation order

The PEL semantics does not specify the evaluation order of the operands to the arithmetic operators and pair construction in PEL. This works formally because the semantics does not distinguish between different kinds of error conditions. Welinder gets away with that by arguing that tracking error conditions in the semantics would be very complex[1] and "does not appear to be needed".

In the real world, however, I think that it *is* important whether a program such as

$$\mathsf{f}_0\,x = (\mathsf{error} + \mathsf{f}_0\,x)$$

[1] This complexity is exaggerated by the fact that Welinder's main goal is to produce mechine-checked proofs about program transformation, so increased complexity in the semantics would have meant extra complexity in the entire rest of his thesis.

p	$::=$	$f\,x = e;\ p_?$	Program
e	$::=$	c	Integer constant
	$\mid$	$(e \diamond e)$	Integer operation
	$\mid$	(e,e)	Pair construction
	$\mid$	$\mathsf{fst}\ e$	Pair destruction
	$\mid$	$\mathsf{snd}\ e$	Pair destruction
	$\mid$	$\mathsf{L}\ e$	Sum construction
	$\mid$	$\mathsf{R}\ e$	Sum construction
	$\mid$	$\mathsf{case}\ e\ \mathsf{of} \left\{ \begin{smallmatrix} \mathsf{L}\,x \mapsto e \\ \mathsf{R}\,x \mapsto e \end{smallmatrix} \right\}$	Sum destruction
	$\mid$	x	Variable reference
	$\mid$	error	Error indication
	$\mid$	$f\ e$	Function application
	$\mid$	$\mathsf{let}\ x = e\ \mathsf{in}\ e\ \mathsf{end}$	Let-binding
$\diamond$	$::=$	$+ \mid - \mid * \mid =$	Integer operators
f	$\in$	$\langle \mathit{Func} \rangle$	Function names
x	$\in$	$\langle \mathit{Var} \rangle$	Variable names
c	$\in$	$\mathbb{N} = \{0, 1, 2, \ldots\}$	The integers

Figure 2.1: Abstract syntax of PEL. Essentially this is Figure 1 of Welinder [1998].

$$\frac{}{E \vdash_p c \Rightarrow c} \qquad \frac{E \vdash_p e_1 \Rightarrow c_1 \qquad E \vdash_p e_2 \Rightarrow c_2}{E \vdash_p (e_1 \diamond e_2) \Rightarrow v}\ v = c_1 \diamond c_2$$

$$\frac{E \vdash_p e_1 \Rightarrow v_1 \qquad E \vdash_p e_2 \Rightarrow v_2}{E \vdash_p (e_1,e_2) \Rightarrow (v_1,v_2)}$$

$$\frac{E \vdash_p e \Rightarrow (v_1,v_2)}{E \vdash_p \mathsf{fst}\ e \Rightarrow v_1} \qquad \frac{E \vdash_p e \Rightarrow (v_1,v_2)}{E \vdash_p \mathsf{snd}\ e \Rightarrow v_2}$$

$$\frac{E \vdash_p e \Rightarrow v}{E \vdash_p \mathsf{L}\,e \Rightarrow \mathsf{L}\,v} \qquad \frac{E \vdash_p e_0 \Rightarrow \mathsf{L}\,v_1 \qquad E\{x_1 \mapsto v_1\} \vdash_p e_1 \Rightarrow v'}{E \vdash_p \mathsf{case}\ e_0\ \mathsf{of} \left\{ \begin{smallmatrix} \mathsf{L}\,x_1 \mapsto e_1 \\ \mathsf{R}\,x_2 \mapsto e_2 \end{smallmatrix} \right\} \Rightarrow v'}$$

$$\frac{E \vdash_p e \Rightarrow v}{E \vdash_p \mathsf{R}\,e \Rightarrow \mathsf{R}\,v} \qquad \frac{E \vdash_p e_0 \Rightarrow \mathsf{R}\,v_2 \qquad E\{x_2 \mapsto v_2\} \vdash_p e_2 \Rightarrow v'}{E \vdash_p \mathsf{case}\ e_0\ \mathsf{of} \left\{ \begin{smallmatrix} \mathsf{L}\,x_1 \mapsto e_1 \\ \mathsf{R}\,x_2 \mapsto e_2 \end{smallmatrix} \right\} \Rightarrow v'}$$

$$\frac{}{E \vdash_p x \Rightarrow E(x)} \qquad \frac{E \vdash_p e_0 \Rightarrow v \qquad \{x \mapsto v\} \vdash_p e_1 \Rightarrow v'}{E \vdash_p f\,e_0 \Rightarrow v'}\ [f\,x = e_1] \in p$$

$$\frac{E \vdash_p e_0 \Rightarrow v \qquad E\{x \mapsto v\} \vdash_p e_1 \Rightarrow v'}{E \vdash_p \mathsf{let}\ x = e_0\ \mathsf{in}\ e_1\ \mathsf{end} \Rightarrow v'}$$

Figure 2.2: Dynamic semantics for PEL. Except for notation, this is Figure 4 of Welinder [1998]. There is intentionally no rule for error.

terminates immediately with an error message or enters an infinite loop. In particular I think we should care whether a program specializer might transform a program that terminates immidiately with an error message to one that enters an infinite loop.

Therefore, let us *pretend* that an arithmetic operator or a pair construction evaluate its left operand before its right one. We will take care to develop our specializer such that each step preserves this evaluation order, even though the PEL semantics as it stands doesn't allow us to reason formally about whether we succeed.

For use in the specification of transformations we introduce the following abbreviations:

$$\begin{aligned} \langle\!\langle e_0 \rangle\!\rangle\, e_1 &\equiv \mathsf{let}\ x_0 = e_0\ \mathsf{in}\ e_1\ \mathsf{end} \\ e_1\, \langle\!\langle e_0 \rangle\!\rangle &\equiv \mathsf{let}\ x_1 = e_1\ \mathsf{in}\ \mathsf{let}\ x_0 = e_0\ \mathsf{in}\ x_1\ \mathsf{end}\ \mathsf{end} \end{aligned}$$

where x_0 and x_1 are "fresh" variables. In both cases the intended semantics is to compute the value of e_0 but also evaluate e_1 for its possible side effects (of committing an error or entering an infinite loop). The difference between the two forms is the evaluation order of e_0 and e_1.

2.2 Types

Because we are concerned with specialization of typed languages, we add a (monomorphic) type system to PEL. Our types will be quite simple, built from the grammar

$$\begin{aligned} \langle \mathit{Typ} \rangle \ni \tau \quad ::= &\quad \mathsf{int} \\ | &\quad \langle \mathsf{L}\,\tau + \mathsf{R}\,\tau \rangle \\ | &\quad (\tau,\tau) \end{aligned}$$

where we follow the lead of languages such as Haskell and let the syntax of a product type imitate the syntax for values of the type.

We need to be able to speak about recursive types. The traditional way of doing that is to introduce a formal recursion operator μ with associated type variables and identities to allow unfolding of the recursion one step at a time. That is, however, too clumsy a representation to be used in actual algorithms, so instead I prefer to view a recursive type as a graph:

Definition 2.1 *A **type graph** is an directed, labeled, ordered graph where each node has out-arity* 0 *or* 2*; every node with out-arity* 0 *is labeled "*int*" and every node with out-arity* 2 *is labeled "*$\langle \mathsf{L} \cdot + \mathsf{R} \cdot \rangle$*" or "*$(\cdot,\cdot)$*".*

*A **type*** τ *is a particular node in a particular type graph.*

With an appropriate amount of mathematical machinery we can give meaning to expressions such as $\langle \mathsf{L}\,\tau_1 + \mathsf{R}\,\tau_2 \rangle$ for arbitrary types τ_1 and τ_2 and to case analysis on types. The gory details are in Makholm [2000, Section A.1].

Each type $\tau \in \langle \mathit{Typ} \rangle$ naturally describes a set of values $[\![\tau]\!] \subseteq \langle \mathit{Val} \rangle$:

$$\begin{aligned} [\![\mathsf{int}]\!] &= \mathbb{N} \\ [\![\langle \mathsf{L}\,\tau_1 + \mathsf{R}\,\tau_2 \rangle]\!] &= \{\, \mathsf{L}\,v \mid v \in [\![\tau_1]\!] \,\} \cup \{\, \mathsf{R}\,v \mid v \in [\![\tau_2]\!] \,\} \\ [\![(\tau_1,\tau_2)]\!] &= \{\, (v_1, v_2) \mid v_i \in [\![\tau_i]\!] \,\} \end{aligned}$$

$$\frac{}{\Gamma \vdash_T \mathsf{c} : \mathsf{int}} \qquad \frac{\Gamma \vdash_T e_1 : \mathsf{int} \qquad \Gamma \vdash_T e_2 : \mathsf{int}}{\Gamma \vdash_T (e_1 \diamond e_2) : \mathsf{int}} \diamond \neq =$$

$$\frac{\Gamma \vdash_T e_1 : \mathsf{int} \qquad \Gamma \vdash_T e_2 : \mathsf{int}}{\Gamma \vdash_T (e_1 = e_2) : \langle \mathsf{L}\,\mathsf{int} + \mathsf{R}\,\mathsf{int} \rangle} \qquad \frac{\Gamma \vdash_T e_1 : \tau_1 \qquad \Gamma \vdash_T e_2 : \tau_2}{\Gamma \vdash_T (e_1 , e_2) : (\tau_1 , \tau_2)}$$

$$\frac{\Gamma \vdash_T e : (\tau_1 , \tau_2)}{\Gamma \vdash_T \mathsf{fst}\, e : \tau_1} \qquad \frac{\Gamma \vdash_T e : (\tau_1 , \tau_2)}{\Gamma \vdash_T \mathsf{snd}\, e : \tau_2}$$

$$\frac{\Gamma \vdash_T e : \tau_1}{\Gamma \vdash_T \mathsf{L}\, e : \langle \mathsf{L}\,\tau_1 + \mathsf{R}\,\tau_2 \rangle} \qquad \frac{\Gamma \vdash_T e : \tau_2}{\Gamma \vdash_T \mathsf{R}\, e : \langle \mathsf{L}\,\tau_1 + \mathsf{R}\,\tau_2 \rangle}$$

$$\frac{\Gamma \vdash_T e_0 : \langle \mathsf{L}\,\tau_1 + \mathsf{R}\,\tau_2 \rangle \qquad \Gamma\{x_1 \mapsto \tau_1\} \vdash_T e_1 : \tau \qquad \Gamma\{x_2 \mapsto \tau_2\} \vdash_T e_2 : \tau}{\Gamma \vdash_T \mathsf{case}\ e_0\ \mathsf{of} \left\{ \begin{smallmatrix} \mathsf{L}\, x_1 \mapsto e_1 \\ \mathsf{R}\, x_2 \mapsto e_2 \end{smallmatrix} \right\} : \tau}$$

$$\frac{}{\Gamma \vdash_T x : \Gamma(x)} \qquad \frac{}{\Gamma \vdash_T \mathsf{error} : \tau}$$

$$\frac{\Gamma \vdash_T e_0 : \tau_0}{\Gamma \vdash_T f\, e_0 : \tau_1}\, T(f) = \tau_0 \to \tau_1 \qquad \frac{\Gamma \vdash_T e_0 : \tau_0 \qquad \Gamma\{x \mapsto \tau_0\} \vdash_T e_1 : \tau_1}{\Gamma \vdash_T \mathsf{let}\ x = e_0\ \mathsf{in}\ e_1\ \mathsf{end} : \tau_1}$$

Figure 2.3: Typing rules for PEL.

Superficially it might be doubtful whether these equations define anything, because types may be recursive. However, it can be seen by induction on v (which *is* always finite) that the definition uniquely determines the truth value of $v \in [\![\tau]\!]$ for all v and τ.

The typing rules for PEL are given in Figure 2.3. They define a judgement "$\Gamma \vdash_T e : \tau$" where $\Gamma : \langle \mathit{Var} \rangle \to \langle \mathit{Typ} \rangle$ and $T : \langle \mathit{Func} \rangle \to \langle \mathit{Typ} \rangle \times \langle \mathit{Typ} \rangle$ are type environments that give types to the program's variables and functions, respectively.

Definition 2.2 *A PEL program* p *is* ***typable*** *iff there is a* T *such that*

$$\forall [f\,x = e] \in p : \exists \tau_0, \tau_1 : \left\{ \begin{array}{l} T(f) = \tau_0 \to \tau_1 \\ \{x \mapsto \tau_0\} \vdash_T e : \tau_1 \end{array} \right.$$

The typing rules are sound with respect to the dynamic PEL semantics:

Theorem 2.3 *If* p *is typable with witness* T*, then for all* E, Γ, e, v, τ *with*

- $\mathrm{Dom}\, E = \mathrm{Dom}\, \Gamma$
- $\forall x \in \mathrm{Dom}\, E : E(x) \in [\![\Gamma(x)]\!]$
- $E \vdash_p e \Rightarrow v$
- $\Gamma \vdash_T e : \tau$

it holds that $v \in [\![\tau]\!]$.

Proof. By induction on the derivation of $E \vdash_p e \Rightarrow v$. Details omitted. □

Although Welinder treats PEL as a dynamically typed language, his self-interpreter is consciously written such that it can be typed in a type system like ours [Welinder 1998, Section 5.4.8].

Chapter 3

Description of MiXIMUM

In this chapter I describe how MiXIMUM works.

Because the goal of this project is to investigate exactly what is necessary to make a specializer optimal, I have tried to follow the principle that MiXIMUM should be as *weak* as possible. That is, I have ruthlessly omitted otherwise "nice" optimizations whenever it turned out that they were not necessary for specializing the PEL self-interpreter optimally.[1]

A secondary goal has been to make MiXIMUM as modular as possible. The ideal here is that nothing happens in more than one place in the specializer, and that two different things does not happen simultaneously if it can be helped. This goal has resulted in a specializer whose internal structure is very different from a specializer that has been written with efficiency in mind. It has also, hopefully, resulted in a specializer that is easier to understand.

Specializing with MiXIMUM consists of five transformations:

1. *Trivial specialization* of the subject program with respect to static input.

   The output of the trivial specialization is a valid $\mathrm{p}_{\mathrm{res}}$ according to equation 1.1; each of the the subsequent transformations is simply a semantics-preserving source-to-source transformation on well-typable PEL programs (except that the partial interpretation needs some amount of binding-time annotations on it input program to guide it).

   The trivial specialization is only implicitly present in Hughes' articles; his type specializer basically specializes closed terms, and trivial specialization for the lambda calculus consists simply of juxtaposing the subject program and the static input.

2. *Partial interpretation* of the trivially specialized program. The goal of this phase is primarily to decide which functions the final residual program will have, so function calls are unfolded and new specialized functions are generated here.

   The partial interpretation phase also removes[2] branches of case expressions where it statically knows which branch will be taken, and re-

[1] There *are* certain limits to this idealism. For example, the self-interpreter does arithmetic only on behalf of the interpreted program, so it can be specialized even by a specializer that cannot do static arithmetic. I left the static arithmetic in MiXIMUM, however, so that I can still specialize classical examples such as the power function.

[2] This is not a exact representation of what actually happens. See Section 3.2 for the details.

places arithmetic operations with their results when the operands are statically known.

Normally, a partial interpreter constructs specialized expressions whose values do not contain the "static" parts of the subject program's expressions' values. Our partial interpreter does not follow that principle: the specialized version of an expression has exactly the same value as the original expression. This difference is not essential; I simply chose to do it this way because the post-processing I need for other reasons can also simplify the data representations in the way the partial interpreter could otherwise have done.

In other words, the partial interpreter changes the program structure but not the data manipulated by the program.

3. *Sum reduction*. This transformation removes as many sum types as possible from the program. A sum type is removable if only one of the constructors are used to construct values of the sum type. Constructors that inject into a removable sum type are simply erased, and case expressions that analyze removable sumtypes are replaced with let bindings (the unused branch of the case expression disappears completely).

   Constructors that were static during the entire partial interpretation always get removed. That corresponds to what Hughes' type specializer does during its specialization pass, but my sum reduction is stronger because is can also remove constructors that were dynamic during the partial interpretation.

   In the context of specializing the PEL self-interpreter, the sum reduction has the following effects:

   - The tags used in the self-interpreter's value encoding can be removed *if* the source program was well-typed program. If the source program was not well-typed, residues of the value encoding remain in some (but not necessarily all) of the values.
   - The sum types that represent "cons" or "nil" in the environment lists get removed. Each environment collapses into a tuple[3] containing the values of all variables.
   - The sum types in the interpreter's representation of the source program are removed. Afterwards, the source program is just a big cluster of nested pairs that contains all of the integers that represent variable and function names, source program constants, and miscellaneous placeholdes.

4. *Product reduction*. This transformation simplifies product types by removing unused data from the program (for an appropriate definition of "unused"). Pairs where one of the components is unused can be eliminated; other unused data can be replaced with dummy values[4]

   This corresponds directly to the postphase that Hughes calls *void erasure*. One small difference is that our product reduction is not guided by types constructed during the specialization; it infers its own types when deciding which values in the program is unused. I do not think this difference is important: my (unsystematic) experiments show that even very

[3] n-"tuples" for $n > 2$ are made up of a nested structure of PEL's binary pairs.

[4] Because PEL lacks an explicit unit type we use integers—normally 0—as dummy values.

conservative estimates of "unused" types in the product reduction phase still leads to optimal specialization.

In the context of specializing the PEL self-interpreter, the only important effect of product reduction is that the function parameters that originally contained the source program (or parts of it) are identified as being unused and become removed.

After the product reduction, the *extensional* behavior of each of the specialized program's functions is identical to that of one of the functions in the source program. That is, encodings and other administrative values from the self-interpreter have been completely removed from each function's input and output.

5. *Let reduction*. This transformation removes "trivial" let bindings from the program. The preceding transformations use let bindings very liberally, which makes the transformations easy to specify and easy to reason about. Now it is time to unfold most of the let bindings, being careful not to duplicate code or change the evaluation order or termination properties of the program.

   Hughes' type specializer has no direct counterpart of this—primarily because it works with a non-strict language where most of the let bindings we eliminate here never need to be at all.

   The let reduction also simplifies expressions of the form

$$\mathsf{let}\ x = (e_1, e_2)\ \mathsf{in}\ e\ \mathsf{end}$$

   where x only appears in e as operand to fst or snd. In the context of specializing the PEL self-interpreter, the effect is that the tuples that contains the values from the self-interpreter's environment eventually detoriate into individual variables.

   Hughes does something similar as part of his void erasure, but states [Hughes 1996a, Section 5] that he "view[s] this as a hack".

3.1 Trivial specialization

The purpose of the trivial specialization phase is to allow the subsequent transformations to be expressed simply as transformations of complete PEL programs. This means that we don't have to be specific about how the specialization phase treats the static input, because to it the static input is simply a constant in the program text.

In the simplest form, trivial specialization just creates a new main function which constructs the static input, pairs it with the dynamic input, and calls the old main function.

In MiXIMUM, the trivial specialization is done by hand.

3.2 Partial interpretation

The partial interpreter in MiXIMUM uses standard techniques. Its most advanced feature is that it supports *partially static structures* in the manner of Mogensen [1987]. The partial interpreter maintains static knowledge of a *prefix* of the value of each expression. The form of a "static prefix" $\psi \in \langle \mathit{StatPfx} \rangle$ is

$$\begin{array}{lcll}
\langle \mathit{StatPfx}\rangle \ni \psi & ::= & \top & [\![\top]\!] = \langle \mathit{Val}\rangle \\
 & | & 0 \mid 1 \mid 2 \mid \ldots & [\![c]\!] = \{c\} \\
 & | & \mathsf{L}\,\psi & [\![\mathsf{L}\,\psi]\!] = \{\mathsf{L}\,v \mid v \in [\![\psi]\!]\} \\
 & | & \mathsf{R}\,\psi & [\![\mathsf{R}\,\psi]\!] = \{\mathsf{R}\,v \mid v \in [\![\psi]\!]\} \\
 & | & (\psi,\psi) & [\![(\psi_1,\psi_2)]\!] = \{(v_1,v_2) \mid v_i \in [\![\psi_i]\!]\} \\
 & | & \bot & [\![\bot]\!] = \varnothing
\end{array}$$

Figure 3.1: The syntax and meaning of static prefixes. Static prefixes are used in the partial interpretation phase.

$$\frac{}{\mathcal{E} \vdash_{p,\mathcal{R}} c : c \hookrightarrow c} \qquad \frac{\mathcal{E} \vdash_{p,\mathcal{R}} e_i : c_i \hookrightarrow e_i'}{\mathcal{E} \vdash_{p,\mathcal{R}} (e_1 \diamond e_2) : c \hookrightarrow \langle\!\langle e_1' \rangle\!\rangle \, \langle\!\langle e_2' \rangle\!\rangle \, c} \diamond \neq \texttt{=},\ c_1 \diamond c_2 = c$$

$$\frac{\mathcal{E} \vdash_{p,\mathcal{R}} e_i : c_i \hookrightarrow e_i'}{\mathcal{E} \vdash_{p,\mathcal{R}} (e_1 \texttt{=} e_2) : \mathsf{R}\,0 \hookrightarrow \langle\!\langle e_1' \rangle\!\rangle \, \langle\!\langle e_2' \rangle\!\rangle \, \mathsf{R}\,0} \; c_1 = c_2$$

$$\frac{\mathcal{E} \vdash_{p,\mathcal{R}} e_i : c_i \hookrightarrow e_i'}{\mathcal{E} \vdash_{p,\mathcal{R}} (e_1 \texttt{=} e_2) : \mathsf{L}\,0 \hookrightarrow \langle\!\langle e_1' \rangle\!\rangle \, \langle\!\langle e_2' \rangle\!\rangle \, \mathsf{L}\,0} \; c_1 \neq c_2$$

$$\frac{\mathcal{E} \vdash_{p,\mathcal{R}} e_i : \psi_i \hookrightarrow e_i'}{\mathcal{E} \vdash_{p,\mathcal{R}} (e_1 \diamond e_2) : \top \hookrightarrow (e_1' \diamond e_2')} \; \psi_1 = \top \vee \psi_2 = \top$$

Figure 3.2: The specification of the partial interpreter, part I. Each premise containing "e_i" is a shorthand for two similar premises with $i = 1, 2$.

given in Figure 3.1, which also defines a function $[\![\cdot]\!] : \langle \mathit{StatPfx}\rangle \to \mathcal{P}(\langle \mathit{Val}\rangle)$ that takes a static prefix to the set of values it "describes".

The core of the partial interpreter is specified in Figures 3.2 and 3.3. These figures define a judgement form

$$\mathcal{E} \vdash_{p,\mathcal{R}} e_1 : \psi \hookrightarrow e_2$$

where

p is the subject program

$\mathcal{R}$ is a *catalogue of specialized functions*, that is, a finite, injective map $\langle \mathit{Func}\rangle \times \langle \mathit{StatPfx}\rangle \to \langle \mathit{Func}\rangle$.

$\mathcal{E}$ is a *specialization environent*, that is, a map $\langle \mathit{Var}\rangle \to \langle \mathit{StatPfx}\rangle$.

e_1 is an expression from the subject program. The subject program is lightly annotated in that it can contain the non-standard expression forms "f @ e" and "$\mathsf{LIFT}\ e$" which denote dynamic function application and explicit generalization, respectively.

ψ is the statically known prefix of e_1's value.

e_2 is a specialized expression which has *the same* value as e_1 when evaluated in an environment that meets the assertions in $\mathcal{E}$.

$$\frac{\mathcal{E} \vdash_{p,\mathcal{R}} e_i : \psi_i \hookrightarrow e'_i}{\mathcal{E} \vdash_{p,\mathcal{R}} (e_1,e_2) : (\psi_1,\psi_2) \hookrightarrow (e'_1,e'_2)}$$

$$\frac{\mathcal{E} \vdash_{p,\mathcal{R}} e : \top \hookrightarrow e'}{\mathcal{E} \vdash_{p,\mathcal{R}} \mathsf{fst}\, e : \top \hookrightarrow \mathsf{fst}\, e'} \qquad \frac{\mathcal{E} \vdash_{p,\mathcal{R}} e : \top \hookrightarrow e'}{\mathcal{E} \vdash_{p,\mathcal{R}} \mathsf{snd}\, e : \top \hookrightarrow \mathsf{snd}\, e'}$$

$$\frac{\mathcal{E} \vdash_{p,\mathcal{R}} e : (\psi_1,\psi_2) \hookrightarrow e'}{\mathcal{E} \vdash_{p,\mathcal{R}} \mathsf{fst}\, e : \psi_1 \hookrightarrow \mathsf{fst}\, e'}\ \psi_1 \notin \mathbb{N} \qquad \frac{\mathcal{E} \vdash_{p,\mathcal{R}} e : (\psi_1,\psi_2) \hookrightarrow e'}{\mathcal{E} \vdash_{p,\mathcal{R}} \mathsf{snd}\, e : \psi_2 \hookrightarrow \mathsf{snd}\, e'}\ \psi_2 \notin \mathbb{N}$$

$$\frac{\mathcal{E} \vdash_{p,\mathcal{R}} e : (c,\psi_2) \hookrightarrow e'}{\mathcal{E} \vdash_{p,\mathcal{R}} \mathsf{fst}\, e : c \hookrightarrow \langle\!\langle e' \rangle\!\rangle\, c} \qquad \frac{\mathcal{E} \vdash_{p,\mathcal{R}} e : (\psi_1,c) \hookrightarrow e'}{\mathcal{E} \vdash_{p,\mathcal{R}} \mathsf{snd}\, e : c \hookrightarrow \langle\!\langle e' \rangle\!\rangle\, c}$$

$$\frac{\mathcal{E} \vdash_{p,\mathcal{R}} e : \perp \hookrightarrow e'}{\mathcal{E} \vdash_{p,\mathcal{R}} \mathsf{fst}\, e : \perp \hookrightarrow \mathsf{fst}\, e'} \qquad \frac{\mathcal{E} \vdash_{p,\mathcal{R}} e : \perp \hookrightarrow e'}{\mathcal{E} \vdash_{p,\mathcal{R}} \mathsf{snd}\, e : \perp \hookrightarrow \mathsf{snd}\, e'}$$

$$\frac{\mathcal{E} \vdash_{p,\mathcal{R}} e : \psi \hookrightarrow e'}{\mathcal{E} \vdash_{p,\mathcal{R}} \mathsf{L}\, e : \mathsf{L}\,\psi \hookrightarrow \mathsf{L}\, e'} \qquad \frac{\mathcal{E} \vdash_{p,\mathcal{R}} e : \psi \hookrightarrow e'}{\mathcal{E} \vdash_{p,\mathcal{R}} \mathsf{R}\, e : \mathsf{R}\,\psi \hookrightarrow \mathsf{R}\, e'}$$

$$\frac{\mathcal{E} \vdash_{p,\mathcal{R}} e_0 : \top \hookrightarrow e'_0 \qquad \mathcal{E}\{x_i \mapsto \top\} \vdash_{p,\mathcal{R}} e_i : \psi_i \hookrightarrow e'_i}{\mathcal{E} \vdash_{p,\mathcal{R}} \mathsf{case}\ e_0\ \mathsf{of} \left\{\begin{smallmatrix} \mathsf{L}\,x_1 \mapsto e_1 \\ \mathsf{R}\,x_2 \mapsto e_2 \end{smallmatrix}\right\} : \top \hookrightarrow \mathsf{case}\ e'_0\ \mathsf{of} \left\{\begin{smallmatrix} \mathsf{L}\,x_1 \mapsto e'_1 \\ \mathsf{R}\,x_2 \mapsto e'_2 \end{smallmatrix}\right\}}$$

$$\frac{\mathcal{E} \vdash_{p,\mathcal{R}} e_0 : \mathsf{L}\,\psi_1 \hookrightarrow e'_0 \qquad \mathcal{E}\{x_1 \mapsto \psi_1\} \vdash_{p,\mathcal{R}} e_1 : \psi \hookrightarrow e'_1}{\mathcal{E} \vdash_{p,\mathcal{R}} \mathsf{case}\ e_0\ \mathsf{of} \left\{\begin{smallmatrix} \mathsf{L}\,x_1 \mapsto e_1 \\ \mathsf{R}\,x_2 \mapsto e_2 \end{smallmatrix}\right\} : \psi \hookrightarrow \mathsf{case}\ e'_0\ \mathsf{of} \left\{\begin{smallmatrix} \mathsf{L}\,x_1 \mapsto e'_1 \\ \mathsf{R}\,x_2 \mapsto \mathsf{error} \end{smallmatrix}\right\}}$$

$$\frac{\mathcal{E} \vdash_{p,\mathcal{R}} e_0 : \mathsf{R}\,\psi_2 \hookrightarrow e'_0 \qquad \mathcal{E}\{x_2 \mapsto \psi_2\} \vdash_{p,\mathcal{R}} e_2 : \psi \hookrightarrow e'_2}{\mathcal{E} \vdash_{p,\mathcal{R}} \mathsf{case}\ e_0\ \mathsf{of} \left\{\begin{smallmatrix} \mathsf{L}\,x_1 \mapsto e_1 \\ \mathsf{R}\,x_2 \mapsto e_2 \end{smallmatrix}\right\} : \psi \hookrightarrow \mathsf{case}\ e'_0\ \mathsf{of} \left\{\begin{smallmatrix} \mathsf{L}\,x_1 \mapsto \mathsf{error} \\ \mathsf{R}\,x_2 \mapsto e'_2 \end{smallmatrix}\right\}}$$

$$\frac{\mathcal{E} \vdash_{p,\mathcal{R}} e_0 : \perp \hookrightarrow e'_0}{\mathcal{E} \vdash_{p,\mathcal{R}} \mathsf{case}\ e_0\ \mathsf{of} \left\{\begin{smallmatrix} \mathsf{L}\,x_1 \mapsto e_1 \\ \mathsf{R}\,x_2 \mapsto e_2 \end{smallmatrix}\right\} : \perp \hookrightarrow \langle\!\langle e'_0 \rangle\!\rangle\, \mathsf{error}}$$

$$\frac{}{\mathcal{E} \vdash_{p,\mathcal{R}} x : \psi \hookrightarrow x}\ \mathcal{E}(x) = \psi \notin \mathbb{N} \qquad \frac{}{\mathcal{E} \vdash_{p,\mathcal{R}} x : c \hookrightarrow c}\ \mathcal{E}(x) = c$$

$$\frac{}{\mathcal{E} \vdash_{p,\mathcal{R}} \mathsf{error} : \perp \hookrightarrow \mathsf{error}}$$

$$\frac{\mathcal{E} \vdash_{p,\mathcal{R}} e_0 : \psi_0 \hookrightarrow e'_0}{\mathcal{E} \vdash_{p,\mathcal{R}} f\ @\ e_0 : \top \hookrightarrow f'\, e'_0}\ \mathcal{R}(f,\psi_0) = f'$$

$$\frac{\mathcal{E} \vdash_{p,\mathcal{R}} e_0 : \psi_0 \hookrightarrow e'_0 \qquad \{x \mapsto \psi_0\} \vdash_{p,\mathcal{R}} e_1 : \psi_1 \hookrightarrow e'_1}{\mathcal{E} \vdash_{p,\mathcal{R}} f\, e_0 : \psi_1 \hookrightarrow \mathsf{let}\ x = e'_0\ \mathsf{in}\ e'_1\ \mathsf{end}}\ [f\, x = e_1] \in p$$

$$\frac{\mathcal{E} \vdash_{p,\mathcal{R}} e_0 : \psi_0 \hookrightarrow e'_0 \qquad \mathcal{E}\{x \mapsto \psi_0\} \vdash_{p,\mathcal{R}} e_1 : \psi_1 \hookrightarrow e'_1}{\mathcal{E} \vdash_{p,\mathcal{R}} \mathsf{let}\ x = e_0\ \mathsf{in}\ e_1\ \mathsf{end} : \psi_1 \hookrightarrow \mathsf{let}\ x = e'_0\ \mathsf{in}\ e'_1\ \mathsf{end}}$$

$$\frac{\mathcal{E} \vdash_{p,\mathcal{R}} e : \psi \hookrightarrow e'}{\mathcal{E} \vdash_{p,\mathcal{R}} \mathsf{LIFT}\, e : \top \hookrightarrow e'}$$

Figure 3.3: The specification of the partial interpreter, part II.

The partial interpreter begins by setting $\mathcal{R} = \{(f_0, \top) \mapsto f_0\}$ and iteratively adding new pairs to $\mathcal{R}$ until

$$\forall(f, \psi) \in \mathrm{Dom}\,\mathcal{R} : \exists x, e, \psi', e' : \left\{ \begin{array}{l} [f\,x = e] \in p \\ \{x \mapsto \psi\} \vdash_{p,\mathcal{R}} e : \psi' \hookrightarrow e' \end{array} \right. \tag{3.1}$$

Along the way, the e's in this equation are collected to form a specialized program.

3.2.1 Discussion

Most of the rules in Figures 3.2 and 3.3 are nonsurprising. Notice that the "$\langle\!\langle e \rangle\!\rangle\, e$" construction is used liberally for preserving the side-effects of static expressions. The let bindings thus introduced will eventually be removed by the let reduction in the common case that there are no side effects to preserve.

On Figure 3.3, notice the special rule for specializing pair destructions and variable references when the result is statically known to be a particular integer constant.

These rule makes it possible for integer constants to propagate from the specialized function where they are created (or computed) to the function where they are used. The net effect is that "integers can be lifted". Without such rules, a specialized interpreter would have to pass around a tuple of all constants in the source program (which arises as the residue of the source program itself) and extract one of those consants whenever one was needed.

It may seem somewhat *ad hoc* to do the integer lifting precisely at variable references and pair destructions and nowhere else. This is, however, complete in the sense that whenever

$$\mathcal{E} \vdash_{p,\mathcal{R}} e : c \hookrightarrow e'$$

can be derived, e' will either be an integer constant or an integer constant wrapped in a series of let and/or case bindings (this is easy to see by induction on the structure of the derivation). Thus every practical opportunity for lifting is in fact employed.

In fact, to achieve optimal specialization of the PEL self-interpreter, it is only necessary to lift integers at variable references. I left the lifting rules for fst and snd in because even minor modifications to the self-interpreter's data format may make them necessary.

One seemingly attractive idea would be to generalize the lifting rules to handle more complex types than integers. There is a problem here, however: Consider specializing

$$\begin{aligned} f_0\,x_0 &= \mathsf{let}\ x_1 = \mathsf{L\,R\,L\,0}\ \mathsf{in}\ (f_1 \mathbin{@} x_1, f_1 \mathbin{@} x_1)\ \mathsf{end} \\ f_1\,x_0 &= \mathsf{LIFT}\,x_0 \end{aligned}$$

If complex types, even fully static, could be lifted, the result might be

$$\begin{aligned} f_0\,x_0 &= (f_1\,0, f_1\,0) \\ f_1\,x_0 &= \mathsf{L\,R\,L\,0} \end{aligned}$$

which is *less* efficient than the original program under several natural cost measures. This behviour does not seem desirable for a program specializer.

One final feature to notice in Figure 3.3 is that the result of a dynamic case or a dynamic function call is always completely generalized. That is a standard choice for partial interpreters, but it is perhaps surprising that such a primitive technique works when the goal is optimal specialization of a typed language.

The answer is this: In traditional partial evaluators there are two ways to define the difference between static and dynamic data. One is that the dynamic data are those that remain in the residual progam, and the static data are those that disappear during specialization. Another one is that the static data are those that are used to select among multiple specialized versions of each subject function. It is a deep-rooted assumption that these two definitions have to be equivalent. Even Hughes' type specializer, which is otherwise very different from traditional partial interpeters, use the assumption that if we want the type tags in the interpreter to disappear, we need to take them into account when creating specialized functions.

The fundamental new idea in MiXIMUM is that this is not the case. A value *can* disappear in the residual program even if it is not one of the values that are important when identifying which specialized functions the program needs. This means that the only consideration that matters when we select the binding times used in the partial-interpretation phase is how we want the residual functions to be identified.

In the main case where we specialize an interpreter with respect to a source program, we want the residual functions to correspond one-to-one to the functions in the source program. Thus the only thing that should matter when selecting which specialized function to use in a call is which source function the call models—*not* which types the arguments to that source function has. Therefore it is safe to let the type tags be dynamic when specializing the interpreter, and therefore simple generalization strategies can be used.

3.2.2 Termination and binding-time analysis

Like other partial interpreters based on the same principles, MiXIMUM's partial interpratation may fail to terminate.

The good news here is that the situation really *is* analogous to traditional partial interpreters. The only significant difference between our partial interpretation and well-known is that ours produce more verbose residual expressions—but that has obviously no bearing on its termination properties.

This means that we can use the entire accumulated knowledge on how to control partial interpreters with MiXIMUM. In particular we can apply the standard idea of a *binding-time analysis* that constructs a consistently annotated version of the program where it can be determined in advance how large the static prefix of each of the program's values will be. MiXIMUM has (as yet) no integrated binding-time analysis, but there would not be any inherent problems in adding one using existing methods.

The traditional way of using a binding-time analysis is to have a human inspect the behavior of the specialization and adjust the parameters for the binding-time analysis if non-termination occurs (or the specialization is simply too slow). Some recent work on how to make a binding-time analysis *guarantee* termination of the specialization also exists [Glenstrup and Jones 1996; Glenstrup 1999]; this can also be applied to the MiXIMUM model.

$$\frac{}{\top \Subset \tau} \qquad \frac{\psi \Subset \tau_1}{\mathsf{L}\,\psi \Subset \langle \mathsf{L}\,\tau_1 + \mathsf{R}\,\tau_2 \rangle} \qquad \frac{\psi \Subset \tau_2}{\mathsf{R}\,\psi \Subset \langle \mathsf{L}\,\tau_1 + \mathsf{R}\,\tau_2 \rangle}$$

$$\frac{}{c \Subset \mathsf{int}} \qquad \frac{\psi_1 \Subset \tau_1 \qquad \psi_2 \Subset \tau_2}{(\psi_1,\psi_2) \Subset (\tau_1,\tau_2)} \qquad \frac{}{\bot \Subset \tau}$$

Figure 3.4: Inductive definition of the relation used in Lemma 3.2

Another common use of a binding-time analysis is to optimize the efficiency of the partial interpreter itself. This is less relevant for MiXIMUM at present, because the partial interpretation phase takes only about 5% of the time used for the entire specialization. (Of course, in a MiXIMUM-like specializer written with efficiency in mind, that figure might be different, so more conventionally on-line techniques might be relevant).

3.2.3 Correctness

Under the assumption that the partial interpretation terminates, it is easy to reason about its correctness. We are interested in two properties:

- The partial interpretation should *preserve the behavior* of the input program.
- The partial interpretation should *preserve the typability* of the input program.

In both of these cases the "input program" should be understood as the input program stripped of "@" or "LIFT" annotations.

Because our partial interpretation does not change the data representation of the program, the induction lemmas needed to prove its correctness are simple:

Lemma 3.1 *Assume the partial interpreter terminates on program* p *with catalogue* $\mathcal{R}$*. Then, for all* E, $\mathcal{E}$, e, e', v, ψ *with*

- $\mathrm{Dom}\, E = \mathrm{Dom}\, \mathcal{E}$
- $\forall x \in \mathrm{Dom}\, E : E(x) \in [\![\mathcal{E}(x)]\!]$
- $\mathcal{E} \vdash_{p,\mathcal{R}} e : \psi \hookrightarrow e'$

it holds that $E \vdash_p e' \Rightarrow v$ *if and only if* $E \vdash_p e \Rightarrow v$*, and if these equivalent conditions hold, then* $v \in [\![\psi]\!]$*.*

Proof. By induction on the derivation of $E \vdash_p e \Rightarrow v$ (in the "if" direction) or $E \vdash_p e \Rightarrow v$ (in the "only if" direction). Most of the cases are reasonably trivial, although it takes some extra lemmas to show that it does not matter that some specialized expressions are executed in larger environments than the original ones. For time reasons I omit the details. □

Lemma 3.2 *Let* $\Subset$ *be the relation defined in Figure 3.4. Assume that* p *is typable with witness* T*, and that the partial interpreter terminates on program* p *with* *catalogue* $\mathcal{R}$. *Let* $T' = \{ f' \mapsto T(f) \mid \mathcal{R}(f, \psi) = f' \}$. *Then for all* $\mathcal{E}$, Γ, e, ψ, τ, e' *with*

- $\mathrm{Dom}\,\mathcal{E} = \mathrm{Dom}\,\Gamma$
- $\forall x \in \mathrm{Dom}\,\mathcal{E} : \mathcal{E}(x) \Subset \Gamma(x)$
- $\mathcal{E} \vdash_{p,\mathcal{R}} e : \psi \hookrightarrow e'$,
- $\Gamma \vdash_{T} e : \tau$,

it holds that $\Gamma \vdash_{T'} e' : \tau$ *and* $\psi \Subset \tau$.

Proof. By induction on the derivation of $\mathcal{E} \vdash_{p,\mathcal{R}} e : \psi \hookrightarrow e'$. The $\Subset$ relation is necessary for proving the cases concerning integer lifting. Details omitted. □

3.3 Sum reduction

The sum reduction phase works by first doing a simple type-based analysis to find which of the sum types in the program can be removed, then reducing the program guided by the results of the analysis.

3.3.1 Non-standard type analysis

The type-based analysis use a non-standard type system where types are built from the grammar (but can be recursive just like PEL types can):

$$\begin{array}{rcl} \langle \mathit{STyp} \rangle \ni \check{\tau} & ::= & \mathsf{int} \\ & | & \langle \mathsf{L}_\sigma\, \check{\tau} + \mathsf{R}_\sigma\, \check{\tau} \rangle \\ & | & (\check{\tau}, \check{\tau}) \\ \langle \mathit{Sign} \rangle \ni \sigma & ::= & \oplus \mid \ominus \end{array}$$

Intuitively, the σs encode whether a each branch of a sum type occurs in practise. $\oplus$ means that it does, and $\ominus$ means that it does not.

The typing rules are given in Figure 3.5. They are almost identical to the standard PEL typing rules. Indeed, if the program p is (PEL) typable with witness T, there is also at least one V such that

$$\forall [f\,x = e] \in p : \exists \check{\tau}_0, \check{\tau}_1 : \begin{cases} V(f) = \check{\tau}_0 \to \check{\tau}_1 \\ \{x \mapsto \check{\tau}_0\} \vdash_V e : \check{\tau}_1 \end{cases}$$

(compare with Definition 2.2), namely the one produced by simply setting every σ to $\oplus$.

The sum reduction computes a typing with as few $\oplus$s as possible given the constraints in the typing rules and the additional rule that every σ in $V(f_0)$ must be $\oplus$ (but $\ominus$ may occur in $V(f)$ for $f \neq f_0$). This additional rule makes sure the the sum reduction does not change the external behavior of the program.

It is easy to convert a standard PEL typing into a “maxilally $\oplus$-free” typing, because the only thing the typing rules ever say about the σs is that some of the σs must be $\oplus$, and different σs cannot influence each other.

$$\frac{}{\check{\Gamma} \vdash_V c : \mathsf{int}} \qquad \frac{\check{\Gamma} \vdash_V e_1 : \mathsf{int} \qquad \check{\Gamma} \vdash_V e_2 : \mathsf{int}}{\check{\Gamma} \vdash_V (e_1 \diamond e_2) : \mathsf{int}} \diamond \neq =$$

$$\frac{\check{\Gamma} \vdash_V e_1 : \mathsf{int} \qquad \check{\Gamma} \vdash_V e_2 : \mathsf{int}}{\check{\Gamma} \vdash_V (e_1 = e_2) : \langle \mathsf{L}_{\oplus}\, \mathsf{int} + \mathsf{R}_{\oplus}\, \mathsf{int} \rangle} \qquad \frac{\check{\Gamma} \vdash_V e_1 : \check{\tau}_1 \qquad \check{\Gamma} \vdash_V e_2 : \check{\tau}_2}{\check{\Gamma} \vdash_V (e_1, e_2) : (\check{\tau}_1, \check{\tau}_2)}$$

$$\frac{\check{\Gamma} \vdash_V e : (\check{\tau}_1, \check{\tau}_2)}{\check{\Gamma} \vdash_V \mathsf{fst}\, e : \check{\tau}_1} \qquad \frac{\check{\Gamma} \vdash_V e : (\check{\tau}_1, \check{\tau}_2)}{\check{\Gamma} \vdash_V \mathsf{snd}\, e : \check{\tau}_2}$$

$$\frac{\check{\Gamma} \vdash_V e : \check{\tau}_1}{\check{\Gamma} \vdash_V \mathsf{L}\, e : \langle \mathsf{L}_{\oplus}\, \check{\tau}_1 + \mathsf{R}_{\sigma}\, \check{\tau}_2 \rangle} \qquad \frac{\check{\Gamma} \vdash_V e : \check{\tau}_2}{\check{\Gamma} \vdash_V \mathsf{R}\, e : \langle \mathsf{L}_{\sigma}\, \check{\tau}_1 + \mathsf{R}_{\oplus}\, \check{\tau}_2 \rangle}$$

$$\frac{\check{\Gamma} \vdash_V e_0 : \langle \mathsf{L}_{\sigma}\, \check{\tau}_1 + \mathsf{R}_{\sigma}\, \check{\tau}_2 \rangle \qquad \check{\Gamma}\{x_1 \mapsto \check{\tau}_1\} \vdash_V e_1 : \check{\tau} \qquad \check{\Gamma}\{x_2 \mapsto \check{\tau}_2\} \vdash_V e_2 : \check{\tau}}{\check{\Gamma} \vdash_V \mathsf{case}\ e_0\ \mathsf{of} \left\{ \begin{smallmatrix} \mathsf{L}\, x_1 \mapsto e_1 \\ \mathsf{R}\, x_2 \mapsto e_2 \end{smallmatrix} \right\} : \check{\tau}}$$

$$\frac{}{\check{\Gamma} \vdash_V x : \check{\Gamma}(x)} \qquad \frac{}{\check{\Gamma} \vdash_V \mathsf{error} : \check{\tau}}$$

$$\frac{\check{\Gamma} \vdash_V e_0 : \check{\tau}_0}{\check{\Gamma} \vdash_V f\, e_0 : \check{\tau}_1} V(f) = \check{\tau}_0 \to \check{\tau}_1 \qquad \frac{\check{\Gamma} \vdash_V e_0 : \check{\tau}_0 \qquad \check{\Gamma}\{x \mapsto \check{\tau}_0\} \vdash_V e_1 : \check{\tau}_1}{\check{\Gamma} \vdash_V \mathsf{let}\ x = e_0\ \mathsf{in}\ e_1\ \mathsf{end} : \check{\tau}_1}$$

Figure 3.5: Non-standard type system for the sum reduction phase. Note that apart from the signs in the sum types these rules are identical to the ones in Figure 2.3.

$$\begin{aligned}
\llbracket \mathsf{int} \rrbracket &= \mathsf{int} \\
\llbracket \langle \mathsf{L}_{\oplus}\, \check{\tau}_1 + \mathsf{R}_{\ominus}\, \check{\tau}_2 \rangle \rrbracket &= \llbracket \check{\tau}_1 \rrbracket \\
\llbracket \langle \mathsf{L}_{\ominus}\, \check{\tau}_1 + \mathsf{R}_{\oplus}\, \check{\tau}_2 \rangle \rrbracket &= \llbracket \check{\tau}_2 \rrbracket \\
\llbracket \langle \mathsf{L}_{\oplus}\, \check{\tau}_1 + \mathsf{R}_{\oplus}\, \check{\tau}_2 \rangle \rrbracket &= \langle \mathsf{L}\, \llbracket \check{\tau}_1 \rrbracket + \mathsf{R}\, \llbracket \check{\tau}_2 \rrbracket \rangle \\
\llbracket (\check{\tau}_1, \check{\tau}_2) \rrbracket &= (\llbracket \check{\tau}_1 \rrbracket, \llbracket \check{\tau}_2 \rrbracket)
\end{aligned}$$

Figure 3.6: The intuitive relation between sum-reduction types and the types of the reduced program. Note that this definition does not specify what $\llbracket \check{\tau} \rrbracket$ should be if $\check{\tau} = \langle \mathsf{L}_{\ominus}\, \check{\tau}_1 + \mathsf{R}_{\ominus}\, \check{\tau}_2 \rangle$ or, for example, $\check{\tau}$ is the recursive type that solves the equation $\check{\tau} = \langle \mathsf{L}_{\oplus}\, \check{\tau} + \mathsf{R}_{\ominus}\, \mathsf{int} \rangle$. Because no terminating computation can have one of those types, it does not matter for us what $\llbracket \check{\tau} \rrbracket$ is in that case. Therefore any $\llbracket \cdot \rrbracket$ which statisfies the specification will work. (I do have a proof that such a $\llbracket \cdot \rrbracket$ exists, but unfortunately the margin is to small to write it down...)

$$
\begin{array}{rcl}
\llbracket \mathsf{c} \rrbracket &=& \mathsf{c} \\
\llbracket (e_1 \diamond e_2) \rrbracket &=& (\llbracket e_1 \rrbracket \diamond \llbracket e_2 \rrbracket) \\
\llbracket (e_1, e_2) \rrbracket &=& (\llbracket e_1 \rrbracket, \llbracket e_2 \rrbracket) \\
\llbracket \mathsf{fst}\, e \rrbracket &=& \mathsf{fst}\, \llbracket e \rrbracket \\
\llbracket \mathsf{snd}\, e \rrbracket &=& \mathsf{snd}\, \llbracket e \rrbracket \\
\llbracket (\mathsf{L}\, e)^{\oplus\oplus} \rrbracket &=& \mathsf{L}\, \llbracket e \rrbracket \\
\llbracket (\mathsf{L}\, e)^{\oplus\ominus} \rrbracket &=& \llbracket e \rrbracket \\
\llbracket (\mathsf{R}\, e)^{\oplus\oplus} \rrbracket &=& \mathsf{R}\, \llbracket e \rrbracket \\
\llbracket (\mathsf{R}\, e)^{\ominus\oplus} \rrbracket &=& \llbracket e \rrbracket \\
\left\llbracket \mathsf{case}\ e_0{}^{\oplus\oplus}\ \mathsf{of} \left\{ \begin{smallmatrix} \mathsf{L}\, x_1 \mapsto e_1 \\ \mathsf{R}\, x_2 \mapsto e_2 \end{smallmatrix} \right\} \right\rrbracket &=& \mathsf{case}\ \llbracket e_0 \rrbracket\ \mathsf{of} \left\{ \begin{smallmatrix} \mathsf{L}\, x_1 \mapsto \llbracket e_1 \rrbracket \\ \mathsf{R}\, x_2 \mapsto \llbracket e_2 \rrbracket \end{smallmatrix} \right\} \\
\left\llbracket \mathsf{case}\ e_0{}^{\oplus\ominus}\ \mathsf{of} \left\{ \begin{smallmatrix} \mathsf{L}\, x_1 \mapsto e_1 \\ \mathsf{R}\, x_2 \mapsto e_2 \end{smallmatrix} \right\} \right\rrbracket &=& \mathsf{let}\ x_1 = \llbracket e_0 \rrbracket\ \mathsf{in}\ \llbracket e_1 \rrbracket\ \mathsf{end} \\
\left\llbracket \mathsf{case}\ e_0{}^{\ominus\oplus}\ \mathsf{of} \left\{ \begin{smallmatrix} \mathsf{L}\, x_1 \mapsto e_1 \\ \mathsf{R}\, x_2 \mapsto e_2 \end{smallmatrix} \right\} \right\rrbracket &=& \mathsf{let}\ x_2 = \llbracket e_0 \rrbracket\ \mathsf{in}\ \llbracket e_2 \rrbracket\ \mathsf{end} \\
\left\llbracket \mathsf{case}\ e_0{}^{\ominus\ominus}\ \mathsf{of} \left\{ \begin{smallmatrix} \mathsf{L}\, x_1 \mapsto e_1 \\ \mathsf{R}\, x_2 \mapsto e_2 \end{smallmatrix} \right\} \right\rrbracket &=& \langle\!\langle \llbracket e_0 \rrbracket \rangle\!\rangle\, \mathsf{error} \\
\llbracket x \rrbracket &=& x \\
\llbracket \mathsf{error} \rrbracket &=& \mathsf{error} \\
\llbracket f\, e \rrbracket &=& f\, \llbracket e \rrbracket \\
\llbracket \mathsf{let}\ x = e_0\ \mathsf{in}\ e_1\ \mathsf{end} \rrbracket &=& \mathsf{let}\ x = \llbracket e_0 \rrbracket\ \mathsf{in}\ \llbracket e_1 \rrbracket\ \mathsf{end}
\end{array}
$$

Figure 3.7: Reduction rules for the sum reduction phase. The expressions inside the $\llbracket\,\rrbracket$*s are supposed to be annotated with sum-reduction types; the pattern "*$e^{\sigma_1\sigma_2}$*" matches an* e *that annotated with type* $\langle \mathsf{L}_{\sigma_1}\, \check{\tau}_1 + \mathsf{R}_{\sigma_2}\, \check{\tau}_2 \rangle$ *for some* $\check{\tau}_{1,2}$.

3.3.2 Reduction

What the sum reduction phase does is perhaps best understood by looking at Figure 3.6, which (almost) defines a function $\llbracket\cdot\rrbracket : \langle \mathit{STyp} \rangle \rightarrow \langle \mathit{Typ} \rangle$ from sum-reduction types to the standard types of the *reduced* program. The interesting rules are the two that define that a sum type *disappears* if only one of the branches are annoated with an $\oplus$.

Figure 3.7 defines a reduction mapping $\llbracket\cdot\rrbracket$ from expressions annotated with sum reduction types to plain PEL expressions. The definition of that mapping is a simple consequences of the mapping from Figure 3.6.

The sum reduction now simply consists of replacing every function definition $f\,x = e$ with $f\,x = \llbracket e \rrbracket$.

3.3.3 Discussion

The primary virtue of the sum reduction is its simplcity. Of course it fulfills the basic requirements of preserving the semantics and typability of the program, and it is also easy to see that it never *decreases* the efficiency of the program. But that is about all of its nice properties.

In particular the sum reduction is not *idempotent*. That is, doing sum reduction twice on a program may improve it more than doing sum reduction once. The reason for this is that sum reduction sometimes removes entire expressions from the program, and one of the expressions that get removed might be the only source of a $\oplus$ annotation that prevented some *other* expression from being removed.

It is not difficult to write down a type system for a more powerful, idempotent sum-reduction analysis, but type inference would become more complicated, probably needing control structures similar to the "demons" of Hughes' type specializer.

On the other hand, extension to a polymorphic type system appears not to be problematic (except for the well-known problem of how to infer polymorphic types in languages where all functions can in principle be mutually recursive).

3.3.4 Correctness

I claim that the sum reduction is correct (that is, the output program is always typable and the behavior of its main function is identical to the behavior of the input program's main function), but I don't even have time to sketch a proof. Sigh.

3.4 Product reduction

The product reduction phase is quite similar to the sum reduction phase: It first does a simple type-based analysis to find which of the product types in the program can be elimiates. Then it then reduces the program guided by the results of the analysis.

3.4.1 Non-standard type analysis

Product-reduction types are built from the grammar

$$\begin{array}{rcl} \langle \mathit{PTyp} \rangle \ni \widehat{\tau} & ::= & \mathsf{int}_\sigma \\ & | & \langle \mathsf{L}\,\widehat{\tau} + \mathsf{R}\,\widehat{\tau} \rangle \\ & | & (\widehat{\tau},\widehat{\tau}) \\ \langle \mathit{Sign} \rangle \ni \sigma & ::= & \oplus \mid \ominus \end{array}$$

and can be recursive in the same manner of standard PEL types.

The typing rules, in Figure 3.8, are similar in nature to the typing rules for the sum-reduction types. Also like in the sum reduction phase, the product reduction analysis consists of computing a "maximally $\oplus$ free" typing Λ such that $\Lambda(\mathsf{f}_0)$ contains no $\ominus$s.

3.4.2 Reduction

The goal of the product reduction is to remove "unnecessary" components of pairs. The job of the product-reduction types is to approximate which values are "unnecessary":

$$\frac{}{\widehat{\Gamma} \vdash_\Lambda c : \mathsf{int}_\sigma} \qquad \frac{\widehat{\Gamma} \vdash_\Lambda e_1 : \mathsf{int}_\oplus \qquad \widehat{\Gamma} \vdash_\Lambda e_2 : \mathsf{int}_\oplus}{\widehat{\Gamma} \vdash_\Lambda (e_1 \diamond e_2) : \mathsf{int}_\sigma} \diamond \neq =$$

$$\frac{\widehat{\Gamma} \vdash_\Lambda e_1 : \mathsf{int}_\oplus \qquad \widehat{\Gamma} \vdash_\Lambda e_2 : \mathsf{int}_\oplus}{\widehat{\Gamma} \vdash_\Lambda (e_1 = e_2) : \langle \mathsf{L}\,\mathsf{int}_\sigma + \mathsf{R}\,\mathsf{int}_\sigma \rangle} \qquad \frac{\widehat{\Gamma} \vdash_\Lambda e_1 : \widehat{\tau}_1 \qquad \widehat{\Gamma} \vdash_\Lambda e_2 : \widehat{\tau}_2}{\widehat{\Gamma} \vdash_\Lambda (e_1, e_2) : (\widehat{\tau}_1, \widehat{\tau}_2)}$$

$$\frac{\widehat{\Gamma} \vdash_\Lambda e : (\widehat{\tau}_1, \widehat{\tau}_2)}{\widehat{\Gamma} \vdash_\Lambda \mathsf{fst}\, e : \widehat{\tau}_1} \qquad \frac{\widehat{\Gamma} \vdash_\Lambda e : (\widehat{\tau}_1, \widehat{\tau}_2)}{\widehat{\Gamma} \vdash_\Lambda \mathsf{snd}\, e : \widehat{\tau}_2}$$

$$\frac{\widehat{\Gamma} \vdash_\Lambda e : \widehat{\tau}_1}{\widehat{\Gamma} \vdash_\Lambda \mathsf{L}\, e : \langle \mathsf{L}\,\widehat{\tau}_1 + \mathsf{R}\,\widehat{\tau}_2 \rangle} \qquad \frac{\widehat{\Gamma} \vdash_\Lambda e : \widehat{\tau}_2}{\widehat{\Gamma} \vdash_\Lambda \mathsf{R}\, e : \langle \mathsf{L}\,\widehat{\tau}_1 + \mathsf{R}\,\widehat{\tau}_2 \rangle}$$

$$\frac{\widehat{\Gamma} \vdash_\Lambda e_0 : \langle \mathsf{L}\,\widehat{\tau}_1 + \mathsf{R}\,\widehat{\tau}_2 \rangle \qquad \widehat{\Gamma}\{x_1 \mapsto \widehat{\tau}_1\} \vdash_\Lambda e_1 : \widehat{\tau} \qquad \widehat{\Gamma}\{x_2 \mapsto \widehat{\tau}_2\} \vdash_\Lambda e_2 : \widehat{\tau}}{\widehat{\Gamma} \vdash_\Lambda \mathsf{case}\ e_0\ \mathsf{of} \left\{ \begin{smallmatrix} \mathsf{L}\, x_1 \mapsto e_1 \\ \mathsf{R}\, x_2 \mapsto e_2 \end{smallmatrix} \right\} : \widehat{\tau}}$$

$$\frac{}{\widehat{\Gamma} \vdash_\Lambda x : \widehat{\Gamma}(x)} \qquad \frac{}{\widehat{\Gamma} \vdash_\Lambda \mathsf{error} : \widehat{\tau}}$$

$$\frac{\widehat{\Gamma} \vdash_\Lambda e_0 : \widehat{\tau}_0}{\widehat{\Gamma} \vdash_\Lambda f\, e_0 : \widehat{\tau}_1} \Lambda(f) = \widehat{\tau}_0 \to \widehat{\tau}_1 \qquad \frac{\widehat{\Gamma} \vdash_\Lambda e_0 : \widehat{\tau}_0 \qquad \widehat{\Gamma}\{x \mapsto \widehat{\tau}_0\} \vdash_\Lambda e_1 : \widehat{\tau}_1}{\widehat{\Gamma} \vdash_\Lambda \mathsf{let}\ x = e_0\ \mathsf{in}\ e_1\ \mathsf{end} : \widehat{\tau}_1}$$

Figure 3.8: Non-standard type system for the product reduction phase. Note that apart from the signs in the int*types these rules are identical to the ones in Figure 2.3.*

Definition 3.3 *The set* $\mathbb{T}\mathrm{r}$ *of* ***trivial types*** *is the smallest subset of* $\langle PTyp \rangle$ *that contains* $\mathsf{int}_\ominus$ *and is closed under formation of product types.*

In case analyses on product-reduction types we write $\widehat{\tau}^\oplus$ to mean any $\widehat{\tau} \notin \mathbb{T}\mathrm{r}$ and $\widehat{\tau}^\ominus$ to mean any $\widehat{\tau} \in \mathbb{T}\mathrm{r}$.

The actual reduction is very similar to the reduction in the sum reduction phase: Figure 3.9 (almost) defines the relation between product-reduction types and the types in the reduced program, and Figure 3.10 defines the corresponding reduction function on expressions.

3.4.3 Discussion

The similarity between Figures 3.10 and 3.7 (and to an even higher degree between Figures 3.9 and 3.6) suggests that sum reduction and product reduction are closely related transformations. In both cases the reduction consists of removing certain applications of a binary type constructor with one of the operands. The actual reduction mappings are straightforward adaptations of that idea to the various introduction and elimintation operations in the language.

The two transformation differ, however, in how they select the type nodes that can be eliminated. Both of the type-based analyses consist of decorating

$$
\begin{array}{rcl}
\llbracket \hat{\tau}^{\ominus} \rrbracket & = & \mathsf{int} \\
\llbracket \mathsf{int}_{\oplus} \rrbracket & = & \mathsf{int} \\
\llbracket \langle \mathsf{L}\,\hat{\tau}_1 + \mathsf{R}\,\hat{\tau}_2 \rangle \rrbracket & = & \langle \mathsf{L}\,\llbracket \hat{\tau}_1 \rrbracket + \mathsf{R}\,\llbracket \hat{\tau}_2 \rrbracket \rangle \\
\llbracket (\hat{\tau}_1^{\oplus}, \hat{\tau}_2^{\oplus}) \rrbracket & = & (\llbracket \hat{\tau}_1 \rrbracket, \llbracket \hat{\tau}_2 \rrbracket) \\
\llbracket (\hat{\tau}_1^{\oplus}, \hat{\tau}_2^{\ominus}) \rrbracket & = & \llbracket \hat{\tau}_1 \rrbracket \\
\llbracket (\hat{\tau}_1^{\ominus}, \hat{\tau}_2^{\oplus}) \rrbracket & = & \llbracket \hat{\tau}_2 \rrbracket
\end{array}
$$

Figure 3.9: The intuitive relation between product-reduction types and the types of the reduced program. As was the case for Figure 3.6 this definition does not completely specify, for example, what $\llbracket \hat{\tau} \rrbracket$ should be if $\hat{\tau}$ is the recursive type that solves $\hat{\tau} = (\hat{\tau}, \mathsf{int}_{\ominus})$.

$$
\begin{array}{rcl}
\llbracket c \rrbracket & = & c \\
\llbracket (e_1 \diamond e_2) \rrbracket & = & (\llbracket e_1 \rrbracket \diamond \llbracket e_2 \rrbracket) \\
\left\llbracket (e_1, e_2)^{\oplus\oplus} \right\rrbracket & = & (\llbracket e_1 \rrbracket, \llbracket e_2 \rrbracket) \\
\left\llbracket (e_1, e_2)^{\oplus\ominus} \right\rrbracket & = & \llbracket e_1 \rrbracket \,\langle\!\langle \llbracket e_2 \rrbracket \rangle\!\rangle \\
\left\llbracket (e_1, e_2)^{\ominus\oplus} \right\rrbracket & = & \langle\!\langle \llbracket e_1 \rrbracket \rangle\!\rangle \,\llbracket e_2 \rrbracket \\
\left\llbracket (e_1, e_2)^{\ominus\ominus} \right\rrbracket & = & \langle\!\langle \llbracket e_1 \rrbracket \rangle\!\rangle \,\llbracket e_2 \rrbracket \\
\llbracket \mathsf{fst}\; e^{\oplus\oplus} \rrbracket & = & \mathsf{fst}\; \llbracket e \rrbracket \\
\llbracket \mathsf{fst}\; e^{\oplus\ominus} \rrbracket & = & \llbracket e \rrbracket \\
\llbracket \mathsf{fst}\; e^{\ominus\oplus} \rrbracket & = & \langle\!\langle \llbracket e \rrbracket \rangle\!\rangle\, 0 \\
\llbracket \mathsf{fst}\; e^{\ominus\ominus} \rrbracket & = & \llbracket e \rrbracket \\
\llbracket \mathsf{snd}\; e^{\oplus\oplus} \rrbracket & = & \mathsf{fst}\; \llbracket e \rrbracket \\
\llbracket \mathsf{snd}\; e^{\oplus\ominus} \rrbracket & = & \langle\!\langle \llbracket e \rrbracket \rangle\!\rangle\, 0 \\
\llbracket \mathsf{snd}\; e^{\ominus\oplus} \rrbracket & = & \llbracket e \rrbracket \\
\llbracket \mathsf{snd}\; e^{\ominus\ominus} \rrbracket & = & \llbracket e \rrbracket \\
\llbracket \mathsf{L}\; e \rrbracket & = & \mathsf{L}\; \llbracket e \rrbracket \\
\llbracket \mathsf{R}\; e \rrbracket & = & \mathsf{R}\; \llbracket e \rrbracket \\
\left\llbracket \mathsf{case}\; e_0 \;\mathsf{of} \left\{ \begin{smallmatrix} \mathsf{L}\, x_1 \mapsto e_1 \\ \mathsf{R}\, x_2 \mapsto e_2 \end{smallmatrix} \right\} \right\rrbracket & = & \mathsf{case}\; \llbracket e_0 \rrbracket \;\mathsf{of} \left\{ \begin{smallmatrix} \mathsf{L}\, x_1 \mapsto \llbracket e_1 \rrbracket \\ \mathsf{R}\, x_2 \mapsto \llbracket e_2 \rrbracket \end{smallmatrix} \right\} \\
\llbracket x \rrbracket & = & x \\
\llbracket \mathsf{error} \rrbracket & = & \mathsf{error} \\
\llbracket f\; e \rrbracket & = & f\; \llbracket e \rrbracket \\
\llbracket \mathsf{let}\; x = e_0 \;\mathsf{in}\; e_1 \;\mathsf{end} \rrbracket & = & \mathsf{let}\; x = \llbracket e_0 \rrbracket \;\mathsf{in}\; \llbracket e_1 \rrbracket \;\mathsf{end}
\end{array}
$$

Figure 3.10: Reduction rules for the sum reduction phase. The expressions inside the $\llbracket\,\rrbracket$s are supposed to be annotated with product-reduction types; the pattern "$e^{\sigma_1 \sigma_2}$" matches an e that is annotated with type $(\hat{\tau}_1^{\sigma_1}, \hat{\tau}_2^{\sigma_2})$.

some of the type nodes in a standard PEL typing with σs, but that similarity turns out to be superficial on closer inspection.

I think it would be valuable to view the sum and product reductions as each other's *duals*, corresponding to the categorical duality between sums and products (or, though the Curry-Howard isomorphism, to the logical duality between disjunction and conjunction).

When viewed in this way, the differences between the type-based analyses begin to make sense: the sum reduction's type system computes how a value with a sum type is *created*, whereas the product reduction's type system computes how a value with a product type is *used*.

Still, that does not completely explain the differences between the two analyses. The most direct dual of the sum reduction would be a product reduction where a product type could be eliminated if expressions of that type never occured as arguments to fst, or to snd. Such a transformation would indeed be possible and sound, but it would not be strong enough for MiXIMUM to be optimal.

Going in the other direction, the direct dual of the product reduction as presented here would be a sum reduction that allowed, for example, the first summand of

$$\langle \mathsf{L}_\oplus \, \langle \mathsf{L}_\ominus \, \check{\tau}_{11} + \mathsf{R}_\ominus \, \check{\tau}_{12} \rangle + \mathsf{R}_\oplus \, \check{\tau}_2 \rangle$$

to be removed because even though there is a $\mathsf{L}\,e$ expression that creates that side of the sum somewhere in the program, that expression is never evaluated because its argument cannot exist. Such a sum reduction would also be possible and sound, but the added complexity would not make MiXIMUM more optimal than it already is.

One can readily imagine making the product reduction stronger than it is presently. The current product reduction will not remove pair components that contain sum types at all, but doing so would be harmless it the sum type in question was never used in a case expression. Indeed, the product reduction in an early version of MiXIMUM attached a σ to each sum type and handled sum types similarly to ints. This turned out not to be used when specializing the PEL self-interpreter, either.

3.4.4 Correctness

As for the sum reduction (mutatis mutandis).

3.5 Let reduction

The sum and product reduction phases are both essentially inter-procedural. In the context of specializing the PEL self-interpreter their main task have been to reduce the argument types of the various specialized versions of the `eval` function such that they become identical to the types of the corresponding functions in the source program.

These phases, however, do not care much about the efficiency of the body of each function, apart from what comes naturally as by-products of the global optimizations. In particular, all of the preceding phases have a tendency of creating very many trivial let bindings. The partial interpretation uses a let binding for each function call it unfolds. The sum reduction replaces case analysis on removable sum types with let bindings. The product reduction uses let bindings (disguised as "$\langle\!\langle e \rangle\!\rangle\, e$" expressions) to avoid throwing any side-effecting expressions away. The most prominent job of the let reduction phase is to find out which of all these let bindings do any useful work and eliminate the remainder.

Not less important, however, is the final elimination of the tuples that are the remainder of the interpreter's environment list. For example, when the interpreter is specialized with respect to the expression

$$\mathsf{let}\ x_5 = 5\ \mathsf{in}\ \mathsf{let}\ x_7 = 7\ \mathsf{in}\ (x_5 + x_7)\ \mathsf{end}\ \mathsf{end}$$

after sum and product reduction the specialized code looks like (in addition to a lot of trivial let bindings)

$$\mathsf{let}\ x = 5\ \mathsf{in}\ \mathsf{let}\ x = (7,x)\ \mathsf{in}\ (\mathsf{snd}\,x + \mathsf{fst}\,x)\ \mathsf{end}\ \mathsf{end}$$

—that is, each variable binding in the original program has resulted in a binding of a tuple containing the values of all variables in scope at the point of the binding. The let reduction phase contains dedicated rules to simplify such constructions.

The let reduction phase is purely intra-procedural—in fact, it consists of nothing more than a series of local "peephole" optimizations.

3.5.1 Selector movement

The first step of the let reduction is to move pair selectors inside variable bindings by repeated applications of the rewrite rules

$$\begin{array}{rcl}
\mathsf{fst}\,\mathsf{let}\ x = e_0\ \mathsf{in}\ e_1\ \mathsf{end} & \hookrightarrow & \mathsf{let}\ x = e_0\ \mathsf{in}\ \ \mathsf{fst}\,e_1\ \mathsf{end} \\
\mathsf{snd}\,\mathsf{let}\ x = e_0\ \mathsf{in}\ e_1\ \mathsf{end} & \hookrightarrow & \mathsf{let}\ x = e_0\ \mathsf{in}\ \ \mathsf{snd}\,e_1\ \mathsf{end} \\
\mathsf{fst}\,\mathsf{case}\ e_0\ \mathsf{of} \left\{ \begin{array}{lcl} \mathsf{L}\,x_1 & \mapsto & e_1 \\ \mathsf{R}\,x_2 & \mapsto & e_2 \end{array} \right\} & \hookrightarrow & \mathsf{case}\ e_0\ \mathsf{of} \left\{ \begin{array}{lcl} \mathsf{L}\,x_1 & \mapsto & \mathsf{fst}\,e_1 \\ \mathsf{R}\,x_2 & \mapsto & \mathsf{fst}\,e_2 \end{array} \right\} \\
\mathsf{snd}\,\mathsf{case}\ e_0\ \mathsf{of} \left\{ \begin{array}{lcl} \mathsf{L}\,x_1 & \mapsto & e_1 \\ \mathsf{R}\,x_2 & \mapsto & e_2 \end{array} \right\} & \hookrightarrow & \mathsf{case}\ e_0\ \mathsf{of} \left\{ \begin{array}{lcl} \mathsf{L}\,x_1 & \mapsto & \mathsf{snd}\,e_1 \\ \mathsf{R}\,x_2 & \mapsto & \mathsf{snd}\,e_2 \end{array} \right\}
\end{array}$$

The point here is that expressions like "$\mathsf{fst}\,x$" enable later optimzations that are not possible with "$\mathsf{fst}\,\mathsf{let}\ \ldots = \ldots\ \mathsf{in}\ x\ \mathsf{end}$".

This transformation is not actually needed for optimal specialization of the PEL specializer, so it really ought to have been removed in MiXIMUM.

3.5.2 The main phase

Then the main phase of the let reduction follows. It is a bottom-up simplifications that simplifies the operand expressions of each let binding before trying to simplify the binding itself (every other expression form is left unchanged at first).

For each expression of the form

$$\mathsf{let}\ x = e_0\ \mathsf{in}\ e_1\ \mathsf{end}$$

the first applicable of the following transformations is applied:

$$
\begin{array}{rcl}
\langle\!\langle c \rangle\!\rangle\; e_1 & \hookrightarrow & e_1 \\
\langle\!\langle (e'_1 \diamond e'_2) \rangle\!\rangle\; e_1 & \hookrightarrow & \langle\!\langle e'_1 \rangle\!\rangle^\dagger\, \langle\!\langle e'_2 \rangle\!\rangle^\dagger\; e_1 \\
\langle\!\langle (e'_1 , e'_2) \rangle\!\rangle\; e_1 & \hookrightarrow & \langle\!\langle e'_1 \rangle\!\rangle^\dagger\, \langle\!\langle e'_2 \rangle\!\rangle^\dagger\; e_1 \\
\langle\!\langle \mathsf{fst}\; e' \rangle\!\rangle\; e_1 & \hookrightarrow & \langle\!\langle e' \rangle\!\rangle^\dagger\; e_1 \\
\langle\!\langle \mathsf{snd}\; e' \rangle\!\rangle\; e_1 & \hookrightarrow & \langle\!\langle e' \rangle\!\rangle^\dagger\; e_1 \\
\langle\!\langle \mathsf{L}\; e' \rangle\!\rangle\; e_1 & \hookrightarrow & \langle\!\langle e' \rangle\!\rangle^\dagger\; e_1 \\
\langle\!\langle \mathsf{R}\; e' \rangle\!\rangle\; e_1 & \hookrightarrow & \langle\!\langle e' \rangle\!\rangle^\dagger\; e_1 \\
\langle\!\langle \mathsf{case}\; e'_0\; \mathsf{of} \left\{ \begin{smallmatrix} \mathsf{L}\, x_1 \mapsto e'_1 \\ \mathsf{R}\, x_2 \mapsto e'_2 \end{smallmatrix} \right\} \rangle\!\rangle\; e_1 & \hookrightarrow & \begin{cases} \langle\!\langle e'_0 \rangle\!\rangle^\dagger\; e_1 \\ \quad \text{(if } e_1 \text{ and } e_2 \text{ have no side effects)} \\ \langle\!\langle \mathsf{case}\; e'_0\; \mathsf{of} \left\{ \begin{smallmatrix} \mathsf{L}\, x_1 \mapsto \langle\!\langle e'_1 \rangle\!\rangle^\dagger\, 0 \\ \mathsf{R}\, x_2 \mapsto \langle\!\langle e'_2 \rangle\!\rangle^\dagger\, 0 \end{smallmatrix} \right\} \rangle\!\rangle\; e_1 \\ \quad \text{(otherwise)} \end{cases} \\
\langle\!\langle x \rangle\!\rangle\; e_1 & \hookrightarrow & e_1 \\
\langle\!\langle \mathsf{error} \rangle\!\rangle\; e_1 & \hookrightarrow & \mathsf{error} \\
\langle\!\langle f\; e' \rangle\!\rangle\; e_1 & \hookrightarrow & \langle\!\langle f\; e' \rangle\!\rangle\; e_1 \\
\langle\!\langle \mathsf{let}\; x = e'_0\; \mathsf{in}\; e'_1\; \mathsf{end} \rangle\!\rangle\; e_1 & \hookrightarrow & \mathsf{let}\; x = e'_0\; \mathsf{in}\; \langle\!\langle e'_1 \rangle\!\rangle^\dagger\; e_1\; \mathsf{end}^\dagger
\end{array}
$$

Figure 3.11: Reduction of let *bindings where the bound variable is never used.*

- If e_0 is a variable, then the let binding is eliminated—that is, the entire let expression is replaced by $e_1[e_0/x]$.
- If e_0 is a pair construction (e_{01}, e_{02}) *and* the only (free) occurences of x in e_1 are as the operand in fst or snd expressions, then the let expression is replaced with

  $$\mathsf{let}\; x_1 = e_{01}\; \mathsf{in}\; \mathsf{let}\; x_2 = e_{02}\; \mathsf{in}\; e_1[x_1/\mathsf{fst}\, x][x_2/\mathsf{snd}\, x]\; \mathsf{end}\; \mathsf{end}$$

  (where x_1 and x_2 are fresh variables). If e_1 contained subexpressions of the form

  $$\mathsf{let}\; x' = \mathsf{fst}\, x\; \mathsf{in}\; e'\; \mathsf{end}$$

  these become egligible for elimination as a result of this substitution; they are thus eliminated. Finally, the two new bindings of x_1 and x_2 are recursively simplified.
- If x does not appear at all in e_1, then the let binding is reduced according to the rules in Figure 3.11. The new let bindings in the reduced expression that are marked with a "†" in Figure 3.11 are then recursively simplified.
- If x is referenced *exactly once* while e_1 evaluated (that is, if x occur in one of the brances of a case expression it must occur once in the other branch, too), *and* no side effects (*i.e.*, function calls or error expressions) in e_1 are evaluated before that reference, then the let binding is eliminated.
- If x is referenced *at most once* while e_1 is evaluated *and* no side effects occur in e_0, then the let binding is eliminated.
- Otherwise the let binding is left in the program.

3.5.3 Discussion

The simplifications in the let reduction phase are not selected in a very systematic way. Even more than the other phases in MiXIMUM, the let reduction phase simply follows the design principle of including whatever turned out to be necessary for achieving optimal specialization of the PEL self-interpreter.

For example, there is no need in this particular application to include a rule to reduce $\mathsf{fst}\,(5,17)$ to 5. It proved to be sufficient to be able to remove pair constructions in the e_0 position of a let binding. Hughes' type specializer, on the other hand, obtains the corresponding effect by reducing selectors that are applied to pair construction.

3.5.4 Correctness

The correctness of the let reduction follows from the fact that each of the individual simplifications preserves semantics and typability.

3.6 Cheating

After the let reduction phase, the very last step in MiXIMUM consists of applying the reduction rule

$$\mathsf{case}\ (e_1 = e_2)\ \mathsf{of} \left\{ \begin{array}{lcl} \mathsf{L}\,x_1 & \mapsto & \mathsf{L}\,0 \\ \mathsf{R}\,x_2 & \mapsto & \mathsf{R}\,0 \end{array} \right\} \hookrightarrow (e_1 = e_2)$$

wherever the left-hand side of the rule occurs. Expressions of this shape arise when specializing the PEL self-interpreter with respect to equality tests. The case expression comes from the self-interpreter where its job is to analyse the result of the equality test so that an equivalent encoded value can be built.

I call this phase the “cheat” phase because I think the problem is really a symptom of a more general problem with specializing interpreter code that encodes complex already existing terms with the universal encoding. The equality test is simply the only place the problem surfaces with the PEL self-interpreter, and I think that a more general solution ought to be found before we can say that the problem of specializing interpreters in strongly typed languages is fully solved.

Chapter 4

Optimal specialization of the PEL self-interpreter

As evidence that MiXIMUM is optimal, I have used it to specialize Welinder's self-interpreter for PEL with respect to itself. That is, I computed

$$[\![spec]\!](pelint, pelint) \tag{4.1}$$

which, if MiXIMUM is "strong enough" according to equation (1.3) on page 4, ought to produce essentially the text of *pelint* as output.

The formal definition of optimality requires that the self-interpreter can be specialized with *any* correct program as its static input. That is, of course, impossible to test in practise, so our options are to provide a mathematical proof of the optimality of the specializer, or to settle for experimental evidence. I do not think the time needed to produce an actual proof of optimality would be well invested at this time. We are still in the process of determining how far the ideas behind MiXIMUM can be stretched at all; in essence we're investigating what would be interesting to prove at all, and work spent on the details of a proof of one system might well be wasted as soon as we discover how to build a better, but different system.

Thus, for the time being, we shall be satisfied with experiments to establish the optimality of the specializer. That leaves the question of which programs we should try to use as static inputs for the self-interpreter.

The self-interpreter itself turns out to be a good candidate. One reason for this is simply that it is a reasonably large program which is already at hand and which does something interesting. Another argument is that the self-interpreter can be expected to exercise every feature of the language, which means that using it as the static input for itself ought to reveal if portions of the interpeter cannot specialized optimally.

4.1 Types and encodings

In real life, the optimality test is not as simple as it looks in (4.1) or (1.3). That is because we work with a typed language and a typed self-interpreter.

Normally when the Futamura projections are derived, the definition of a self-interpreter is supposed to be

$$[\![\mathit{sint}]\!](\mathrm{p},\mathrm{d}) = [\![\mathrm{p}]\!](\mathrm{d}) \tag{4.2}$$

That does not work well with a typed language such as our typed version of PEL. Here, the type of the input to a program is determined by the program itself. For example, a program might expect its input to be an integer; then it cannot also be used with a pair of integers as its input (at least not if it does anyting nontrivial with its input). However, another program might indeed expect a pair of integers as its input. If a self-interpreter *sint* were to satisfy (4.2) for both of these programs, it would have to accept input whose second component could be either an integer or a pair of integers—and then the self-interpreter itself would not be well-typed.

The solution adopted by the PEL self-interpreter is to accept the interpreted program's input (and deliver its output) in the same encoding as it uses internally. Let us call the type that describes values in this encoding *Univ*.

Then, for each PEL type τ there exist functions $\mathit{encode}_\tau : \tau \to \mathit{Univ}$ and $\mathit{decode}_\tau : \mathit{Univ} \to \tau$ which encodes, respectively decodes, values of type τ. Note that decode_τ is a partial function: It fails if its input is an encoding of a value that is not in $[\![\tau]\!]$.

For each τ, encode_τ and decode_τ can be defined in PEL. Slightly abusing notation, we'll also call the PEL functions that realise the mathematical functione encode_τ and decode_τ. Note that the PEL functions depend on τ; there is no general *encode* or *decode* that is expressible in PEL.

Now, we can write a better algebraic specifiction of a self interpreter such as *pelint*. *pelint* itself has type $(\mathit{Pgm},\mathit{Univ}) \to \mathit{Univ}$ (where *Pgm* is some type that can represent program texts). The defining property of *pelint* is that for each program p of type $\tau_1 \to \tau_2$ and each input $\mathrm{d} \in [\![\tau_2]\!]$,

$$\mathit{decode}_{\tau_2}([\![\mathit{pelint}]\!](\mathrm{p}, \mathit{encode}_{\tau_1}(\mathrm{d}))) = [\![\mathrm{p}]\!](\mathrm{d}) \tag{4.3}$$

If we combine this with (1.1), we arrive at

$$\mathit{decode}_{\tau_2}([\![[\![\mathit{spec}]\!](\mathit{pelint},\mathrm{p})]\!](\mathit{encode}_{\tau_1}(\mathrm{d}))) = [\![\mathrm{p}]\!](\mathrm{d}) \tag{4.4}$$

or, equivalently,

$$[\![[\![\mathit{spec}]\!](\mathit{pelint},\mathrm{p})]\!](\mathrm{d}') = \mathit{encode}_{\tau_2}([\![\mathrm{p}]\!](\mathit{decode}_{\tau_1}(\mathrm{d}'))) \tag{4.5}$$

The consequence of these equations is that we cannot hope for $[\![\mathit{spec}]\!](\mathit{pelint},\mathrm{p})$ to be comparable to p itself, because the equations say they behave differently.

My conclusion is that to implement the first Futamura projection for a typed langage, one should not compute $[\![\mathit{spec}]\!](\mathit{pelint},\mathrm{p})$ but

$$[\![\mathit{spec}]\!](\mathit{decode}_{\tau_2} \circ \mathit{pelint} \circ (\mathit{Id} \times \mathit{encode}_{\tau_1}),\mathrm{p}) \tag{4.6}$$

where $\mathit{decode}_{\tau_2} \circ \mathit{pelint} \circ (\mathit{Id} \times \mathit{encode}_{\tau_1})$ is the symbolic composition of *pelint* with the PEL versions of the encoding and decoding functions, that is,

$$\begin{array}{rcl}
f_0\,x & = & f_\omega\, f_0'\,(\mathsf{fst}\,x, f_\alpha\,\mathsf{snd}\,x) \\
f_\alpha\,x & = & (\text{PEL definition of } \mathit{encode}_{\tau_1}(x)) \\
f_\omega\,x & = & (\text{PEL definition of } \mathit{decode}_{\tau_2}(x)) \\
f_0'\,x & = & (\text{the text of the general PEL self-interpreter})
\end{array}$$

With some algebraic juggling we can see that the output of (4.6) really ought to behave the same as p itself. Thus when using the specializer in that way we can hope for optimality.

4.2 Specializing the self-interpreter to itself

When we instantiate p in (4.6) to *pelint* we get

$$[\![spec]\!](decode_{Univ} \circ pelint \circ (Id \times encode_{(Pgm,Univ)}), pelint) \qquad (4.7)$$

I have constructed $decode_{Univ} \circ pelint \circ (Id \times encode_{(Pgm,Univ)})$ by hand and trivially specialized it to the text of the original *pelint* by hand. I then ran the MiXIMUM prototype on the trivially specialized program.

The output is not α-equivalent to the original *pelint*, but has the shape

$$Id_{Univ} \circ pelint \circ Id_{(Pgm,Univ)} \qquad (4.8)$$

where the two Id_τ functions are function that take a value of type τ apart and builds it again from its primitive components. For example, the PEL definition of Id_{Univ} is

$$\mathsf{f}\, x_1 = \mathsf{case}\ x_1\ \mathsf{of} \left\{ \begin{array}{l} \mathsf{L}\, x_2 \mapsto \mathsf{L}\, x_2 \\ \mathsf{R}\, x_2 \mapsto \mathsf{case}\ x_2\ \mathsf{of} \left\{ \begin{array}{l} \mathsf{L}\, x_3 \mapsto \mathsf{R\,L}\,(\mathsf{f\ fst}\, x_3, \mathsf{f\ snd}\, x_3) \\ \mathsf{R}\, x_3 \mapsto \mathsf{case}\ x_3\, \mathsf{of} \left\{ \begin{array}{l} \mathsf{L}\, x_4 \longmapsto \mathsf{R\,R\,L\,f}\ x_4 \\ \mathsf{R}\, x_4 \longmapsto \mathsf{R\,R\,R\,f}\ x_4 \end{array} \right\} \end{array} \right\} \end{array} \right\}$$

The two Id_τs in (4.8) are the residues of $decode_{Univ}$ and $encode_{(Pgm,Univ)}$. I think that despite their presence it is fair to call MiXIMUM optimal—after all the pieces it can't remove completely are not part of a layer of interpretation at all.

Still, it ought to be possible to write a general automatic transformation that identify and eliminate functions (or expressions) that behave like Id_τ functions. I think this is basically the same problem as the one mentioned in Section 3.6.

The *pelint* part of (4.8) is not completely equivalent to the original *pelint* text but slightly better. If the original *pelint* text is run through the let reduction phase of MiXIMUM, it becomes α-equivalent to the *pelint* from (4.8). The only differences are in the order and naming of the functions, in the names of variables, and in the the main function where calls to the Id_τ functions have been inserted. If the function order is adjusted manually, and MiXIMUM's prettyprinter is used to canonicalize the variable names, the two programs become textually identical (again except for the main function).

4.3 Reproducing the results

The MiXIMUM prototype I have developed is available electronically at

⟨`http://www.diku.dk/~makholm/miximum.tar.gz`⟩

The system can be compiled in a unixish environment where GNU make and Moscow ML 1.44 is installed. Follow the instructions in the `README` file.

Beware that the system is very much a prototype. The user interface is kludgy; the error messages when anything goes wrong are wildly misleading;

the transformations themselves are at least an order of magnitude slower than they need be; and the source code for the system is hard to understand except perhaps for its author. Also, the PEL dialect it works with is sligtly different from the one that occurs in Welinder [1998] in that the subtraction operator is missing. That is because the machine-readable version of the PEL self-interpreter I have available is from an early draft of the thesis, before he decided to add subtraction.

Still the system can be used to specialize *pelint* to itself, and it also contains a framework for doing

$$[\![\mathit{spec}]\!](\mathit{decode}_{\mathsf{int}} \circ \mathit{pelint} \circ (\mathit{Id} \times \mathit{encode}_{\mathsf{int}}), \mathtt{p})$$

thus specializing *pelint* to any program $\mathtt{p}$ that accepts and produces integers (in that case the encoding and decoding functions disappear completely during specialization). You can also try to specialize *pelint* to programs that are not type correct.

Chapter 5

Conclusion

I have developed and described an optimal (or at least very close to optimal) program specializer for a strongly typed language. The specializer is simpler and easier to understand than previous proposals for how to create optimal specializers for typed languages. Unlike the type specializer proposed by Hughes it never rejects static input as long as the static input is syntactically correct.

It still remains to be seen whether the techniques described here can be extended to more advanced language features such as first-class functions, exceptions, or imperative primitives. Though not described in this paper, the MiXIMUM prototype already supports n-ary sums and products (by viewing them as syntactic sugar in the parser; conversely, the prettyprinter can be set to reintroduce n-ary sums and products by collapsing the binary ones in the output program. That way arbitrary types can be generated for the residual program).

Another possible (long-term) goal would be to produce a self-applicable MiXIMUM, but it is not clear how much practical value that would have, given that most of the complexity in the current system lies in the post-processes.

This project has ended up looking very differently from what I imagined when I started it. One direct conclusion I can make from that is that is is not advisable to find out halfway through a project that the problem you're solving is not the relevant problem—at least not in projects with deadlines.

References

Bjørner, D., Ershov, A. P., and Jones, N. D. (eds) [1987]. *Partial Evaluation and Mixed Computation (IFIP TC2 Workshop, Gammel Avernæs, Denmark)*. North-Holland, Amsterdam, The Netherlands, ISBN 0-444-70491-4.

Bjørner, D., Ershov, A. P., and Jones, N. D. (eds) [1988]. *Partial Evaluation and Mixed Computation (IFIP TC2 Workshop, October 1987, Gammel Avernæs, Denmark)*, special issue of *New Generation Computing*, 6(2–3).

Danvy, O. [1996]. Type-directed partial evaluation. In *Principles of Programming Languages (23rd ACM SIGPLAN-SIGACT Symposium, POPL '96, St. Petersburg Beach, FL, USA)*, pages 242–257. ACM Press, New York, NY, USA, ISBN 0-89791-769-3.
⟨`ftp://ftp.daimi.au.dk/pub/empl/danvy/Papers/danvy-popl96.ps.gz`⟩.

Futamura, Y. [1971]. Partial evaluation of computation process – an approach to a compiler-compiler. *Systems, Computers, Controls*, 2(5):45–50. Reprinted as Futamura [1999].

Futamura, Y. [1999]. Partial evaluation of computation process—an approach to a compiler-compiler. *Higher-Order and Symbolic Computation*, 14(4):381–391. Reprint of Futamura [1971].

Glenstrup, A. J. [1999]. Terminator II: Stopping partial evaluation of fully recursive programs. Master's thesis, Department of Computer Science, University of Copenhagen. DIKU-TR-99/8. ⟨`http://www.diku.dk/~panic/TerminatorII/`⟩.

Glenstrup, A. J. and Jones, N. D. [1996]. BTA algorithms to ensure termination of off-line partial evaluation. In Bjørner, D., Broy, M., and Pottosin, I. V. (eds), *Perspectives of System Informatics (2nd International Andrei Ershov Memorial Conference, PSI '96, Novosibirsk, Russia)*, volume 1181 of *Lecture Notes in Computer Science*, pages 273–284. Springer-Verlag, Heidelberg, Germany, ISBN 3-540-62064-8. ⟨`ftp://ftp.diku.dk/diku/semantics/papers/D-274.ps.gz`⟩.

Hughes, J. [1996a]. Type specialization for the λ-calculus; or, A new paradigm for partial evaluation based on type inference. In Danvy, O., Glück, R., and Thiemann, P. (eds), *Partial Evaluation (International Seminar, Dagstuhl Castle, Germany)*, volume 1110 of *Lecture Notes in Computer Science*, pages 183–251. Springer-Verlag, Heidelberg, Germany, ISBN 3-540-61580-6.
⟨`http://www.cs.chalmers.se/~rjmh/Papers/typed-pe.ps`⟩.

Hughes, J. [1996b]. An introduction to program specialization by type inference. In *Glasgow Workshop on Functional Programming*. Glasgow University. ⟨`http://www.cs.chalmers.se/~rjmh/Papers/glasgow-96.dvi`⟩.

Jones, N. D. et al. [1987]. Challenging problems in partial evaluation and mixed computation. In Bjørner et al. [1987], pages 1–14. Also pages 291–303 of Bjørner et al. [1988].

Jones, N. D., Gomard, C. K., and Sestoft, P. [1993]. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, Englewood Cliff, NJ, USA, ISBN 0-13-020249-5. ⟨`http://www.dina.kvl.dk/~sestoft/pebook/pebook.html`⟩.

Makholm, H. [2000]. Region-based memory management in Prolog. Master's thesis, Department of Computer Science, University of Copenhagen. ⟨`http://www.diku.dk/~makholm/speciale.ps`⟩.

Mogensen, T. Æ. [1987]. Partially static structures in a self-applicable partial evaluator. In Bjørner et al. [1987], pages 325–347.

Welinder, M. [1998]. *Partial Evalutation and Correctness*. PhD thesis, Department of Computer Science, University of Copenhagen. DIKU-TR-98/13.