Chapter 0

Program Specialization using Input Distribution Spectra

Egad, I think the interpreter is the hardest to be understood of the two!

— RICHARD BRINSLEY SHERIDAN (1751–1816)
The Critic, Act 1, Scene 2

Chapter Summary

This chapter presents background and technical detail for the first of three original contributions of this dissertation, namely:

> *Spectral Specialization* - source-to-source program specialization with respect to *context spectra*, constituting dynamically adaptive, profile-driven, polyvariant on-line partial evaluation grounded in statistical inference.

0.0 Preview

This chapter begins by establishing the formal foundations on which DESCARTES was built. It outlines the formal methods and historical developments that constitute *program specialization*, reviewing the traditional notational conventions that permeate the field. In passing, it also offers copious references for those new to the topic and for those inclined toward rigorous review of things fundamental and their provenance.

Having established this technical context, DESCARTES' so-called *spectral specialization* can then be related to existing work and the details of its novel aspects can be highlighted.

Specifically, *spectral specialization* in DESCARTES is a relatively straightforward generalization of Erik Ruf's techniques for *polyvariant on-line partial evaluation* [Ruf 93] to use *dynamic abstract contexts* (as embedded within what I term *context spectra*). These *context spectra* are described in detail in a subsequent chapter (Chapter **??**, p. **??**). They are, in essence, a generalization of the idea of "types" to "statistically inferred run-time distributions of sets of weighted disjoint (non-intersecting) structural datatypes". Program specialization with respect to these *context spectra* is therefore a small matter of presenting the standard traditional specialization rules in this setting, being mindful of the total correctness[0] of the underlying partial evaluation process.

Second, the information encapsulated within *context spectra* subsumes what is normally evinced either: a) by incorporating a fixed-point induction engine into the on-line partial evaluator [Ashley & Consel 94], or b) by performing a pre-pass program annotation phase (called *binding time analysis* [Christensen, Glück & Laursen 2000]) for the off-line case. This means that the termination conditions for the partial evaluator have a pleasingly simple, elegant flavor, avoiding the obligatory concomitant esoteric machinery of traditional approaches.

Moreover, consideration of the efficacy of the resulting specialized code, as well as the time and space efficiency of the specializer itself, leads to a natural generalization of Urs Hölzle's *polyvariant in-line caches* (PICs) [Hölzle, Chambers & Ungar 91] [Hölzle 94]. That is, whereas PICs cache fixed-numbers of simple type-based dispatch operations using random replacement, DESCARTES employs an analogous mechanism to maintain variable-sized dispatch "trampolines" (§ **??**) to guide run-time polyvariant specialized code invocation while ensuring code safety. This has the effect of rendering otherwise-unsafe specializations safe.

Finally, a few novel concepts and opportunities present themselves as a consequence of the probabilistic weights (or, relative proportions) associated with the sub-nodes of the *dynamic abstract contexts* of *context spectra* as well as the inferential empirical nature of of their discovery.

The chapter ends with a short reflection on the importance and relevance of this first original contribution. It also briefly reiterates and reinforces the importance of formal methods (*viz., denotational semantics*) in the development of this approach.

0.1 Deep Background: Formal, Historical and Notational Context

Program specialization in DESCARTES is achieved by means of *partial evaluation*, which is an instance of *denotational abstract interpretation* grounded in *Kleene's S_n^m Theorem.*

0.1.0 Kleene's S_n^m Theorem

Historically, *Kleene's S_n^m Theorem* [Kleene 50, § 65, Theorem XIII] formally established that a procedure of $m + n$ formal parameters can be refactored as two procedures, the first of which is a procedure of m parameters whose body is, in turn, a (second) procedure of the remaining n parameters. When the outer procedure is applied to m arguments and the resulting inner procedure (closed over these m initial arguments) is then applied to n arguments, the result corresponds exactly to applying the original unfactored procedure to all $m + n$ arguments simultaneously. That is:
Therefore, if one defines, say,

$$\psi_{y_1,\ldots,y_m} \triangleq \phi'(y_1, \ldots, y_m)$$

[0] *I.e., congruence* and *termination*, defined below (p. 70).

$$\boxed{\phi(y_1,\ldots,y_m,x_1,\ldots,x_n) \equiv (\phi'(y_1,\ldots,y_m))(x_1,\ldots,x_n)}$$

Figure 0-0: The equational essence of *Kleene's S_n^m Theorem*

then one can replace all instances of $\phi(y_1,\ldots,y_m,x_1,\ldots,x_n)$ with $\psi_{y_1,\ldots,y_m}(x_1,\ldots,x_n)$ throughout. Note that this replaces the $(n+m)$-argument invocations of ϕ with m-argument invocations of $\psi_{y_1,\ldots,y_m}$, thereby effectively reducing by m the *arity* at each call site. This is traditionally called *arity reduction.*

Moreover, this S_n^m refactorization transformation can be defined as a *general recursive function* ([Gödel 34,65] [Kleene 50, Chapter XI, § 55]), so it therefore corresponds to an *effectively calculable procedure* (courtesy of the seminal *Church-Turing (Post) Thesis* [Church 36,65] [Turing 36–37,65] [Post 36,65]).

This last point (and the main consequence of Kleene's theorem for our purposes) formally established the feasibility of implementing *partial evaluation* as a meta-program.[1] Specifically, although *Kleene's S_n^m Theorem* merely established the definability of such computable refactorization procedures without committing to why one might wish to do so, *partial evaluation* extends this to program transformers with the goal of program optimization via source-to-source transformation (like, for example, the optimization transforms of [Burstall & Darlington 77]).

0.1.1 Partial Evaluation as S_n^m Refactorization

Informally, *partial evaluation* [Jones, Gomard & Sestoft 93] [Jones 96] [Bender 2006] takes a source-level program and an indication of which arguments and data are static or dynamic, then produces a *residual* specialized program that has the static internals compile-time reduced (*folded*). The resulting residual program, when given the withheld dynamic data, produces the same result as the original program would have if given the static and dynamic data together. This definition is traditionally represented equationally as in Figure 0-1.

$$\boxed{\mathcal{PE}[\![f, \text{static\_args}]\!](\text{dynamic\_args}) \equiv f(\text{static\_args}, \text{dynamic\_args})}$$

Figure 0-1: Traditional equational representation of *partial evaluation*

What distinguishes $\mathcal{PE}$ as a special class of S_n^m transformer is that, in addition to refactoring the formal parameter list, it also employs *denotational abstract interpretation* to optimize the original procedure's body with respect to the fixed (static) parameters. Equationally, one might express this as:

...where, in a slight abuse of traditional notation, $[\{s_1,\ldots,s_m\} \mapsto^* \text{static\_args}]$ denotes a

[1]Historical Note: The observation that *partial evaluation* is an instance of *Kleene's S_n^m Theorem* was first made by Ershov [Ershov 78], according to Futamura [Futamura & Nogi 88]. The first instance of its realization appears to have been substantially earlier, however, as embodied by [Lombardi & Raphael 64/66] [Lombardi 67], according to [Consel & Danvy 93].

$$\mathcal{PE}[\![\texttt{(lambda (}s_1, \ldots, s_m, d_1, \ldots, d_n\texttt{)}\ e\texttt{)}, \text{static\_args}]\!]$$
$$\Longrightarrow$$
$$\texttt{(lambda (}d_1, \ldots, d_n\texttt{)}\ \mathcal{PE}[\![e, [\{s_1, \ldots, s_m\} \mapsto^* \text{static\_args}]]\!]\texttt{)}$$

Figure 0-2: *Partial evaluation* is S_n^m plus *denotational abstract interpretation*

partial evaluation "environment" in which the formal parameters $s_1, \ldots, s_m$ are associated with the (set of) values denoted by static_args, each in turn.

The salient point here is that a partial evaluator, $\mathcal{PE}$, transforms an $(n+m)$-argument procedure into an m-argument procedure whose body is recursively descended and thus partially evaluated in an environment where the static formal parameters are bound to the (known) static arguments. If/when, say, a call to a primitive procedure is encountered in which all its arguments are static, the procedure can be reduced to a concrete static result in the resulting residual procedure's body.

In this sense, *partial evaluation* can be thought of both as being an instance of partial *compilation* of a program (given a subset of its inputs) as well as being just a highly aggressive form of constant folding.

0.1.2 Partial Evaluation via Denotational Abstract Interpretation

Moreover, as we are about to see, *partial evaluation* is mathematically grounded in the formal logic of *denotational abstract interpretation* [Cousot & Cousot 77], lending its soundness and correctness to formal verification (*e.g.,* using *denotational semantics* to formally prove that the program transformations preserve the semantics of the original source programs [Mulmuley 86]).

To wit, *abstract interpretation* is a modest generalization of normal program interpretation. Specifically, a traditional *language interpreter* takes as input a syntactic expression and a *valuation environment*, ρ, mapping *identifier*s to *concrete values* in the *concrete value domain.* It then produces a *concrete value* corresponding to the concrete execution of the source program in the specified *valuation environment* using a *concrete valuation function* ($\mathcal{E}$).

By contrast, a traditional *abstract interpreter* takes as input a syntactic expression and an *abstraction environment*, $\rho^\sharp$, mapping *identifier*s to *abstract values* in the *abstract value domain.* It then produces an *abstract value* corresponding to the abstract execution of the source program in the specified *abstraction environment* using an *abstract valuation function* ($\mathcal{E}^\sharp$).

Language Interpreter (Standard Semantics)

For example (Figure 0-3), a traditional *language interpreter* [Abelson & Sussmans 96] for integer arithmetic might behave roughly as follows, based on standard semantics.

$$
\begin{array}{lll}
 & \mathcal{E}[\![\texttt{((lambda (a b) (+ a b)) 1 3)}]\!]\,\rho & \text{[call]} \\
\Longrightarrow & & \\
 & \big(\mathcal{E}[\![\texttt{(lambda (a b) (+ a b))}]\!]\,\rho & [\lambda] \\
 & \mathcal{E}[\![1]\!]\,\rho & [\Bbbk] \\
 & \mathcal{E}[\![3]\!]\,\rho\big) & [\Bbbk] \\
\Longrightarrow^{3} & & \\
 & \big(\lambda\, x, y\ .\ (\mathcal{E}[\![\texttt{(+ a b)}]\!]\,\rho[\texttt{a} \mapsto x][\texttt{b} \mapsto y])\ 1\ 3\big) & \text{[apply]} \\
\hookrightarrow_{\beta} & & \\
 & \mathcal{E}[\![\texttt{(+ a b)}]\!]\,\rho[\texttt{a} \mapsto 1][\texttt{b} \mapsto 3] & \text{[call]} \\
\Longrightarrow & & \\
 & \big(\mathsf{add}\ 1\ 3\big) & \text{[primitive]} \\
\hookrightarrow_{\delta} & & \\
 & 1 + 3 & [\textit{deus ex machina}] \\
\rightsquigarrow_{\varkappa} & & \\
 & [\![4]\!]^{-1} & \text{[return]} \\
\texttt{4}\ \hookleftarrow_{\varpi} & &
\end{array}
$$

Figure 0-3: A simple *language interpreter* (*standard semantics*) demonstration

Here, following long-standing convention [Stoy 77], the double bracket *quasi-quotation* operator, $[\![\ldots]\!]$, maps sequences of `denotation` characters (*i.e., input syntax*) into corresponding *expressions* in the (implicit) underlying syntax of source programs.[2]

This starts with the source program `((lambda (a b) (+ a b)) 1 3)` and an initial *valuation environment*, ρ, (presumably mapping `+` to the primitive addition procedure). Recognizing this expression as a procedure call form, the operator and operand expressions are first evaluated then the result is applied as per the β-reduction rule of the standard λ-calculus [Church 51]. This results in a recursive call to the *concrete valuation function* ($\mathcal{E}$) on the body of the `lambda` expression in the *valuation environment* ρ extended to bind the formal parameter `a` to 1 and `b` to 3. This, in turn, is recognized as another procedure call form, so the operator and operands are first evaluated then the result is applied, in this case using the standard δ-reduction rule of the λ-calculus since the operator is a base primitive procedure. Finally, this corresponds to the arithmetic sum of 1 and 3, which is 4. The result is therefore the numeral (denotation) `4`—not the arithmetic number 4 from the *concrete value domain*.

This is also an (informal) example of *denotational semantics* [Stoy 77] [Gordon 79] [Schmidt 86], which maps program source text ("denotations") to their underlying mathematical meanings ("semantics") in an underlying recursive domain model of values [Scott 76]. As illustrated

[2]Historical Note: This traditional so-called "*quasi-quote*" double bracket notation derives from Quine's "canonical notation" [Quine 60]. Whereas Quine used single brackets or, at times, $\ulcorner\ldots\urcorner$, Stoy both unified and generalized this notation slightly [Stoy 77] to admit meta-syntactic variables within source syntax (although no such meta-syntactic variables appear in the present example). The resulting "$[\![\ldots]\!]$" notation is now standard, if not ubiquitous, in the field of formal semantics.

above, this reduction of *syntactic expressions* to elements of the *concrete value domain* is rooted in Church's λ-calculus model of computation [Church 51]. This wedding of recursive domain models with the λ-calculus is the quintessence of *denotational semantics* [Scott 70] [Stoy 77].

In the particular case of a *language interpreter*, the *denotational semantics* guides the mapping of *syntactic expressions* into the domain of *concrete values*.

Abstract Interpreter (Non-Standard Semantics)

By contrast, a comparable traditional *abstract interpreter* [Nielson 86] corresponding to the same source program might be constructed from the preceding standard (concrete) semantics as an approximating (abstract) semantics, something like the following.

First, note that in the preceding *denotational semantics* for our traditional *language interpreter*, I implicitly defined recursive *concrete value domains* [Scott 76] comprised of integers and (total) functions from values to values (where "+" below denotes *disjoint union*):

$$\begin{array}{lcll} v \in \mathcal{V} & = & \mathbb{Z} + \mathcal{F} & \text{[Values]} \\ f \in \mathcal{F} & = & (\mathcal{V}^* \to \mathcal{V}) & \text{[Functions]} \\ z \in \mathbb{Z} & & & \text{[Integers]} \end{array}$$

I also assumed an initial *valuation environment*, $\rho \in \mathbf{E}$, to map the names of the primitive procedures in the input syntax of source programs (*viz.*, *identifiers*, $\iota \in \mathbf{I}$) to their corresponding underlying integer arithmetic primitive *base operators*, $\phi \in \Phi$. I then presented a fragmentary example of the (implied) standard *concrete valuation function*, $\mathcal{E}$, which maps source text into their concrete integer counterparts in this *concrete value domain*, $\mathcal{V}$, using this initial *valuation environment*.

Now, for the present traditional *abstract interpreter*, we extend this standard semantics with an *abstraction function* to map *concrete values* to their abstract (approximate) counterparts in an *abstract value domain*. We also must define *abstract base operators* to function on these abstract inputs appropriately (*i.e.*, as semantically *faithful* and *congruent* mirrors of their concrete primitive procedure counterparts).[3] Appropriate *abstract base operators* are assumed.

First, of course, we must choose an abstraction.

Following tradition, I choose the integer parity function as our first example *abstraction function*.[4] This projects elements of the integers, $z \in \mathbb{Z}$, onto appropriate elements of the set $\{\text{EVEN}, \text{ODD}\}$, odd integers to ODD and even to EVEN. It also must project each *concrete base operator* onto some appropriate corresponding *abstract base operator*.

Following traditional notation [Mycroft 81], this chosen *abstraction function*, denoted "α", maps elements of the *concrete value domain*, $v \in \mathcal{V}$, to corresponding elements of the *abstract*

[3] The terms *faithful* and *congruent* in this context have their usual formal meanings of, informally, being "reasonable" and "well behaved" [Jones, Gomard & Sestoft 93]. Those details are not important for the present informal discussion. (See page 70 for formal definitions.)

[4] An alternative traditional first example might be the signum function, which maps zero to 0, positive integers to +1, and negative integers to −1, but that tends to generate confusion when one sees, for example, that the *abstract base operator* for addition is defined as per $1 + 1 = 1$, and similar such apparent nonsense.

value domain, $v^\sharp \in \mathcal{V}^\sharp$. Formally:

$$\begin{array}{rcl} \alpha(v) & \rightarrow & v^\sharp \\ \alpha(f) & \rightarrow & f^\sharp \\ \alpha(z) & \rightarrow & z^\sharp \end{array}$$

For our specific choice of the integer parity procedure as the core *abstraction function*, α, this naturally induces the following *abstract value domain*:

$$\begin{array}{rcl} v^\sharp \in \mathcal{V}^\sharp & = & \mathcal{Z}^\sharp \ + \ \mathcal{F}^\sharp \\ f^\sharp \in \mathcal{F}^\sharp & = & (\mathcal{V}^{\sharp^*} \rightarrow \mathcal{V}^\sharp) \\ z^\sharp \in \mathcal{Z}^\sharp & = & \text{ODD} \ + \ \text{EVEN} \end{array}$$

The *abstract base operators*, $\phi^\sharp \in \Phi^\sharp$, are then defined as appropriately well-behaved[5] abstract functions from $\mathcal{V}^{\sharp^*}$ to $\mathcal{V}^\sharp$. For instance, $+^\sharp$ behaves as:[6]

$$\begin{array}{rcccl} \text{EVEN} & +^\sharp & \text{EVEN} & = & \text{EVEN} \\ \text{EVEN} & +^\sharp & \text{ODD} & = & \text{ODD} \\ \text{ODD} & +^\sharp & \text{EVEN} & = & \text{ODD} \\ \text{ODD} & +^\sharp & \text{ODD} & = & \text{EVEN} \end{array}$$

Figure 0-4: Sample integer parity abstraction of the addition base operator

We now presume an initial *abstraction environment*, $\rho^\sharp \in \mathbf{E}^\sharp$, to map the names of the primitive procedures in the input syntax of source programs (*viz., identifiers*, $\iota \in \mathbf{I}$) to their corresponding underlying integer arithmetic function primitive *abstract base operators*, $\phi^\sharp \in \Phi^\sharp$.

[5] That is, *faithful* and *congruent*, defined below (p. 70).

[6] It may amuse some to note that, if we encode EVEN as 0 and ODD as 1, then $+^\sharp$ can be defined as straightforward addition modulo 2.

$$
\begin{array}{lll}
 & \mathcal{E}^\sharp[\![\texttt{((lambda (a b) (+ a b)) 1 3)}]\!]\,\rho^\sharp & \text{[call]} \\
\Longrightarrow & & \\
 & \big(\mathcal{E}^\sharp[\![\texttt{(lambda (a b) (+ a b))}]\!]\,\rho^\sharp & [\lambda] \\
 & \mathcal{E}^\sharp[\![\texttt{1}]\!]\,\rho^\sharp & [\Bbbk] \\
 & \mathcal{E}^\sharp[\![\texttt{3}]\!]\,\rho^\sharp\big) & [\Bbbk] \\
\Longrightarrow^3 & & \\
 & \big(\lambda\, x, y\ .\ (\mathcal{E}^\sharp[\![\texttt{(+ a b)}]\!]\,\rho^\sharp[\texttt{a} \mapsto x][\texttt{b} \mapsto y])\ \alpha(1)\ \alpha(3)\big) & \text{[apply]} \\
\hookrightarrow_\beta & & \\
 & \mathcal{E}^\sharp[\![\texttt{(+ a b)}]\!]\,\rho^\sharp[\texttt{a} \mapsto \textsc{odd}][\texttt{b} \mapsto \textsc{odd}] & \text{[call]} \\
\Longrightarrow & & \\
 & \big(\alpha(\mathtt{add})\ \textsc{odd}\ \textsc{odd}\big) & \text{[primitive]} \\
\hookrightarrow_\delta & & \\
 & \textsc{odd}\ +^\sharp\ \textsc{odd} & [\textit{deus ex machina}] \\
\rightsquigarrow_\varkappa & & \\
 & [\![\textsc{even}]\!]^{-1} & \text{[return]} \\
\texttt{even}\ \hookleftarrow_\varpi & &
\end{array}
$$

Figure 0-5: A simple *abstract interpreter* (*non-standard semantics*) demonstration

Finally, we're now equipped to demonstrate the *abstract valuation function*, $\mathcal{E}^\sharp$, as follows (Figure 0-5). The result in this case is the syntactic symbol (denotation) `even`— not the domain element EVEN from the *abstract value domain*.

Aside from the liberal sprinkling about of $\sharp$'s, note the *en passant* appearance of the *abstraction function*, α, when binding the parameters to their (abstractly) evaluated arguments in the *abstraction environment* of the β-reduction step. Contrast this with the like-labeled step in the standard semantics of the preceding traditional *language interpreter*, where it tacitly converted the syntactic numeral inputs (`1` and `2`) to their corresponding underlying arithmetic *concrete values* (1 and 2) without fanfare. This exposes a minor subtlety where I've suppressed an intermediate step that a more rigorous demonstration might have included, albeit at the risk of obscuring the earlier example.

That intermediate step is the usual $\Bbbk$ rule, which projects literal constant input expressions into the domain of *concrete values*, typically by invoking a projection operator named $\mathcal{K}$. For the standard semantics of language interpreters, this is an obvious and straightforward mapping (albeit machine dependent in concrete implementations), so it is typically deemed "uninteresting" and therefore not shown. For the non-standard semantics of abstract interpreters, however, the *en passant* invocation of α is essential. It was better to have delayed it until now in order to highlight its ultimate purpose, if only for dramatic effect.

This illustrates how, in an *abstract interpreter*, the *denotational semantics* guides the mapping of *syntactic expressions* into the domain of *abstract values*.

Partial Evaluation as Abstract Interpretation

Extending *abstract interpretation* one step further, *partial evaluation* aggressively reduces all static subexpressions of an expression to their underlying *concrete values*. Moreover, as we shall see shortly, a *partial evaluator* maps expressions to expressions, not expressions to values like the *abstract interpreter* shown above. Nonetheless, *partial evaluation is* a generalized instance of *abstract interpretation*, albeit with non-traditional domain equations [Jones 97]. That is, the mechanism of *abstract interpretation* is generalized with respect to both its *abstraction environment* and its *abstract value domain.*

Specifically, whereas the traditional *abstract interpretation* rule for `lambda` forms demonstrated above delayed the further (abstract) interpretation of the `lambda` body until it is invoked:

$$\begin{array}{ll} & \mathcal{E}^\sharp[\![\texttt{(lambda (a b) (+ a b))}]\!]\,\rho^\sharp \qquad\qquad [\lambda] \\ \Longrightarrow & \\ & \lambda\, x, y\ .\ (\mathcal{E}^\sharp[\![\texttt{(+ a b)}]\!]\,\rho^\sharp[\texttt{a} \mapsto x][\texttt{b} \mapsto y]) \end{array}$$

Figure 0-6: The `lambda` reduction rule for an *abstract interpreter*

...a *partial evaluation* rule for `lambda` would behave more like:

$$\begin{array}{ll} & \mathcal{PE}[\![\texttt{(lambda (a b) (+ a b))}]\!]\,\tau \qquad\qquad [\lambda] \\ \Longrightarrow & \\ & \text{Let}\ \ e_\tau = \mathcal{PE}[\![\texttt{(+ a b)}]\!]\,\tau \\ & \ \ \text{in} \\ & \qquad [\![\lambda\, a, b\ .\ e_\tau]\!]^{-1} \end{array}$$

Figure 0-7: The `lambda` reduction rule for *partial evaluation*

There are two main points to note here.

First, the normal *valuation environment*, ρ, of the standard semantics has been replaced by a static/dynamic determination environment, τ, which maps *identifiers* to tuples of the form: $< v \times b >$, for value $v \in \mathcal{V}$ and static/dynamic *binding indicator* $b \in \mathcal{B}$ where:

$$b \in \mathcal{B} = \{\text{STATIC}, \text{DYNAMIC}\}$$

Just how these *binding indicators* get set is unimportant for now. Imagine a global program analysis oracle to generate these appropriately, for example. I'll come back to this later.

Second, the "inverse" *quasi-quotation* operator, $[\![\ldots]\!]^{-1}$ was used to suggest that a concrete λ-value is generated and then immediately mapped back into a *syntactic expression*. Although well intensioned [*sic*],[7] this is somewhat of an abuse of notation. A more standard notation would be:

$$[\![\texttt{(lambda (a b)}\ e_\tau\texttt{)}]\!] \uparrow \mathcal{S}$$

...where the "$\uparrow \mathcal{S}$" decoration indicates projection of the resulting quasi-quoted expression back up into the *syntactic expression* domain of program source text. Without that bit of decoration, the result would be an expression (an internal machine representation, if you prefer) rather than syntax (program source code text, if you like).

To my taste, the slightly abusive "$[\![\ldots]\!]^{-1}$" notation is less distracting, frankly, when comparing the traditional *abstract interpretation* rule with the comparable *partial evaluation* rule. Henceforward, however, I'll use the projected quasi-quote form instead. That tends to correspond more closely to the source code of the *partial evaluator* itself, which carries the additional advantage of being far less onerous to typeset in the running text.

It should also be noted in passing that some authors use a distinguished `lift` operator for the same purpose [Jones, Gomard & Sestoft 93]. Since this is a meta-operator used only to project expressions into denotations, I prefer the "$\uparrow \mathcal{S}$" decoration instead, as it more clearly suggests a domain projection rather than an expression computation *per se*.

Partial Evaluation via Denotational Abstract Interpretation: Summary

To summarize this section, then, *partial evaluation* is essentially just a specific instance of the broader notion of *denotational abstract interpretation*. In the particular instance of *program specialization*, the objective is to *partial evaluate* program source code with an eye toward reducing constituent subexpressions to their ultimate run-time values wherever possible and safe. This task is enabled by viewing programs as having a semantics in the abstract domain of known/static values *as well as* unknown/dynamic values. The goal, then, is to use this framework to reduce the source code where possible with respect to the static/known information, wrapping these fully reduced static components ("residues") in appropriate dynamic/unknown contextual code fragments in order to generate "residualized" specialized code that will behave exactly as the original code would have, only with the fully and partially static subcomponents somewhat optimized through code specialization.

The next section illustrates core variants on this theme with a few short examples. The one following that then briefly addresses the off-line -v- on-line *partial evaluation* debate. Afterward, following a brief section to narrow the language domain a bit, the *partial evaluation* rules for DESCARTES are finally presented.

0.1.3 Partial Evaluation Variants: A Brief Illustration

Equipped now with the basic underpinnings of *partial evaluation*, consider the following sample program:

[7] Pun and/or *double entendre* intended.

```
-----------------------------------------------------------------------------
 (define       square   (lambda (n)   (* n n)))
 (define sum-of-squares  (lambda (a b) (+ (square a) (square b))))

 (define hypotenuse-3-4  (lambda () (sqrt (sum-of-squares 3 4))))
-----------------------------------------------------------------------------
```

…a *partial evaluation* of a given `lambda` form would first scan the entire program file to determine if any of its arguments are always known (static) values, generating equivalent `lambda` forms with all static formal parameters removed from the formal parameter list and replaced in the `lambda`'s body by their static associates. In so doing, a *partial evaluator*, in effect, transforms expressions to (equivalent, presumably simpler) expressions rather than reducing expressions to values.

Under strict monovariant partial evaluation, for instance, $\texttt{sum-of-squares}_{\{3,4\}}$ is generated but not $\texttt{square}_3$ nor $\texttt{square}_4$ because `sum-of-squares` has a single call site (hence, monovariant) while `square` has two incongruent call sites. The resulting system would look something like:

```
-----------------------------------------------------------------------------
 (define       square   (lambda (n)   (* n n)))
 (define sum-of-squares  (lambda (a b) (+ (square a) (square b))))

 (define sum-of-squares_3x4 (lambda () (+ (square 3) (square 4))))

 (define hypotenuse-3-4  (lambda () (sqrt (sum-of-squares_3x4))))
-----------------------------------------------------------------------------
```

This is, of course, unsatisfying.

As an alternative, a strict monovariant partial evaluation could decide instead to "*dynamize*" the two incongruent calls to `square` by, for example, specializing on the most specific generalization (least upper bound) of the distinct call sites. To wit:

```
-----------------------------------------------------------------------------
 (define       square   (lambda (n)   (* n n)))
 (define sum-of-squares  (lambda (a b) (+ (square a) (square b))))

 (define       square_int (lambda (n) (int:* n n)))
 (define sum-of-squares_3x4 (lambda () (+ (square_int 3) (square_int 4))))

 (define hypotenuse-3-4  (lambda () (sqrt (sum-of-squares_3x4))))
-----------------------------------------------------------------------------
```

…where `int:*` is the (non-generic) integer-only multiply primitive.

Thus, when doing specialization by types [Haraldsson 78] [Weise & Ruf 90′] [Danvy 96] [Danvy 96p] [Danvy 98] rather than just specialization by concrete values, even in the monovariant case the partial evaluator could generate $\texttt{square}_{\texttt{INTEGER}}$ to narrow the generic arithmetic to the fixed input domain consistent with all call sites.[8]

[8] What I dub "call sites" are called "program points" by [Jones, Gomard & Sestoft 93] (who derive their nomenclature from flow chart programs). These refer merely to distinct instances of invocation of a syntactically

Alternatively, generalizing by *disjoint union* rather than by *least upper bound* in the type domain, a type-driven monovariant partial evaluation could produce the following:

```
-----------------------------------------------------------------------------
 (define       square   (lambda (n)   (* n n)))
 (define sum-of-squares  (lambda (a b) (+ (square a) (square b))))

 (define       square_3v4 (lambda (n) (case n
                                        ((3)  9)
                                        ((4) 16)
                                        (else (error "Not 3 or 4?!")))))

 (define sum-of-squares_3x4 (lambda () (+ (square_3v4 3) (square_3v4 4))))

 (define hypotenuse-3-4  (lambda () (sqrt (sum-of-squares_3x4))))
-----------------------------------------------------------------------------
```

This capitalizes on the base case of allowing primitive procedures (like `+`) to be fully reduced when all arguments are static/known. This strategy, however, might be maligned as being merely a poorly-disguised "localized" approximation to polyvariant specialization.

Going one step further, therefore, under true polyvariant partial evaluation, the two variant `square`s *would* be generated separately, as per:

```
-----------------------------------------------------------------------------
 (define       square   (lambda (n)   (* n n)))
 (define sum-of-squares  (lambda (a b) (+ (square a) (square b))))

 (define       square_3    (lambda ()  9)) ;;; i.e., (* 3 3)
 (define       square_4    (lambda () 16)) ;;; i.e., (* 4 4)
 (define sum-of-squares_3x4 (lambda () 25)) ;;; i.e., (+ (square_3)(square_4))

 (define hypotenuse-3-4  (lambda () 5))     ;;; i.e., (sqrt (sum-of-squares_3x4))
-----------------------------------------------------------------------------
```

Note here that a knock-on effect has permitted the generation of a cavalcade of constant functions. These intermediate specializations, like `square_3` and even `sum-of-squares_3x4`, needn't be retained post-specialization, but they are shown here for clarity (and because they typically are retained, at least temporarily, to afford opportunities for avoiding redundant regeneration later [Ruf & Weise 92r]).

The liability in all this, in both the monovariant and polyvariant cases, however is the dependence on whole-program analysis, which presumes that the whole program is available for scrutiny. In a dynamic development environment, where new code is being written and/or new programs modules are dynamically linked in, while deprecated code is retired (such as when a fresh code patch is downloaded into a live system), this "closed world assumption" breaks down.

Imagine, for example, adding a new definition to the original system:

identifiable operator within an expression, *i.e.,* visually distinguishable places in the text where a given operator is called.

```
----------------------------------------------------------------------------
 (define       square   (lambda (n)   (* n n)))
 (define sum-of-squares  (lambda (a b) (+ (square a) (square b))))

 (define hypotenuse-3-4  (lambda () (sqrt (sum-of-squares 3 4))))

 (define mean-variance   (lambda (mean)
                            (lambda (x y) (sum-of-squares (- x mean)
                                                          (- y mean)))))
 (define norm-variance
        (mean-variance 0))
----------------------------------------------------------------------------
```

A strict value-based monovariant approach would be rendered inert due to the incongruent call sites for `sum-of-squares`: one cannot even assume that `x` and `y` are constrained to be integers, for example.

The polyvariant case is less stymied: it can do as before with the old code but, by exploiting the idempotence of `0` with respect to subtraction, generate new specializations as per:

```
----------------------------------------------------------------------------
 (define mean-variance_0 (lambda ()
                            (lambda (x y) (sum-of-squares x y))))
 (define norm-variance
        (mean-variance_0))
----------------------------------------------------------------------------
```

The plot thickens still further when one considers a re-factorization of the code, as follows:

```
----------------------------------------------------------------------------
 (define       square   (lambda (n)   (* n n)))
 (define sum-of-squares  (lambda (a b) (+ (square a) (square b))))

 (define hypotenuse-3-4  (lambda () (norm-deviation 3 4)))  ;;; <==

 (define mean-variance   (lambda (mean)
                            (lambda (x y) (sum-of-squares (- x mean)
                                                          (- y mean)))))
 (define norm-variance
        (mean-variance 0))

 (define mean-deviation  (lambda (mean)
                            (lambda (x y)
                               (sqrt ((mean-variance mean) x y)))))
 (define norm-deviation
        (mean-deviation 0))
----------------------------------------------------------------------------
```

I leave this for the reader to ponder. Suffice it to say, for now, that a dynamic program development environment poses potentially severe challenges to non-incremental (“closed world”) specialization methodologies. DESCARTES therefore embodies an incremental, dynamically adaptive approach, by contrast.

0.1.4 Partial Evaluation Strategies: On-Line -v- Off-Line

This still begs the question of exactly how the global program analysis is done. Specifically, how exactly does an algorithm determine which formal parameters are static and which dynamic? How are expressions and their subexpressions identified as static or dynamic? How does one guarantee termination of this analysis for recursive function definitions?

All this is addressed by so-called *binding time analysis* [Jones & Schmidt 80] [Mogensen 89] for off-line partial evaluation, where the analysis is done in a pre-pass over the whole program text before partial evaluation proper is initiated. In on-line partial evaluation, this analysis is done as part of the partial evaluation process itself.

Some controversy persists within the PE community as to which approach is to be preferred and for what reasons. It has been argued [Christensen & Glück 2004], however, that the two approaches are equipotent, at least in terms of accuracy, when using *maximally polyvariant BTA* [Christensen, Glück & Laursen 2000]. Nonetheless, [Christensen & Glück 2004] further note that residualization/generalization decisions to ensure termination are naturally flow-sensitive in on-line PE— not so for off-line PE— so on-line PE can use the specialization history to make better generalization ("dynamization") choices, ensuring termination while affording more aggressive specialization. Ultimately, however, the difference may just boil down to a matter of taste. *De gustibus non est disputandum.*

In DESCARTES, on-line partial evaluation is adopted since this tends to simplify the presentation while making comparison to related work (especially, [Ruf 93]) more immediate. It is also more in keeping with the run-time/dynamic/on-the-fly profile-driven approach advocated here.

0.1.5 Deep Background: Section Summary

This section covered a lot of ground. It briefly traced the mathematical foundations of *partial evaluation* from formal logic, through *denotational semantics* to *abstract interpretation*. It then touched briefly on design alternatives, such as monovariant -v- polyvariant approaches and off-line -v- on-line approaches. Along the way, it also briefly touched on the idea of *type-driven partial evaluation* and the notions of *residualization* based on a *most specific generalization* of dynamic variants. We will revisit these issues as they arise later during the detailed discussion of the DESCARTES prototype system.

In the meanwhile, the high-level picture to keep in mind is this: DESCARTES employs a polyvariant, on-line approach to program specialization, continually adapting the collective set of generated specializations based on emergent trends in program use and definition. The remainder of this chapter details this approach to program specialization *per se*. Subsequent chapters, in turn, address the type system on which this is based (*viz., spectral type distributions*), and then the statistical foundations on which that data representation is based, as well as the statistical feedback tuning that makes the whole approach viable.

We proceed now by first narrowing the domain of discourse then forging ahead with the DESCARTES *partial evaluation* rules.

0.2 A Convenient Syntactic Preliminary: Grammatical Retract

This section reduces the source code input language, MIT SCHEME, into a small kernel dialect with fortuitous annotations to support tag-directed dispatch and explicit syntactic disambiguation within the source-to-source program specializer. The resulting dialect is still executable as SCHEME code. It is generated at run time from live (compiled) code through a process dubbed LERKS *normalization*, which re-writes source expressions into `Let`-Extracted *Rufian* KMP SCHEME *Normal Form.*

The impatient reader may prefer to skip ahead to the summary LERKS BNF grammar on page 31 of Section 0.3.0.

Otherwise, the retraction proceeds in several steps.

0. First, the full MIT SCHEME dialect is reduced to a small kernel subdialect of only a handful of syntactic forms using the native run-time `unsyntax` procedure, perhaps more familiar to most as the "pretty printer" (`pp`), its most popular incantation. This constitutes simple *syntactic retraction* in its purest form. The resulting subdialect is commonly referred to as "The Gang of Eight" (or "Nine", depending on how you count).

1. Next, the resulting kernel subdialect is decorated such that all sub-forms are explicitly tagged, including procedure calls and variable references. This is traditional LISP retraction to so-called "KMP form" (or "`CALL/LOOKUP`" form). Self-evaluating constants are also tagged, for universal consistency.

2. Afterward, a "de-nesting" transformation is performed such that all nested sub-forms are explicitly named (*à la* `LET` wrappers). This reversible intermediate transformation allows the specializer to annotate subexpressions during partial evaluation without resorting either to an opaque internal *annotated parse tree* representation or to source-level conversion into pervasive *continuation passing style*— two common internal compiler transformations that, alas, either render the intermediate program representation non-executable (in the former case) or else obfuscate and over-specify the source program (in the latter). This de-nesting transformation has become known as "monadic normal form" or "administrative normal form" or "CPS without continuations", *etc.* (Citations below.) I prefer to call it `let`-*extraction* in this context.

3. Finally, a modest refinement of the coarse-grain `CALL/LOOKUP` tags of step 1 is performed in order to distinguish variables according to binding category: local, global, formal parameter, or primitive. This is a slight elaboration of Erik Ruf's tagging schema. *E.g.,* it renders trivial the *ex post facto* elicitation of the "free variables" of an expression.

Details, definitions and copious scholarly citation of attributions follow.

0.2.0 Syntax Retraction

First, the full SCHEME language includes many superfluous macros which are convenient for programming but a distraction for automated source manipulation. Therefore, it is standard practice to implement only a kernel subdialect that spans the entire space of the full language. Such a minimal kernel subdialect is called a (syntactic) *retract* of the full language grammar, while the function that maps the full language to its *retract* is called a *retraction* [Stoy 77].[9]

This is normally where authors insert the phrase "without loss of generality" to indicate that adding additional forms introduces no new interesting problems. In the case of DESCARTES, since it retrieves the source code from live, running programs in order to specialize them, this is more an argument of "without loss of functionality". Specifically, the aim here is to re-write the original source code associated with any live procedure into a kernel subdialect while avoiding "throwing out the baby with the bath water".

To that end, in MIT SCHEME, we get a fairly complete *retraction* of standard SCHEME directly from using the built-in `unsyntax` procedure. With a few modest adjustments to elide embedded comments and avoid showing internal `DEFINE`s as `#!auxiliary` parameters (neither of which are standard SCHEME anyway) this retraction becomes standard SCHEME.[10]

We can thankfully disregard all MIT SCHEME special form extensions to standard SCHEME— namely, `DEFAULT-OBJECT?`, `UNASSIGNED?`, `DECLARE`, `LOCAL-DECLARE`, `DEFINE-INTEGRABLE`, and such. All of these macros expand into innocuous procedures or equivalent standard SCHEME forms or else they entirely evaporate when compiled.

Moreover, when the input program is standard SCHEME, then so too will be the *retraction* output since no new MIT SCHEME-specific special forms are inserted by the un-syntaxer on compiled code. . . except for the name-space extensions. I gleefully ignore those and imagine we live in a world with one, flat, universal name space. Future work can sort that out.

In short, everything below the dotted waterline in the grammar below can be ignored in the remainder of this dissertation without loss of functionality.

With a fluid re-binding of its internally defined `collect-lambda` to use `collect-named-lambda` so that even anonymous `LAMBDA` expressions are canonicalized into `NAMED-LAMBDA`s (with unique un-interned names), we get the following kernel dialect of MIT SCHEME from `unsyntax`:[11]

---

[9] To paraphrase Stoy [Stoy 77], a *retraction* is a (continuous) function which maps elements of its input domain to a subspace of that domain, with the restriction that it be the identity function on the elements of its range. Its range is therefore a subspace its domain. Quoting, "its range and its fixed point set coincide, and it is therefore idempotent: applying it more than once makes no difference." [*Op. cit.,* p. 133] Another way to view this is as an injective mapping of domain elements to equivalence class representatives in the domain. Formally, again borrowing from Stoy (*ibid.*):

$$f = f \circ f$$

[10] This *retraction* does still support some MIT SCHEME arcana, like `sequence` and `disjunction` and other backward compatible special forms.

[11] Depicted as a ubiquitous Backus Normal Form (BNF) [Backus 59] context-free grammar.

```
--------------------------------------------------------------------------------
<expr> ::= (QUOTE <datum-literal>)      : i.e., symbol, null, pair, vector, etc.
        |  <self-evaling-literal>       : i.e., Boolean, number, char, string

        |        <identifier>           : References (i.e., identifier instance)
        | (SET!   <identifier> <expr>)  : Assignment
        | (DEFINE <identifier> <expr>)  : Note:  <expr> may be a NAMED-LAMBDA

        | (BEGIN >expr< <expr>)         : Here >expr< is <expr> excluding BEGIN
        | (BEGIN >expr< >expr< <expr>)  : (spec.,always 2 or 3 element chains)

        | (NAMED-LAMBDA (<name> <formals>) <expr>)  : NB:  MIT Scheme-specific

        | (<expr> <expr>*)              : Combination/application/invocation

        | (IF <expr> <expr> <expr>)     : No ``one-armed'' IFs
        | (OR <expr> <expr>)            : Two-elt variant only (with nesting)

        | (DELAY <expr>)                : Standard optional library extension

        ................................ MIT Scheme extension:  name spaces

        | (ACCESS  <identifier> <expr>) : MIT Scheme extension
        | (SET!                         : MIT Scheme extension
           (ACCESS <identifier> <expr>)
           <expr>)

        | (IN-PACKAGE <expr> <expr>)    : MIT Scheme extension

        | (THE-ENVIRONMENT)             : MIT Scheme extension

        ________________________________ MIT Scheme extension:  interpreter

        | (UNASSIGNED? <identifier>)    : MIT Scheme extension

        | (      DECLARE <declaration>) : MIT Scheme extension
        | (LOCAL-DECLARE <declaration>  : MIT Scheme extension
            <expr>)

 Note:  <formals> can be null, dotted, ``#!rest'' and/or ``#!optional'' params.
--------------------------------------------------------------------------------
```

The forms below the solid waterline evaporate when compiled, due to macrology. The forms below the dotted waterline are MIT Scheme name-space extensions. Also, both `OR` and `DELAY` are derived syntax that can be re-written in terms of the others above them. Finally, the three-element `BEGIN` form can be trivially re-written as two two-element `BEGIN` forms. That leaves nine core distinct forms total.

Note finally that:

0. This canonicalizes `LAMBDA` expressions to `NAMED-LAMBDA` (MIT SCHEME-ism). It just stores a label in what would otherwise be an anonymous `LAMBDA`. It is helpful for debugging. This carries no semantics beyond `LAMBDA`.[12]

1. All macros are expanded so we needn't worry about the various and sundry macrology special forms (*e.g.,* `LET-SYNTAX`, `DEFINE-MACRO`, *etc.*). By the time we look at the compiled code's source, macros are gone.

2. The following all syntactically evaporate courtesy of de-sugaring:

```
AND, COND, CASE, DO, LAMBDA, named LET, LET, LET*, LETREC, FLUID-LET,
CONS-STREAM, DEFAULT-OBJECT?, DEFINE-INTEGRABLE, DEFINE-STRUCTURE,
MAKE-ENVIRONMENT, QUASIQUOTE, UNQUOTE, UNQUOTE-SPLICING
```

3. Moreover, these special forms all evaporate courtesy of the compiler:

```
UNASSIGNED?, DECLARE, LOCAL-DECLARE
```

   The `UNASSIGNED?` form expands into a primitive with `QUOTE`d *`<identifier>`*.

   The `DECLARE` forms both become `QUOTE`d declaration constants if compiled.

0.2.1 KMP Scheme: An Explicitly Tagged Kernel Subdialect

All the *retraction* into the preceding kernel subdialect is automatically provided by the MIT SCHEME `unsyntax` procedure. For example:

```
1 ]=> (define (or-snark a b c) (or a b c))

1 ]=> (unsyntax/truthfully or-snark)
;Value: (named-lambda (or-snark a b c) (or a (or b c)))
```

...where "`1 ]=>`" is the default level-one, user-interaction input prompt in MIT SCHEME (a.k.a. the so-called "rocket" prompt).

It is common to further transform these to make *all* expressions tagged. This involves introducing vacuous "noise" wrappers around the presently untagged expressions: self-evaluating literals, *identifier*s and combination/application/invocation forms.

[12]Specifically, the embedded *`<name>`* is *not* bound in the environment to this `NAMED-LAMBDA` form by mere virtue of this label. Rather, the reverse holds: the *`<name>`* label appears in the `NAMED-LAMBDA` form by virtue of the environment binding (when it was defined using the `(DEFINE (<name> ...) ...)` variant of the `DEFINE` form).

This is traditionally done as follows (although names vary among implementors):

```
------------------------------------------------------------------------------
        (<CONSTANT> <self-evaling-literal>)     : All capitalized tags here are
        (<CALL> <expr> <expr>*)                 :  literal symbol constant, not
        (<LOOKUP> <identifier>)                 :  grammatical meta-variables.
------------------------------------------------------------------------------
```

For example:

```
;;       -----
(define (fib-x z)
  (if (< z 2)
      z
      (+ (fib-x (-1+ z))
         (fib-x (-   z 2)))))
==>
;;               -----
(define          fib-x
  (named-lambda (fib-x z)
    (if (<call> (<lookup> <) (<lookup> z) (<constant> 2))
        (<lookup> z)
        (<call> (<lookup> +)
                (<call> (<lookup> fib-x)
                        (<call> (<lookup> -1+)
                                (<lookup> z)))
                (<call> (<lookup> fib-x)
                        (<call> (<lookup> -)
                                (<lookup> z)
                                (<constant> 2)))))))
```

This helps make various syntax walkers tag-driven by making all expressions be explicitly tagged. It simplifies the grammar to be trivially left linear with one-token lookahead (`LL(1)`) [Aho & Ullman 72] [Rosenkrantz & Hunt 87].

The resulting explicitly tagged kernel subdialect of SCHEME is often called KMP SCHEME.[13]

0.2.2 Subexpression De-nesting via `Let`-*Extraction*

Typically, the preceding transformation into an explicitly tagged kernel subdialect is the end of the story. For our purposes, however, it is useful to arrange for all non-trivial nested subexpressions to be explicitly named. In essence, this transforms the source code parse tree into one where every non-leaf node is labeled with a lexically non-colliding name. Specifically, this allows the on-line partial evaluator to associate bundles of auxiliary information with each non-trivial subexpression in the specialization environment as it walks the source code, while allowing whole expressions to be "copied" or "duplicated" by name wherever desirable.

[13]This is in honor of Kent M. Pitman, who first coined it. This is not to be confused with the KMP-test used to benchmark partial evaluators: that "KMP" refers to the Knuth-Morris-Pratt algorithm for specializing a simple pattern matcher for a given static pattern.

This is done through a process of *(sub)expression de-nesting*. Its aim is two-fold: 1) to name every non-trivial nested subexpression; then, 2) to refer to them by name in the source code. This is accomplished by a standard transformation technique called `let`-*insertion*[14] [Bondorf & Danvy 90/91] [Mogensen 88] [Danvy 96p] [Jones, Gomard & Sestoft 93, § 5.5.4], where the "trivial" subexpressions are defined to be: self-evaluating literal data, *identifiers*, and syntactic keywords. For example:

[14] Context: [Bondorf & Danvy 90/91] simplified, and thereby popularized, this now-common mechanism, which they attributed to a technique in [Mogensen 88].

They also defined an elegant yet simple *abstract occurrence counting analysis* (an approximating abstract interpretation) to eliminate (reduce) unnecessary insertions in the resulting residualized programs. Since their input source programs were not assumed to be in canonical form, they could not distinguish programmer-authored instances from specializer-injected occurrences. This *abstract occurrence counting analysis* was used subsequently in [Danvy 96p] as well, but that work was stymied by not having access to the source code. There, he contrasts this counting approach to Sestoft's *duplication risk analysis* [Sestoft 88], which employs a similar analysis to annotate expressions rather than transform them into native `LET` forms, in order to avoid *duplicate calls* and *duplicate code*. The former is a correctness issue for the specializer; the later, a space issue for both the specializer and the code it produces.

The technique of `let`-*insertion* is also mentioned in [Jones, Gomard & Sestoft 93, § 5.5.4], but there one is left with the impression that this is done only for arguments of `LAMBDA` forms, much the way careful programmers define their macros to avoid argument expression duplication, as in:

```
(define-macro (square expr) (let ((a expr)) (* a a)))
```

This is an instance of the restricted case of *inserting identity LET expressions in the source code* [Bondorf & Danvy 90/91].

In all the preceding cases, however, `let`-*insertion* is used at specialization generation time, not as a pre-processing input canonicalization. In this regard, `let`-*extraction* in DESCARTES is more akin to what Danvy calls "CPS without continuations" [Danvy 96p] and equates with *monadic normal form* [Hatcliff & Danvy 94] and the *Administrative Normal Form* (*A-normal form*) of [Flanagan *et al.* 93] [Flanagan *et al.* 2003].

The crucial difference, however, is that previous work could not distinguish programmer-introduced `LET` forms from those introduced by input canonicalization. By first re-writing all programmer introduced `LET` forms as explicit `LAMBDA` applications, DESCARTES' `let`-*extraction* makes this distinction clear: *all* `LET` forms in LERKS *normal form* are use-once, specializer-introduced instances. Therefore, extracted `LET` forms can always be reduced at *partial evaluation* time without the need for *abstract occurrence counting analysis* nor *duplication risk analysis*.

This seemingly small point will become critical when considering DESCARTES' partial evaluation rules (§ 0.3, p. 31).

```
;;      -----
(define (fib-x z)
  (if (< z 2)
      z
      (+ (fib-x (-1+ z))
         (fib-x (-   z 2)))))
```

$\Longrightarrow$

```
;;      -----
(define (fib-x z)
  (if (< z 2)
      z
      (let ((fib:z-1 (let   ((z-1 (-1+ z)))
                       (fib-x z-1)))
            (fib:z-2 (let   ((z-2 (-   z 2)))
                       (fib-x z-2))))
        (+   fib:z-1
             fib:z-2))))
```

This illustrates the basic idea but glosses over a few non-obvious details. The following subsections enumerate a few of the more important ones.[15]

Generating Fresh Non-Colliding Names

The simplest among the non-obvious details is how to generate fresh, locally non-colliding names for the newly introduced bindings.

Random name generation (`gensym`) could be used but it obfuscates the code. Since a subsidiary goal of DESCARTES is that specializations be accessible to the programmer in a comprehensible format, a more systematic (less arbitrary) naming convention is employed.

Instead, DESCARTES embeds de Bruijn numbers [de Bruijn 72,95] in syntactically distinguished positional names since they make explicit the lexical depth and offset of where a referenced variable came from in the original nested expression. Such depth and offset annotations should seem natural to programmers while simple syntactic decoration should easily visually distinguish them from *identifier* names introduced in the original code. For example:

[15] One such issue, for example, is why the non-trivial predicate and alternative sub-components of the above `IF` expression were not also extracted out and named. In short, this could alter the semantics of the program in light of non-termination or side effects. Section 0.2.2 addresses this and similar boundary cases— *viz.,* `BEGIN` sequences and `NAMED-LAMBDA` scoping. Interestingly enough, these boundary cases are what make these syntactic forms convenient to retain in the retracted kernel dialect: `IF` for conditional execution, `BEGIN` for serialization, and `NAMED-LAMBDA` for lexical scoping. Of course, in the strictest sense, only `NAMED-LAMBDA` is essential, given Church's ingenious encoding of Booleans (and the natural numbers!) as λ-expressions [Church 51], combined with *applicative order* β-reduction [Barendregt 84]. Such a minimal kernel language would prove pragmatically intolerable for our purposes, albeit theoretically equipotent, suggesting, perhaps, a certain "lack of theology and geometry" [Toole 80,94].

When a true genius appears in the world, you may know him by this sign;
that the dunces are all in confederacy against him.

— JONATHAN SWIFT (1667–1745)
Thoughts on Various Subjects

```
;;       _____    .------------------:------- depth of lexical contour line
(define (fib-x z) | .----------------|-:---- offset of occurrence w/in expr
  (if (< z 2)     | |                | |
      z           v v                v v
      (let ((_Arg_1_1_ (let   ((_Arg_2_1_ (-1+ z)))
                         (fib-x _Arg_2_1_)))
            (_Arg_1_2_ (let   ((_Arg_2_1_ (-   z 2)))
                         (fib-x _Arg_2_1_))))
        (+   _Arg_1_1_
             _Arg_1_2_))))
```

Here the names are generated in mixed case— which users normally won't do so it is a poor man's disjoint name space— with ugly underscore prefix, infix and suffix characters (to set them apart visually from what reasonable SCHEME programmers would use). The buzz tag "Arg" indicates argument position while "Rator" (or just "Rat") indicates operator position (in case the operator is ever itself a non-trivial subexpression or if we decide to extract *all* subexpressions regardless of triviality, as below (p. 23)).

Slightly less obvious, the two numerals embedded within the names reflect the depth of the subexpression nesting and the offset within the current expression. The obvious and straightforward recursive descent algorithm performs this bit of fluff.

To wit, the depth and offset start at zero. Each time a lexical contour line is crossed (*e.g.,* `LAMBDA`, `LET`, and other binding forms) the depth is incremented in the descent. We do not increment the depth count on entry into non-binding special forms (like `IF`).

In our `fib-x` example, `+` is at offset 0 and depth 1 inside the (implied) `NAMED-LAMBDA` expression being `DEFINE`d. The two recursive calls to `fib-x` are nested inside the `+`'s application so they are both at depth 1 with offsets 1 and 2, respectively. Their nested argument subexpressions, in turn, are both at depth 2. They also happen both to be at offset 1.

As Guillermo J. Rozas has noted [Rozas 2007], one way to avoid changing the shape of contours so that every variable is evaluated in an isomorphic context (isotropic contour line)— hence simplifying the code that is ultimately generated— is to extract *all* components of an application form if *any* component requires *de-nesting* extraction. For example, this would result in the following isotropic *de-nesting* of `fib-x`:

```
;;        _____       .------------------:------- depth of lexical contour line
(define (fib-x z)    | .----------------|-:---- offset of occurrence w/in expr
  (if (< z 2)        | |                | |
      z             v v                | |
      (let   ((_Rat_1_0_ +)            v v
              (_Arg_1_1_ (let   ((_Rat_2_0_ fib-x)
                                  (_Arg_2_1_ (-1+ z)))
                           (<CALL> _Rat_2_0_
                                   _Arg_2_1_)))
              (_Arg_1_2_ (let   ((_Rat_2_0_ fib-x)
                                  (_Arg_2_1_ (-   z 2)))
                           (<CALL> _Rat_2_0_
                                   _Arg_2_1_))))
       (<CALL> _Rat_1_0_
               _Arg_1_1_
               _Arg_1_2_))))
```

...where I've taken the liberty of using "Rat" instead of "Rator", and I've inserted explicit `<CALL>` markers in order to make the code more symmetric (nominally, `funcall` in standard LISP parlance [Steele 85/90]).

Amusingly enough, this may actually make the resulting *de-nested* code slightly more readable, at least to my eye, since it allows one to visually ignore the arguably ugly rats' nest of "Rat/Arg" clusters, as well as the interlaced regular sprinkling about of `LET/CALL` bracing, and focus only on the code to the right of these clusters and braces. To the well trained eye, for example, the above imprints my mind as if it read:

```
;;        _____       .------------------:------ yadda yadda yadda
(define (fib-x z)    | .----------------|-:---- wacka wacka wacka
  (if (< z 2)        | |                | |
      z             v v                | |
      (\\\   ((######### +)            v v
              (######### (\\\   ((######### fib-x)
                                  (######### (-1+ z)))
                           (\\\\\\ #########
                                   #########)))
              (######### (\\\   ((######### fib-x)
                                  (######### (-   z 2)))
                           (\\\\\\ #########
                                   #########))))
       (\\\\\\ #########
               #########
               #########))))
```

In effect, therefore, the '`#`' and '\' noise serve only to inflate the already-present nesting structure of the code.

At any rate, I declare by fiat that user programs are not allowed to introduce variable names that would collide with this idiosyncratic naming scheme. More sophisticated machinery could be used to ensure this but that would complicate the exposition whereas this simple method suffices for our purposes.

More importantly, once DESCARTES has specialized a chunk of source code in this form, it is easy to identify which LET-bound *identifiers* can safely be substituted back into the resulting specialized code (assuming they still appear only once each in the transformed (specialized) residual code so that this *de-nesting* can be undone). This, for instance, defuses the objection that inserting LETs where there were none before alters the environment contours of embedded subexpressions and introduces gratuitous closures where none were needed before. Specifically, think of *de-nesting* as a temporary intermediate form that is undone when it has served its purpose. It just provides a labeled parse tree for the partial evaluator to hang its hat while in flight.

This modest modification of standard `let`-*insertion* using disjoint naming to support subsequent identification and reversal of the extractions I dub `let`-*extraction*. The subsequent re-nesting by removal of the inserted LETs is therefore dubbed `let`-*projection*. I tend to prefer the terms *de-nesting* and *re-nesting*, however, since they emphasize the goal over the mechanism by which it is accomplished.

Conditional Predicate Extraction

```
;;       -----
(define (fib-x z)
  (if (< z 2)
      z
      (+ (fib-x (-1+ z))
         (fib-x (-   z 2)))))
```

Note that in the *de-nesting* illustration, I was careful not to `let`-*extract* the non-trivial subexpressions of the IF conditional special form. This is because it would be incorrect to hoist either the consequent clause or the alternative clause from the predicate-guarded body of the IF expression: one and only one of them must be executed at run time, never both in the same invocation.

It is, however, correct (and very useful) to extract and name non-trivial predicate subexpressions. Witness:

```
;;       -----
(define (fib-x z)
  (let ((<>pred<> (< z 2)))      ;;; Name the non-trivial predicate,
   (if  <>pred<>                 ;;;   and then use it by name.
        z
        (+ (fib-x (-1+ z))
           (fib-x (-   z 2))))))
```

Note that no depth or offset embedding is needed for extracted IF predicate clauses since there can be no collision among predicates of cascaded IF chains. This naming strategy does however presume that the *identifier* **<>pred<>** never be used by programmers in the source code, obviously. I declare it so: double diamond names are reserved.

This, then, allows the specializer to propagate useful information about the specialized predicate expression by associating it with **<>pred<>** in the specialization environment during the recursive descent embodied by the partial evaluation process. For instance, in the consequent clause, the partial evaluator "knows" that the predicate was satisfied (non-`false`), while in the alternative clause it must have been `false`. By associating inferred *spectral type* information with **`<>pred<>`**, this kind of constraint inference can bound the inferred spectrum that can be in play along each branch path. This is discussed more formally and clearly later (§ 7).

The treatment of the two-armed kernel `OR` special form is similar:

```
    (or <expr>_1 <expr>_2)
==>
    (let ((<>pred<> <expr>_1))   ;;; For <expr>_1 non-trivial
      (or  <>pred<> <expr>_2))
==>
    (let ((<>test<> <expr>_1))   ;;; For <expr>_1, regardless of triviality
      (if  <>test<>
           <>test<> <expr>_2))
```

For that matter, as the second transformation shows, since we're going to the trouble already of extracting and naming the first subexpression, `OR` can be de-sugared into `IF` in the standard way, using a distinctive name for the extracted first expression so it can be re-written back to the original `OR` form after specialization (so as not to annoy the program inspector).

By always `let`-*extracting* and `IF`-desugaring *all* `OR` expressions regardless of whether or not the first expression is trivial, we have one fewer special form to worry about in the specializer. If the occasion ever arises where it might be useful to distinguish when the original code was actually an `OR` or `IF` (*e.g.,* for debugging or re-nesting), the distinctive **`<>test<>`** injected *identifier* can easily decide the matter.

Summary: "To `LET` or not to `LET`?"[16]

Finally, it is traditional within the *partial evaluation* community to raise an obligatory (if perfunctory) objection to `let`-*insertion* of any sort.

Aside from the obvious complaint that it makes code difficult to read, it also introduces an implied closure (`LET`s are just sugared `LAMBDA` applications, after all) thereby raising the lexical contour depth of source expressions (which carries implications for various run-time optimizations and compile-time peephole transformations). Another way to view this objection is that it takes low-order straight expressions and makes them higher-order by embedding new `LAMBDA` abstractions under the guise of `LET`s.[17] Even though these implied `LAMBDA` liftings are always immediately applied, it still makes the code higher order if the specializer cannot recognize them and treat them specially.

---

[16] A corruption of [Shakespeare 1603, Hamlet, Act III, Scene 1, Line 56].

[17] Strictly speaking, this objection is a bit specious from the outset since introducing a fresh `LET` expression as an explicit `LAMBDA` application does not so much raise the order of the source code (since the new `LAMBDA` expression is immediately applied to the `LET` bindings) as it raises the order of the code analysis: whereas code which contains no first-class `LAMBDA` procedures requires no higher-order analysis, one that naïvely introduces `LET` forms as explicit `LAMBDA` applications might, at least if simply implemented. "You'll pay dearly for your foolishness, Mr.Bond."

Moreover, the target kernel language was free of all `LET` expressions up to this point, but no sooner was this accomplished then *de-nesting* promptly inserted new instances. What then was really accomplished?

DESCARTES sidesteps all these worries by injecting `LET`s that are easily distinguished from any user-introduced `LET/LAMBDA` expressions of the original source code that will have been retracted away. These newly injected `LET` expressions are therefore easily removed from the specialized source before being compiled and loaded into the running system. That was the whole point of `let`-*projection*/*re-nesting* mentioned earlier.

What this accomplishes is the *de-nesting* and labeling of the source code's parse tree temporarily to make specialization easier. It is trivially undone later, *since each newly introduced identifier will appear exactly once within the binding form, without the possibility of naming collisions.*[18] It is a handy temporary intermediate form only: an ephemeral canonicalization.

One might have chosen, for example, not to call encode these extractions as "`LET`"s at all, instead coining a new special keyword— like, say, `BIND` or `CLOSE` or even `<LET>` and such— as a directive to the partial evaluator. Such embedded annotations are not uncommon, usually being performed by a partial evaluation pre-pass that, not surprisingly, is traditionally known as *binding time analysis* (normally abbreviate: BTA) [Jones & Schmidt 80] [Mogensen 89].

In fact, the only important way in which `let`-*extraction* differs from typical *binding time analysis* annotation of source code is that `let`-*extracted* code is directly executable whereas source code annotated with BTA information typically is not.[19]

In short, *reversible* `let`-*extraction* should not offend nor alarm the squeamish. It is as harmless as it is handy, as we shall see soon (§ 0.2.2). In this instance, the concerns expressed above really amount to just so much confusion between text (*intensional*) and executable (*extensional*) representations of programs. To wit, the `let`-*extraction* occurs as one goes from the latter to the former (executable to source) while `let`-*projection* undoes this when going from the former back to the latter (new specialized source to new specialized executable). Embedding the one in the same language as the other (namely, SCHEME) invites this confusion but it aids debugging and tracing of the specializer since all intermediate forms are well-formed SCHEME programs.

This is fairly well known and understood but still haunts some in the community so I thought it worth addressing outright. It's subtle if not straighforward.

One final point is that, instead of embedding `let`-*extraction* wrappers in the code, one could also opt for `NAMED-LAMBDA` explicit applications with distinctive reversible names (a syntactically reversible generalization of the η-conversion rule of the standard λ-calculus called *eta expansion* [Danvy, Malmkjær & Palsberg 94] [Danvy, Malmkjær & Palsberg 95/96][Palsberg 98] or a similar injection of a local definition block (using `BEGIN` and `DEFINE`). These, of course, merely alter the textual appearance of the extraction without altering the semantics of the resulting

[18] This very useful property is exploited later (in the *`<CALL>`* form specialization rule (p. 67) as well as the `LET` form specialization rule (p. 49)), allowing the specializer to seamlessly project `let`-*extracted* forms back into the *specialized code* in passing, obviating the need for an *occurrence counting analysis* [Bondorf & Danvy 90/91].

[19] At least not without, for example, resorting to a *two-level language* [Nielson & Nielson 92] [Nielson & Nielson 96] [Danvy 96, Appendix A], which SCHEME is not.

code. That is to say, although intensionally distinguishable, they are extensionally equivalent. Please do not confuse the conceptual mechanism with a specific implementation.

0.2.3 One Last Refinement: *Rufian* KMP Scheme

In passing, DESCARTES does one last further refinement of *identifier*s and combination/application/invocation forms. It separates them into: a) four disjoint classes of *identifier*; and, 2) five disjoint classes of combination/application/invocation, as follows:

```
-------------------------------------------------------------------------------
        (<LOCAL>     <identifier>)      : Locally bound LET-extracted variable
        (<GLOBAL>    <identifier>)      : Globally bound free variable
        (<PARAMETER> <identifier>)      : Formal parameter of a NAMED-LAMBDA
        (<PRIMITIVE> <identifier>)      : Primitive procedure

        (<CALL_LOCAL>       (<LOCAL>     <identifier>) <expr>*)
        (<CALL_GLOBAL>      (<GLOBAL>    <identifier>) <expr>*)
        (<CALL_PARAMETER>   (<PARAMETER> <identifier>) <expr>*)
        (<CALL_PRIMITIVE>   (<PRIMITIVE> <identifier>) <expr>*)

        (<CALL_NAMED-LAMBDA> (NAMED-LAMBDA ...stuff...) <expr>*)
-------------------------------------------------------------------------------

 Note:  All capitalized tags here are literal symbol constants,
         not grammatical meta-variables.

-------------------------------------------------------------------------------
```

This further simplifies the internal dispatcher of the specializer. It also makes collecting the set of global free variable names trivial. In fact, each tagged datum is inventoried as part of this *de-nesting* process.

For example:

```
    (pp/code (de-nesting-inventory/<GLOBAL>s
              (de-nested-expr/inventory
                (de-nest (unsyntax/truthfully fib-x)))))
==>
    ((<global> fib-x) (<global> fib-x))      ;;; It occurs twice.
```

This last refinement was inspired by Eric Ruf's doctoral dissertation [Ruf 93]. In tribute, the resulting final retracted kernel subdialect of SCHEME used in DESCARTES I dub *Rufian* KMP SCHEME.

On our running `fib-x` example, this retraction looks like the following:

```
;;       -----
(define (fib-x z)
  (if (< z 2)
      z
      (+ (fib-x (-1+ z))
         (fib-x (-   z 2)))))
```

⟹

```
;;               -----
(define          fib-x
  (named-lambda (fib-x z)
    (if (<call_primitive> (<primitive> <) (<parameter> z) (<constant> 2))
        (<parameter> z)
        (<call_primitive> (<primitive> +)
                          (<call_global> (<global> fib-x)
                                         (<call_primitive> (<primitive> -1+)
                                                           (<parameter> z)))
                          (<call_global> (<global> fib-x)
                                         (<call_primitive> (<primitive> -)
                                                           (<parameter> z)
                                                           (<constant> 2)))))))
```

Figure 0-8: The *Rufian* KMP Scheme rendition of `fix-b`

0.2.4 Finally: Let-Extracted *Rufian* KMP Scheme (LERKS) Normal Form

The final fully *de-nested* `Let`-Extracted *Rufian* KMP SCHEME (LERKS) normal form for `fib-x` produced by DESCARTES is:

```
;;               -----
(define          fib-x
  (named-lambda (fib-x z)
    (let ((<>pred<>
           (<call_prim> (<primitive> &<) (<parameter> z) (<constant> 2))))
      (if (<local> <>pred<>)
          (<parameter> z)
          (let ((_Arg_1_2_
                 (let ((_Arg_2_1_
                        (<call_prim> (<primitive> &-)
                                     (<parameter> z)
                                     (<constant> 2))))
                   (<call_global> (<global> fib-x) (<local> _Arg_2_1_))))
                (_Arg_1_1_
                 (let ((_Arg_2_1_
                        (<call_prim> (<primitive> -1+) (<parameter> z))))
                   (<call_global> (<global> fib-x) (<local> _Arg_2_1_)))))
            (<call_prim> (<primitive> &+)
                         (<local> _Arg_1_1_)
                         (<local> _Arg_1_2_)))))))
```

Figure 0-9: The `Let`-Extracted *Rufian* KMP SCHEME (LERKS) Normal Form of `fib-x`

Its *de-nesting* inventory includes the following:

```
-------------------------------------------------------------------------------
   ((<constant> 2) (<constant> 2))          ;;; This occurs twice in the body.

   ((<local> <>pred<>) (<local> _Arg_1_2_)
                       (<local> _Arg_1_1_)
                       (<local> _Arg_2_1_)
                       (<local> _Arg_2_1_))

   ((<global> fib-x) (<global> fib-x))      ;;; This occurs twice in the body.

   ((<parameter> z) (<parameter> z) (<parameter> z) (<parameter> z))  ;;; 4 times

   ((<primitive> &+) (<primitive> -1+) (<primitive> &-) (<primitive> &<))

   ((<call_global> (<global> fib-x) (<local> _Arg_2_1_)) ;;; This occurs twice.
    (<call_global> (<global> fib-x) (<local> _Arg_2_1_)))

   ((<call_prim> (<primitive> &+) (<local> _Arg_1_1_) (<local> _Arg_1_2_))
    (<call_prim> (<primitive> -1+) (<parameter> z))
    (<call_prim> (<primitive> &-) (<parameter> z) (<constant> 2))
    (<call_prim> (<primitive> &<) (<parameter> z) (<constant> 2)))
-------------------------------------------------------------------------------
```

From this inventory, we make the following observations:

1) The set of used formal parameters and free global variables occurring in `fib-x` (excluding primitive procedures) is:

$$\{\texttt{z}, \texttt{fib-x}\}$$

2) The set of primitive procedures in operator position is:

$$\{\texttt{+}, \texttt{-1+}, \texttt{-}, \texttt{<}\}$$

3) The only other object in operator position is `fib-x` itself.

4) The only constants or other literals is the constant 2.

From the above facts, together with the proposition that parameter `z` is an integer and the assumption that `fib-x` is well typed, one may conclude, using only the *Peano Axioms* (or *Peano Postulates*) [Peano 1889,1967/1977] [Dedekind 1890,1967/1977], that this implementaton of `fib-x` is closed on the integers [Dedekind 1888,1901/1963,1996/1999] .[20]

[20] Technically, the Peano Axioms apply only to the natural numbers, but we already noted that `fib-x` is the identity procedure on negative integers, so the extension of our claim of closure to the full set of the integers is immediate.

0.3 Spectral Specialization Rules for Descartes

0.3.0 Let-Extracted *Rufian* KMP Scheme (LERKS) Normal Form Grammar

To summarize, here is the Backus Normal Form (BNF) [Backus 59] context-free grammar for SCHEME programs in Let-Extracted *Rufian* KMP SCHEME (LERKS) Normal Form:

```
--------------------------------------------------------------------------------
<triv> ::= <tlit>
       |   <tref>

<tlit> ::= (QUOTE   <datum-literal>)    : i.e., symbol, null, pair, vector, etc.
       | (<CONSTANT> <self-evaluating>) : i.e., Boolean, number, char, string

<tref> ::= (<LOCAL>     <identifier>)   : Locally bound LET-extracted variable
         | (<GLOBAL>    <identifier>)   : Globally bound free variable
         | (<PARAMETER> <identifier>)   : Formal parameter of a NAMED-LAMBDA
         | (<PRIMITIVE> <identifier>)   : Primitive procedure

<lerk> ::=                      <triv>  : Trivial cases
       | (SET!     <tref>       <triv>) : Assignment of ``trivial'' references
       | (DEFINE   <identifier> <triv>^λ): Note:  <triv>^λ may be a NAMED-LAMBDA

       | (BEGIN  >lerk< <lerk>)         : Here >lerk< is <lerk> excluding BEGIN

       | (NAMED-LAMBDA (<name> <formals>) <lerk>)  : NB:  MIT SCHEME-specific

       | (<CALL> <triv> <triv>*)        : Combination/application/invocation
       | (IF     <triv> <lerk>  <lerk>) : No ``one-armed'' IFs

       | (LET    ((<identifier> <lerk>) : LET-Extraction (not user LET form)
                  ⋮)
           <lerk>)

 Note:  <formals> can be null, dotted, ``#!rest'' and/or ``#!optional'' params.
--------------------------------------------------------------------------------

 Note:  All capitalized tags here are literal symbol constants,
        not grammatical meta-variables.

--------------------------------------------------------------------------------
```

Note the occurrence of `LET` forms introduced by LERKS normalization. These are *not* `LET` forms introduced by the programmer, since those will all have been syntaxed out as explicit `NAMED-LAMBDA` invocations (items 0 and 2 on page 18), as per:

```
--------------------------------------------------------------------------------
        (LET ((<identifier>_1 <expr>_1)
              (<identifier>_2 <expr>_2)
              ⋮)
          <expr>_body)
⟶ℒ
        ((NAMED-LAMBDA (<>She-turned-me-into-a-newt!<>[21]
                        <identifier>_1
                        <identifier>_2
                        ⋮)
           <expr>_body)
         <expr>_1
         <expr>_2
         ⋮)
--------------------------------------------------------------------------------
```

That said, of the following *specialization* rules, all but those for the <CALL> and IF forms are relatively straightforward.

0.3.1 Brief Preview of the Form of Spectral Specialization Rules

Before delving into specific rules, consider their general form.

Each rule takes as input the *source code expression* to be specialized and the *specialization bookkeep*. This so-called "bookkeep" contains the *relevance weight* of the current specialization task, the *spectral context* (as encoded within a *specialization environment*), the queue of *pending specialization* sub-tasks in flight, and so on. Most of these components are ignored by most of the rules. Those which are pertinent will be elucidated as needed within the rules that manipulate them.

The specializer produces as output two results: the *specialized code* for this input in this specialization context, and the inferred *structural type* of the resulting datum denoted by this *specialized code* (hereafter referred to simply as the *specialized type*).[22]

So what is a *specialized type* and how is it generated?

As we shall explore below, this computation of the type of the resulting *specialized code* is rooted in the system (structural) type function, `fulltype`, denoted as domain function $\mathcal{T}$. For example, in a *specialization environment* where variable `two_Pi` is bound to a pair whose left element is 2 and whose right element is 3.14:

$$
\begin{array}{rcll}
\mathcal{T}[\![\texttt{two\_Pi}]\!]\,\mathbf{B} & \triangleq & (\mathbf{B}[\textsc{type}])\,[\![\texttt{two\_Pi}]\!] \equiv \texttt{fulltype}\big((2\ .\ 3.14)\big) & \Longrightarrow\ \texttt{(fixnum . flonum)} \\
\mathcal{T}[\![\texttt{6.022e23}]\!]\,\mathbf{B} & \triangleq & \texttt{fulltype}(6.022e23) & \Longrightarrow\ \texttt{flonum}
\end{array}
$$

[21] [Python 75].

[22] Technically, it also produces a possibly updated *bookkeep* to pass along, but that's best ignored for now. It becomes pertinent only when side-effects occur in the source code (*viz., à la* `SET!` and `DEFINE`), since these perturb the *specialized code* and *specialized type* and such associated with an *identifier* by the *partial evaluation* process.

In the meantime, "I lied about the trees." [Brachman 85]

This depicts $\mathcal{T}$ as an *abstract valuation function* that maps syntactic *identifier*s to their (full) structural types relative to the *specialization environment* of a *specialization bookkeep*, **B** (pretending that *identifier*s are *not* tagged with their reference class (*viz.,* **<LOCAL>**, **<GLOBAL>**, *etc.*)). This is equivalent to computing the `fulltype` of the variable's value, assuming its concrete value is known. In this case, the `fulltype` of pair *(2 . 3.14)* is the type expression `(fixnum . flonum)`. Literal constants are mapped to their `fulltype` directly by $\mathcal{T}$. For example, *6.022e23* is of type `flonum`. We will not extend $\mathcal{T}$ beyond these two trivial classes of input expression: it's just a base case *abstract valuation function* for the specializer to use to build up return types of specialized code.

Note how the structural type conveniently mirrors the structure of the datum described, *i.e.,* the structural type of a pair is represented by a pair comprised of the structural types of the pair's constituents. This I call *categorical isomorphism* (between *concrete value*s and their attendant *abstract type expressions*). It will become handy later.

0.3.2 Function Symmetry between Partial Evaluation & Concrete Valuation

In summary, it is perhaps helpful to reflect briefly on the ways in which the *partial evaluation function.* $\mathcal{PE}$, mirrors the standard *concrete valuation function*, $\mathcal{E}$.

Specifically, the *concrete valuation function* takes an input expression and an environment and produces a value. When top-level `DEFINE` forms are permitted to extend the (global) input environment, this might yield the following domain signature:

$$\mathcal{E}: \;\; < \text{EXPRESSION} \times \text{ENVIRONMENT} > \;\; \to \;\; < \text{VALUE} \times \text{ENVIRONMENT} >$$

Similarly, one might depict the *partial evaluation function*'s domain signature as:

$$\mathcal{PE}: \;\; < \text{EXPRESSION} \times \text{BOOKKEEP} > \;\; \to \;\; < \text{CODE} \times \text{TYPE} \times \text{BOOKKEEP} >$$

Recall (p. 32) that the *partial evaluation*'s *specialization bookkeep*, **B**, is just a modest generalization of the analogous *concrete valuation*'s *valuation environment*, ρ.

The key difference here, then, is that the *concrete valuation function*, $\mathcal{E}$, need not explicitly generate a *type* expression, since the full structural type of the generated value is latently manifest within the concrete value itself. The *partial evaluation function*, $\mathcal{PE}$, by contrast, must generate the (inferred) type of the *specialized code* explicitly, since the *specialized code* is an abstraction, not a concrete value. For example, the *specialized code* might well be a **<CALL>** form— like, say. `(square x)` in a *spectral context* where `x` has type `real`— for which it is not immediately obvious (and therefore not trivially inferred) what the type of the result may be. Valuable information is therefore lost should the *partial evaluator* return only the *specialized code* and not also its concomitant *specialized type.*

An alternative domain model (and the one actually implemented in the DESCARTES prototype) is:

$$\mathcal{PE}: \;\; < \text{EXPRESSION} \times \text{BOOKKEEP} > \;\; \to \;\; < \text{SPECIALIZATION} \times \text{BOOKKEEP} >$$

where

$$\text{SPECIALIZATION} \;=\; < \text{CODE} \times \text{TYPE} >$$

This defines a SPECIALIZATION domain to be comprised of tuples consisting of a *specialized code* instance (from domain CODE) and its companion *specialized type* (from domain TYPE). The *partial evaluation* function, $\mathcal{PE}$, is thus re-defined to map an *expression* and *bookkeep* to a *specialization* and *bookkeep*.

This new domain model highlights the symmetry between $\mathcal{E}$ and $\mathcal{PE}$, while drawing attention to the fact that the *partial evaluation* abstraction corresponding to a *concrete valuation* value is *not* simply a code fragment— it is a *pair* of a code fragment and a type signature— that type signature identifying what type of value is denoted by the code fragment in the context in which it was generated.[23]

That having been established, the *spectral specialization* rules embodied by the DESCARTES prototype follow.

[23]One final note: the hardcore language implementor may note that the above glosses over the *control continuation*, κ, and *persistent store*, σ, typically also included as inputs of a full *concrete valuation function* (*e.g.,* [Stoy 77]). That's true, of course, but I won't dwell on that now (nor define just what these are), except to note that the *partial evaluation* process never exposes their abstract equivalents within the result of a *specialization*, so they are invisible to the outside observer.

That is, just as one typically suppresses explicit mention of the *continuation* and *store* for $\mathcal{E}$ unless and until they are reified (*e.g.,* via capturing a *control continuation* or allocating a datum in the *persistent store* (a.k.a., the "heap")), so too $\mathcal{PE}$ suppresses any such mention since they have no observable reification analogues in the abstract domain of *specializations*.

Within the *partial evaluator* itself, however, these are inherited from the underlying SCHEME runtime system in which the *partial evaluation* process is executed. In this setting, the *bookkeep* serves as a *persistent store* for any global *specializations* that are spawned and registered for future re-use. Nevertheless, this *specialization registry* (p. 60) is merely a device to avoid generating redundant *specializations*. It is therefore a pragmatic issue only, not a semantic instrument with deep theoretical underpinnings. Still, one worth noting in passing. Otherwise, nevermind.

0.3.3 Rule No. 0: *Trivial Literal* Forms

```
-------------------------------------------------------------------------------
<tlit> ::= (QUOTE   <datum-literal>)    : i.e., symbol, null, pair, vector, etc.
       | (<CONSTANT> <self-evaluating>) : i.e., Boolean, number, char, string
-------------------------------------------------------------------------------

 Note:  All capitalized tags here are literal symbol constants,
         not grammatical meta-variables.

-------------------------------------------------------------------------------
```

Explicitly `QUOTE`d forms (like symbols, constant pairs (including lists), constant vectors and so on) are trivial to *partial evaluate*[24] since they denote literal constant data. So too are the self-evaluating literals (like the Booleans (`#t` and `#f`), numeric literals, character constants and so on). Together, these constitute "trivial literals", identified in the BNF grammar by the terminal: ***<tlit>***.[25]

They *partial evaluate* to themselves, and the resulting types of their code specializations are straightforward to compute directly from the input source.

Formally:

Input PE bookkeep: B
Input source code: s ::= (QUOTE *<datum-literal>*) | (<CONSTANT> *<self-evaluating>*)

Specialized code: $\mathcal{PE}[\![\mathtt{s}]\!]\,\mathtt{B} \triangleq \mathtt{s}$
Specialized type: $\mathcal{T}[\![\mathtt{s}]\!]\,\mathtt{B} \triangleq \mathtt{fulltype}([\![\mathtt{s}]\!])$

Output bookkeep: B

Note that the *specialization bookkeep* input, **B**, is not consulted. Only the *type valuation function*, $\mathcal{T}$, is used, which ignores the *bookkeep*, since data literals denote context free constants.

In sum, one can characterize the *code specialization* and companion *specialization type inference* computation sub-tasks for *literal* forms as:

Specialize code: ι (a.k.a. the `identity` function)
Specialize type: $\mathcal{T}$ (a.k.a. the `fulltype` function)

Simplicity itself: *trivial literal*s are literally trivial to *partial evaluate*.

[24] Natural language purists might prefer "partially evaluate" but *partial evaluate* seems prevalent within the *partial evaluation* community, so I follow that convention, albeit denoting such usage as a technical term.

[25] The distinction is made between these two forms merely in order to separate the quoted literals from the non-quoted literals. Specifically, recall (p. 18) that *all* expressions are tagged in LERKS *normal form*, so non-quoted literal expressions must be wrapped by a `<CONSTANT>` tag whereas quoted forms already sport the tag `QUOTE`.

0.3.4 Rule No. 1: *Trivial Reference* Forms

```
--------------------------------------------------------------------------------
<tref> ::= (<LOCAL>     <identifier>)   : Locally bound LET-extracted variable
         | (<GLOBAL>    <identifier>)   : Globally bound free variable
         | (<PARAMETER> <identifier>)   : Formal parameter of a NAMED-LAMBDA
         | (<PRIMITIVE> <identifier>)   : Primitive procedure
--------------------------------------------------------------------------------
 Note:  All capitalized tags here are literal symbol constants,
        not grammatical meta-variables.
--------------------------------------------------------------------------------
```

Variable references are fairly trivial to *partial evaluate*. They simply look up their associated *specialized code* and *specialized type* in the appropriate binding environment within the *bookkeep*. Note the convenient class tags, courtesy of LERKS *normal form*.

Formally:

Input PE bookkeep: $\mathtt{B}$
Input source code: $\mathtt{s} ::= (\texttt{<}tag\texttt{>}\ \texttt{<}identifier\texttt{>})$ for $\texttt{<}tag\texttt{>} ::= \texttt{<LOCAL>} \mid \texttt{<PARAMETER>} \mid \texttt{<GLOBAL>} \mid \texttt{<PRIMITIVE>}$
Specialized code: $\mathcal{PE}[\![\mathtt{s}]\!]\,\mathtt{B} \triangleq \big(\mathtt{B}[\textsc{code}][\texttt{<}tag\texttt{>}]\big)\ \texttt{<}identifier\texttt{>}$
Specialized type: $\mathcal{T}[\![\mathtt{s}]\!]\,\mathtt{B} \triangleq \big(\mathtt{B}[\textsc{type}][\texttt{<}tag\texttt{>}]\big)\ \texttt{<}identifier\texttt{>}$

Output bookkeep: $\mathtt{B}$

In short, the *bookkeep*'s appropriate CODE and TYPE binding environments are consulted in a straightforward *lookup* of the tagged **<identifier>**.[26]

[26] This "$\big(\mathtt{B}[\textsc{code}][\texttt{<}tag\texttt{>}]\big)\ \texttt{<}identifier\texttt{>}$" notation is meant to suggest a two-level array dispatch to access the appropriate *specialization environment*, followed by a *dereference* of the `<identifier>` within that environment. This is *not* intended to connote any specific implementation of the *bookkeep*, however. For profiling purposes, for example, it may well be a *Curried function* [Curry & Feys 58/68].[27]

[27] A "Curried function" is one which takes multiple parameters one at a time rather than taking multiple arguments all at once, such as Curry's famous combinators:

$$\begin{array}{rcl} S & \triangleq & (\lambda x\ .\ (\lambda y\ .\ (\lambda z\ .\ ((x\ z)(y\ z))))) \\ K & \triangleq & (\lambda x\ .\ (\lambda y\ .\ x)) \\ I & \triangleq & (\lambda x\ .\ x) \\ \therefore\quad S(KI) & \equiv & (SK)K\ \equiv\ I \end{array}$$

This so-called "Currying" is commonly employed to restrict the domain of discourse to single-parameter functions only, without loss of generality, in order to simplify logic systems.

This mechanism was first devised by Schönfinkel [Schönfinkel 24,67/77] then widely employed (and thus popularized) by Haskell Curry [Curry & Feys 58/68], after whom it is now traditionally named. A non-eponymous synonymous term might be:

> *kathenometric function*— a function taking multiple parameters, one at a time. This is a largely syntactic device for expressing an otherwise polyadic/multi-argument *parametric function* as an equivalent monadic/unary *monometric function* through the use of cascaded, nested *function closure*s. Synonym: *Curried function*.

One can imagine either a single unified environment or separate environments for each kind of reference (CODE, TYPE, *etc.*), mapping *identifiers* to their associated *specialized code* and *specialized type*, either individually or as a tuple. Such details of the *bookkeep* representation are immaterial here. For now, suffice it to say that the various binding forms (like `NAMED-LAMBDA` and, by extension, `LET` and `<CALL>`) maintain a consistent mapping from *identifiers* to their associated specialized code and type. The global variable map is likewise maintained by global `DEFINE`s and `SET!`s to global *identifiers*.[28] The primitive procedure map is maintained by the SCHEME system internally.

In sum, one can characterize the *code specialization* and companion *specialization type inference* computation sub-tasks for *reference* forms as mere *bookkeep* lookups. This is why I refer to tagged *identifiers* as "trivial references", identified in the BNF grammar by the terminal: *`<tref>`*.

---

[28] Local `DEFINE`s do not appear in code in LERKS *normal form*, courtesy of the `unsyntax` procedure: they are re-written in terms of, say, `LET` (and then `NAMED-LAMBDA`) and `SET!`, at least conceptually anyway (environment contour issues notwithstanding).

For example,

```
(define (arithmetic-parity non-negative-integer)
  ;; Local definitions
  (define (even-or-odd non-neg) (if (zero? non-neg) 'EVEN (odd-or-even (-1+ non-neg))))
  (define (odd-or-even non-neg) (if (zero? non-neg) 'ODD  (even-or-odd (-1+ non-neg))))
  ;; Body
  (even-or-odd non-negative-integer)
  )
```

$\longrightarrow_{\mathscr{L}}$

```
(define (arithmetic-parity non-negative-integer)
  ;; Local bindings
  (let ((even-or-odd ':<To-Be-Assign!ed>)
        (odd-or-even ':<To-Be-Assign!ed>))
    ;; Local assignments
    (set! even-or-odd
     (nλ (even-or-odd non-neg) (if (zero? non-neg) 'EVEN (odd-or-even (-1+ non-neg)))))
    (set! odd-or-even
     (nλ (odd-or-even non-neg) (if (zero? non-neg) 'ODD  (even-or-odd (-1+ non-neg)))))
    ;; Body
    (even-or-odd non-negative-integer)
    )
  )
```

"What is it that hangs on the wall, is green, wet— and whistles?"[29]

One subtle complication, in passing, however, concerns replication of aggregate data, such as quoted pairs, if a variable binding instance is *partial evaluate*d at several distinct textual reference sites, as in:

```
    (LET ((clowns (QUOTE ("Bozo" . "Krusty"))))
      (EQ? clowns clowns))
==> #t
```

It would violate identity semantics to *partial evaluate* this as:

```
    (EQ? (QUOTE ("Bozo" . "Krusty"))
         (QUOTE ("Bozo" . "Krusty")))
==> #f
```

...since the above original `LET` expression evaluates to `#t` while the (incorrect) *partial evaluate*d form shown evaluates to `#f` (since two distinct, though isomorphic, pairs are allocated in the run-time heap: they look the same but are *not* the same object in memory).

Note that this is not merely an efficiency concern, either for the *partial evaluator* to avoid duplicate effort at partial evaluation time or for the resulting *specialized code* to avoid allocating redundant storage at run time: it is a correctness concern, as the resulting *specialized code* does not properly preserve the semantics of the original source code.

This is a bit of a red herring, though, since the real issue is in `LET`-binding of identifiers in the *specialization environment* during *partial evaluation* of the `LET` expression's body, not a problem with the specialization rule for references *per se*. So, although this is where the mistake will manifest itself if made, this is not the root cause of the difficulty.

I mention this here as the question naturally arises here, but I defer details of its resolution until later, where it is more naturally addressed.[30]

---

[29] From [Rosen 68,2001]:

> The first riddle I ever heard, one familiar to almost every Jewish child, was propounded to me by my father:
>
> "What is it that hangs on the wall, is green, wet— and whistles?"
> I knit my brow and thought and thought, and in final perplexity gave up.
> "A herring," said my father.
> "A herring," I echoed. "A herring doesn't hang on the wall!"
> "So hang it there."
> "But a herring isn't green!" I protested.
> "Paint it."
> "But a herring isn't wet."
> "If it's just painted it's still wet."
> "But— " I sputtered, summoning all my outrage, "— a herring doesn't whistle!!"
> "Right," smiled my father. "I just put that in to make it hard."
>
> — Leo Rosten *The Joys of Yiddish*

Jokes Mailing List Archive, Joke Number 127 [`http://www.hehe.at/funworld/archive/fun4you.php?joke=127`].

[30] Specifically, the solution involves a delicate handling of `LET` as LERKSed manifest `NAMED-LAMBDA` calls. So the real issue is one of `NAMED-LAMBDA` forms as operators to `<CALL>` expressions, as we'll see below (p. 67).

0.3.5 Rule No. 2: SET! Forms

```
-----------------------------------------------------------------------------
        (SET! <tref> <triv>)          : Assignment of ''trivial'' references
-----------------------------------------------------------------------------
```

... where (as before):

```
-----------------------------------------------------------------------------
<tref> ::= (<LOCAL>     <identifier>)   : Locally bound LET-extracted variable
         | (<GLOBAL>    <identifier>)   : Globally bound free variable
         | (<PARAMETER> <identifier>)   : Formal parameter of a NAMED-LAMBDA
         | (<PRIMITIVE> <identifier>)   : Primitive procedure
-----------------------------------------------------------------------------
```

To *partial evaluate* an assignment statement, we first *partial evaluate* the value constituent, ***<triv>***, then generate a new, specialized assignment statement from the old by substituting this new *specialized code* value for the old ***<triv>*** subexpression.[31] In passing, we *update* the binding associated with the specified target reference, ***<tref>***, in the appropriate *specialization environment* within the *specialization bookkeep*, to reflect the resulting *specialized code* and *specialized type* of this specialized value.

Thankfully, LERKS *normal form* grammar conveniently makes this a fairly simple pair of tasks, since the value constituent is either a *trivial literal* or *trivial reference* (courtesy of `let`-*extraction*, p. 19). Likewise, the target reference is comparably trivial, having been LERKS normalized to a ***<tref>***.

Formally:

$$
\begin{array}{ll}
\texttt{Input PE bookkeep:} & \mathtt{B} \\
\texttt{Input source code:} & \mathtt{s} ::= (\texttt{SET!}\ (\mathit{<tag>}\ \mathit{<identifier>})\ \mathit{<triv>}) \\
 & \quad \text{for} \quad \mathit{<tag>} ::= \texttt{<LOCAL>}|\texttt{<GLOBAL>}|\texttt{<PARAMETER>}|\texttt{<PRIMITIVE>} \\
 & \\
\texttt{Specialized code:} & \mathcal{PE}[\![\mathtt{s}]\!]\,\mathtt{B} \triangleq [\![(\texttt{SET!}\ (\mathit{<tag>}\ \mathit{<identifier>})\ \boxed{\mathcal{PE}[\![\mathit{<triv>}]\!]\,\mathtt{B}})]\!] \uparrow \mathcal{S} \\
\texttt{Specialized type:} & \mathcal{T}[\![\mathtt{s}]\!]\,\mathtt{B} \triangleq \textsc{unit} \ \text{ or } \ (\mathtt{B}\,[\textsc{type}][\mathit{<tag>}])\ \mathit{<identifier>} \\
 & \qquad\qquad\quad \text{ or } \ (\mathtt{B}'[\textsc{type}][\mathit{<tag>}])\ \mathit{<identifier>} \\
 & \\
\texttt{New bookkeep code:} & \mathtt{B}' := (\mathtt{B}[\textsc{code}][\mathit{<tag>}])\,[\mathit{<identifier>} \mapsto \mathcal{PE}[\![\mathit{<triv>}]\!]\,\mathtt{B}\,] \\
\texttt{New bookkeep type:} & \mathtt{B}' := (\mathtt{B}[\textsc{type}][\mathit{<tag>}])\,[\mathit{<identifier>} \mapsto \mathcal{T}[\![\mathit{<triv>}]\!]\,\mathtt{B}\,] \\
 & \\
\texttt{Output bookkeep:} & \mathtt{B}'
\end{array}
$$

The *specialized code* for the `SET!` expression itself is thus just the original input expression (since dynamic assignments cannot be statically eliminated, at least not in the general case),[32]

[31] *I.e.,* $[(\mathcal{PE}[\![\mathit{<triv>}]\!]\,\mathtt{B})/\mathit{<triv>}]\,\mathtt{s}$, where "[new/old] expr" denotes the (prefix) substitution of new for old in expr. This is the traditional notation from the field. I prefer instead to show the substitution outright, as seen in the *specialized code* definition shown here.

[32] Standard program analysis techniques like liveness analysis, control flow analysis and dead code elimination could be brought into play to, perhaps, eliminate provably non-observable assignments, but I prefer to defer that

but with the value constituent replaced with its *partial evaluated* equivalent.

The only minor subtlety is that the *specialized type* of the overall `SET!` form is either UNIT or the *specialized type* of the old or new value. depending on which dialect of SCHEME is being specialized. That is, in standard SCHEME [IEEE 91], the value returned by a `SET!` form is unspecified, so the *specialized type* of the overall `SET!` will vary, and may even be discarded, depending on the underlying SCHEME dialect.

To summarize, the *bookkeep*'s appropriate CODE and TYPE binding environments are consulted in a straightforward *lookup* of the tagged `<identifier>`. The old *specialized code* is replaced and the old *specialized type might* be returned as the result of the specialization or, depending on the language implementation, an unspecified "unit" semaphore type, UNIT, might be returned (to merely signal completion), or even the *specialized type* of the now-specialized new value.[33]

More important is the effect an assignment form has on the *partial evaluation* process itself. Namely:

```
Input PE bookkeep:  B
Input source code:  s ::= (SET! (<tag> <identifier>) <triv>)
                          for  <tag> ::= <LOCAL>|<GLOBAL>|<PARAMETER>|<PRIMITIVE>

New bookkeep code:  B' :=  (B[CODE][<tag>]) [<identifier> ↦ PE⟦<triv>⟧B]
New bookkeep type:  B' :=  (B[TYPE][<tag>]) [<identifier> ↦  T⟦<triv>⟧B]

Output bookkeep:    B'
```

In short, the "trivial" argument value constituent, `<triv>`, is *partial evaluate*d in the *specialization bookkeep* and its resulting *specialized code* and *specialized type* replace the old associations for this *identifier* within the *specialization environment*.[34]

One can imagine doing this *bookkeep* update in (at least) one of three ways. The first and most aggressive approach is to update the existing *identifier* binding frames in place within the *bookkeep* data structure. On the other hand, a slightly less aggressive approach is to generate fresh new frames to associate the *identifier* with its new code and type, then append these to the head of the *specialization environment*(s), either "shadowing" (overriding) the old frame

sort of non-local "peep hole" optimization to the underlying SCHEME compiler.

[33] Although the SCHEME language standard [IEEE 91] does not specify a return value for `SET!`, MIT SCHEME returns the old (pre-`SET!`) value, in order to provide an atomic "test and set" [Ward & Halstead 90] operator (called "atomic swap" on the left coast [Patterson & Hennessy 2006] [Hennessy & Patterson 2005/2007]). Other SCHEME implementations return a distinguished "unit" object. Still others return arbitrage of various ilk. Though possibly an amusing digression, and certainly the source of vigorous, if sometimes contentious, debate withing the SCHEME community, this detail is not critical for our present purposes. I won't bring it up again but felt obliged to at least acknowledge the issue in passing, if just this once.

[34] This "(B[CODE][<tag>]) [<identifier> ↦ $\mathcal{PE}$⟦<triv>⟧B]" notation is meant to suggest a two-level array dispatch to access the appropriate *specialization environment*, followed by a *rebinding* of the `<identifier>` within that environment. This is *not* intended to connote any specific implementation of the *bookkeep*, however. *Cf.,* footnote (26), p. 36.

or actively removing the old frame to produce a new environment structure, possibly sharing structure with the old environment if not just the old unaffected binding frames. The resulting new environment could then either be installed directly in place of the old environment within the overall *bookkeep*, or, on the third hand,[35] a new *bookkeep* can be generated and passed along as part of the result of the specialization (*i.e.,* along with the *specialized code* and *specialized type*).

Those familiar with *fluid binding* (*e.g.,* as embodied by SCHEME's `FLUID-LET` syntactic form) may recognize this as an instance of *deep binding* versus *shallow binding* versus a *functional* approach. This is a pragmatic implementation choice, but it carries implications for just what sort of *partial evaluation* process is generated. Some are more easily proved correct (and made self-specializing) than others. For the DESCARTES prototype, I've chosen a functional approach, where new frames are added to shadow the old and the resulting new environment(s) are used to construct a new *bookkeep* from the components of the old. This is then passed along as part of the result of each specialization rule.

All told, whereas a normal language interpreter, in the general case, takes as input an expression, environment, and mutable store [Stoy 77] and produces as output a result expression, environment and store, the DESCARTES specializer's *partial evaluation* process in this functional form needs no such mutable store. That is, mutation in the programs being specialized need not entail side effects within the process being used to specialize them.

This may come as a mild surprise to those who may have assumed that a dynamic (run-time) on-line specializer must at some point resort to running code "under the rug", within the underlying language SCHEME process, for example, and therefore may incur side effects in the shared process space— that is, the process space of the specializer and the process space of the code being run-time specialized. To simplify the system hygiene, (and to facilitate debugging and profiling of the specializer itself), I opted to make the specializer process purely functional.

In sum, one can characterize the *code specialization* and companion *specialization type inference* computation sub-tasks for `SET!` forms as a simple update of the *specialization bookkeep*, replacing the *specialized code* and *specialized type* associated with the target `<identifier>` to that of the trivial `<triv>` argument.

[35] [Niven & Pournelle 74].

0.3.6 Rule No. 3: DEFINE Forms

```
--------------------------------------------------------------------------------
        (DEFINE <identifier> <triv>^λ)   : Note:  <triv>^λ may be a NAMED-LAMBDA
--------------------------------------------------------------------------------
```

Thankfully, internal `DEFINE`s will have been eliminated courtesy of LERKS *normal form* (*cf.*, footnote (28), p. 37), so only global `DEFINE`s need be considered in this rule.

Given that, `DEFINE` forms are the equivalent of a `SET!` to the ***<GLOBAL>*** *specialization environment*. If no binding yet exists for the target ***<identifier>***, one is created. The resulting updated *bookkeep* is then propagated along with the returned *specialized code* and *specialized type*.

The resulting *specialized code* for the `DEFINE` form itself is the result of substituting this specialized ***<triv>***$^\lambda$ value into the `DEFINE` form. The *specialized type* is like that of `SET!` (§ 0.3.5, p. 39–p. 40), only more so: since no previous value will exist for newly defined variables, the returned value of a `DEFINE` form is unspecified, although some implementations return the name of the variable (***<identifier>***).

Formally:

```
Input PE bookkeep:   B
Input source code:   s ::= (DEFINE <identifier> <triv>^λ)

Specialized code:   𝒫ℰ⟦s⟧B  ≜  ⟦(DEFINE <identifier> (𝒫ℰ⟦<triv>^λ⟧B))⟧ ↑ 𝒮
Specialized type:    𝒯⟦s⟧B  ≜  UNIT   or   SYMBOL

New bookkeep code:  B′ :=  (B[CODE][<GLOBAL>]) [<identifier> ↦ 𝒫ℰ⟦<triv>^λ⟧B]
New bookkeep type:  B′ :=  (B[TYPE][<GLOBAL>]) [<identifier> ↦  𝒯⟦<triv>^λ⟧B]

Output bookkeep:    B′
```

One minor grammatical quibble is the restriction of the value argument to ***<triv>***$^\lambda$, denoting either a ***<triv>*** or `NAMED-LAMBDA` form. This simplifies the *termination* proof for the *partial evaluation* process. The admission of `NAMED-LAMBDA` is a concession to the native MIT Scheme **`unsyntax`**ing of "call template" procedure definitions into `NAMED-LAMBDA` forms (p. 16):

```
       (DEFINE (<name> <formals>) <body>)   ;;; Note the "(<name> <formals>)" parens.
⟶MIT
       (DEFINE <name> (NAMED-LAMBDA (<name> <formals>) <body>))   ;;; In MIT SCHEME
```

Here the second (rightmost) appearance of ***<name>*** is a noise marker of no semantic significance.

One could as well relax this to allow a general ***<lerk>*** in place of ***<triv>***$^\lambda$, but the current restriction suffices since a general value can always be assigned to the defined *identifier* with a subsequent `SET!` (of a possibly `let`-*extracted* value) once the binding has been established via `DEFINE`.

In fact, given this observation, one could restrict, rather than relax, the `DEFINE` form in LERKS *normal form* to admit only the MIT Scheme non-binding form:

```
(DEFINE <identifier>)
```

This merely establishes the ***<identifier>*** in the environment without ascribing a binding value to it— a subsequent `SET!` accomplishes that.

The corresponding LERKS normalalization schema for "call template" procedure `DEFINE` forms would then be:

```
(DEFINE (<name> <formals>) <body>)   ;;; Note the ''(<name> <formals>)'' parens.
```

$\longrightarrow_{\mathscr{L}}$

```
(BEGIN
  (DEFINE <name>)
  (LET         ((<>definiendum<>  (NAMED-LAMBDA (<name> <formals>) <body>)))
    (SET! <name> <>definiendum<>))
  )
```

The current ***<triv>***$^{\lambda}$ schema, therefore, is a workable compromise between the two extremes of a fully general ***<lerk>*** schema or a fully restricted non-binding `DEFINE` schema.

In sum, one can characterize the *code specialization* and companion *specialization type inference* computation sub-tasks for (global) `DEFINE` forms as a minor variation on the `SET!` rule. The only distinguishable difference occurs in the *specialized type*. Since the return value of a `DEFINE` form— and, for that matter, `SET!` as well— is not specified by the language standard anyway [IEEE 91], this is not a substantive difference. Still, it was a worthwhile digression all the same, even if solely pragmatic,

0.3.7 Rule No. 4: `BEGIN` Forms

```
-----------------------------------------------------------------------------------
        (BEGIN >lerk< <lerk>)          : Here >lerk< is <lerk> excluding BEGIN
-----------------------------------------------------------------------------------
```

This rule exposes a few subtleties with ensuing simplifications until, ultimately, it evaporates.

Thankfully, we needn't worry about nested `BEGIN`s, since LERKS *normal form* transforms these into linear chains of cascaded dyadic `BEGIN`s, as per:

```
      (BEGIN (BEGIN a b) (BEGIN (BEGIN c (BEGIN d e) f) (BEGIN g)))   ;;; "nested" polyadic
```
$\longrightarrow_{\mathscr{L}}$
```
      (BEGIN a (BEGIN b (BEGIN c (BEGIN d (BEGIN e (BEGIN f g))))))   ;;; "chained" dyadic
```

or, more to the point:

```
      (BEGIN (BEGIN (foo bar) (baz quux)) (mumble frotz) (BEGIN (snark fnord)))
```
$\longrightarrow_{\mathscr{L}}$
```
      (BEGIN (foo bar) (BEGIN (baz quux) (BEGIN (mumble frotz) (snark fnord))))
```

Note that the resulting de-nested, chained dyadic forms are careful *not* to alter the evaluation order of the original sub-forms. The point is that the first term of every (now-dyadic) `BEGIN` form is never itself a manifest `BEGIN` form. These constituents, however, are *not* trivial: they can be any class of `<lerk>` *except* a `BEGIN` form.[36]

The resulting specialization rule for LERKS-normal `BEGIN`s is relatively straighforward. The leading `>lerk<` is specialized, its *specialized code* is embedded into the resulting `BEGIN` form, and its *specialized type* is discarded. The resulting (presumably updated) *bookkeep* is then used to *partial evaluate* the subsequent `<lerk>`. The resulting *specialized code* is likewise embedded into the resulting `BEGIN` form, and its *specialized type* and (potentially further updated) *bookkeep* are then returned as the *specialized type* and *bookkeep* of the overall `BEGIN` form.

Formally:

`Input PE bookkeep:` B

`Input source code:` s ::= (BEGIN `>lerk<` `<lerk>`) for "`>lerk<`" a non-BEGIN

$$\texttt{Specialized code:}\quad \mathcal{PE}[\![\mathtt{s}]\!]\,\mathrm{B} \triangleq [\![(\mathrm{BEGIN}\ \boxed{\mathcal{PE}[\![\mathtt{>lerk<}]\!]\,\mathrm{B}}\ \boxed{\mathcal{PE}[\![\mathtt{<lerk>}]\!]\,\mathrm{B}'})]\!]\uparrow\mathcal{S}$$

$$\texttt{Specialized type:}\quad \mathcal{T}[\![\mathtt{s}]\!]\,\mathrm{B} \triangleq \mathcal{T}[\![\mathtt{<lerk>}]\!]\,\mathrm{B}'$$

`where`

$$\mathrm{B}' \longleftarrow \boxed{\mathcal{PE}[\![\mathtt{>lerk<}]\!]\,\mathrm{B}}$$

$$\mathrm{B}'' \longleftarrow \boxed{\mathcal{PE}[\![\mathtt{<lerk>}]\!]\,\mathrm{B}'}$$

`Output bookkeep:` B''

Note the connecting arrows in the above, which signify consumption dependency constraints on the constituent *partial evaluation* subtasks. This complication can be avoided, as follows.

[36] This, for instance, allows the *partial evaluator* processes to linearly iterate over `BEGIN` forms rather than worry about nested recursive tree descent. It's a pragmatic issue.

Rule No. 4′: `BEGIN` Forms (Simplified)

```
------------------------------------------------------------------------------
         (BEGIN (<LOCAL> <>!<>) <lerk>) : Old >lerk< was <lerk> excluding BEGIN
------------------------------------------------------------------------------
          N.B.: (<LOCAL> <>!<>) <------ was a >lerk<.
```

Specifically, one could extend the *specialization bookkeep* to record not only the CODE and TYPE associated with *identifiers* but also the resultant BOOKKEEP of their *partial evaluate*d binding forms. This then could exploit the underlying domain signature of $\mathcal{PE}$ (p. 33) to simplify both the LERKS *normal form* grammar entry for `BEGIN` forms and the ensuing `BEGIN` specialization rule, as follows.

First, in light of the fact that internal `DEFINE`s will have been eliminated courtesy of LERKS *normal form* (*cf.,* footnote (28), p. 37), LERKS normalization needn't preserve the singular lexical contour line of `BEGIN` chains. Specifically, the leading subexpression can be `let`-*extracted.* Thus, the following new canonicalization schema can be added as part of LERKS normalization:

```
       (BEGIN (foo bar) (BEGIN (baz quux) (mumble frotz)))
⟶𝓛
       (LET   ((<>!<> (foo bar)))
         (BEGIN <>!<> (LET   ((<>!<> (bar quux)))
                       (BEGIN <>!<> (mumble frotz)))))
```

Note that the embedded "`<>!<>`" magic marker is a fixed, distinguished, locally `LET`-bound *identifier.* This admits the simpler `BEGIN` entry in the LERKS *normal form* grammar, as shown above.

Formally, the former `BEGIN` specialization rule now can be simplified to:

`Input PE bookkeep:` B
`Input source code:` s ::= (BEGIN (<LOCAL> <>!<>) *<lerk>*)

`Specialized code:` $\mathcal{PE}[\![\mathtt{s}]\!]\mathtt{B} \triangleq [\![(\mathtt{BEGIN}\ \Diamond' \ \boxed{\mathcal{PE}[\![\mathtt{<}lerk\mathtt{>}]\!]\mathtt{B}'})]\!] \uparrow \mathcal{S}$
`Specialized type:` $\mathcal{T}[\![\mathtt{s}]\!]\mathtt{B} \triangleq \mathcal{T}[\![\mathtt{<}lerk\mathtt{>}]\!]\mathtt{B}'$

`where`
$\Diamond' \triangleq \big(\mathtt{B}[\textsc{code}][\textsc{local}]\big)\ \mathtt{<>!<>}$
$\mathtt{B}' \triangleq \big(\mathtt{B}[\textsc{bookkeep}][\textsc{local}]\big)\ \mathtt{<>!<>}$
$\mathtt{B}'' \hookleftarrow \boxed{\mathcal{PE}[\![\mathtt{<}lerk\mathtt{>}]\!]\mathtt{B}'}$

`Output bookkeep:` $\mathtt{B}''$

Note that this hard-wires the *bookkeep* lookups for the local "`<>!<>`" magic marker's CODE and BOOKKEEP bindings. If we wanted to allow this *identifier* to range over fresh generated names (say, for debugging purposes), the `(<LOCAL> <>!<>)` could be replaced in the above with, say, ***<tref>***, with commensurate changes.

Of course, by exploiting the underlying domain signature of $\mathcal{PE}$ (p. 33) in this way to leverage the latent BOOKKEEP propagation, an even simpler rule is possible, as follows next.

Rule No. 4″: `BEGIN` Forms (Overdrive)

```
-------------------------------------------------------------------------------
         (BEGIN (<LOCAL> <>this!<>)      : Was ``(<LOCAL> <>!<>)'' magic marker
                (<LOCAL> <>that!<>))     : Was <lerk>
-------------------------------------------------------------------------------
```

Taking this one step further, one can leverage to the hilt the *applicative order* semantics of β-reduction [Barendregt 84] implicit in the `LET` form, as follows.

First, the following new canonicalization schema is added as part of LERKS normalization of dyadic `BEGIN` forms:

```
       (BEGIN (foo bar) (BEGIN (baz quux) (mumble frotz)))
```

$\longrightarrow_{\mathscr{L}}$

```
       (LET     ((<>this!<> (foo bar)))
         (LET   ((<>that!<> (LET     ((<>this!<> (baz quux)))
                              (LET   ((<>that!<> (mumble frotz)))
                                (BEGIN <>this!<>
                                       <>that!<>)))))
           (BEGIN <>this!<>
                  <>that!<>)))
```

This admits the trivial `BEGIN` entry in the LERKS *normal form* grammar, as shown above. Moreover, note that this is still trivially reversed back into the original source text, as per the *specialized code* below. (More aggressive, further transformation could easily lose this property.)

Formally, the former `BEGIN` specialization rule now can be simplified to:

`Input PE bookkeep:` B
`Input source code:` s ::= `(BEGIN (<LOCAL> <>this!<>)`
`(<LOCAL> <>that!<>))`

`Specialized code:` $\mathcal{PE}[\![\mathtt{s}]\!]\,\mathtt{B} \;\triangleq\; [\![\mathtt{(BEGIN}\ \Diamond'\ \Diamond''\mathtt{)}]\!] \uparrow \mathcal{S}$
`Specialized type:` $\mathcal{T}[\![\mathtt{s}]\!]\,\mathtt{B} \;\triangleq\; \mathcal{T}[\![\mathtt{<}lerk\mathtt{>}]\!]\,\mathtt{B}'$

`where`

$\Diamond' \triangleq \big(\mathtt{B}[\textsc{code}][\textsc{local}]\big)\ \mathtt{<>this!<>}$

$\Diamond'' \triangleq \big(\mathtt{B}[\textsc{code}][\textsc{local}]\big)\ \mathtt{<>that!<>}$

$\mathtt{B}' \triangleq \big(\mathtt{B}[\textsc{bookkeep}][\textsc{local}]\big)\ \mathtt{<>this!<>}$

$\mathtt{B}'' \triangleq \big(\mathtt{B}[\textsc{bookkeep}][\textsc{local}]\big)\ \mathtt{<>that!<>}$

`Output bookkeep:` B″

Note that this hard-wires the *bookkeep* lookups for both the local "**`<>this!<>`**" and "**`<>that!<>`**" magic markers' CODE and BOOKKEEP bindings. If we wanted to allow these *identifiers* to range over fresh generated names (say, for debugging purposes), the `(<LOCAL> <>hack!<>)` could be replaced in the above with, say, $\mathtt{<}\mathit{tref}\mathtt{>}_i$, with commensurate changes.

But wait! There's more!! ...

Rule No. 4‴: BEGIN Forms (Redux)

```
------------------------------------------------------------------------------
        ``BEGINs!?  I don't have to show you any stinking BEGINs!''[37]
------------------------------------------------------------------------------
```

Given the preceding, one might be tempted simply to eliminate `BEGIN` forms entirely from the LERKS *normal form* grammar, as follows.

```
       (BEGIN (foo bar) (BEGIN (baz quux) (mumble frotz)))
-->L
       (LET     ((<>Fnord!<> (foo bar)))
         (LET   ((<>Fnord!<> (baz quux)))
           (mumble frotz)))
```

This exploits the observation that the intermediate **`<>Fnord!<>`** variables need not be retained beyond their point of binding, so long as the `LET` rule is careful to propagate the (potentially updated) BOOKKEEP latent in the underlying domain signature of $\mathcal{PE}$ (p. 33). In a phrase: "Don't see the *Fnord!*s!"[38]

Formally, this entirely supplants the `BEGIN` rule with the `LET` rule (which follows, p. 49).[39]

The problem with this, however, is that it violates the invariant that `let`-*extracted identifiers* always appear *exactly once* within the body of the generated `LET` form (§ 0.2.2, p. 26). I emphasize the stipulation "*exactly once*" here since that's crucial to the correctness of the `LET` rule (which follows, p. 49). Specifically, each `let`-*extracted identifier* must occur *at most* one time, *but no fewer*, in order for the resulting *specialized code* to be correct (as we shall see).

The key point is that, although valid for standard SCHEME semantics [IEEE 91], this elimination of `let`-*extracted identifiers* from the `LET` body is *not* valid for the LERKS *normal form* dialect. Albeit executable as SCHEME source, LERKS *normal form* code is *not* "just SCHEME": it is a carefully constrained subdialect designed specifically for the *partial evaluation* rules.

For this reason, in order to satisfy this *exactly once* invariant, the above re-write would instead have to introduce manifest `NAMED-LAMBDA` forms rather than `LET`s, as per:

```
       (BEGIN (foo bar) (BEGIN (baz quux) (mumble frotz)))
-->L
       ((NAMED-LAMBDA    (_Begun_ <>Fnord!<>)
          ((NAMED-LAMBDA (_Begun_ <>Fnord!<>)
             (mumble frotz))  ;;; Invoked 3rd
           (baz quux)))       ;;; Invoked 2nd
        (foo bar))            ;;; Invoked 1st
```

This avoids violating the `LET` invariant for code in LERKS *normal form* by injecting equivalent explicit `NAMED-LAMBDA` invocations instead of `LET` forms.[40] Alas, it also obfuscates the original

[37] With apologies to [Traven 27,35] and [Houston & Traven 48].

[38] [Shea & Wilson 76,84]!

[39] This, by the way, is why the "*"Gang of Eight" (or "Nine", depending on how you count)*" qualifier appeared in item 0 on page 15: if you include `BEGIN` then there are nine special forms in the final, minimal BNF grammar on page 16; otherwise only eight distinct forms remain, *apropos* the concluding comment there.

[40] For clarity, I forgo the subsequent `<CALL_NAMED-LAMBDA>` decorations that would also be required for strict LERKS *normal form* canonicalization here.

`BEGIN` source code. By generating the `NAMED-LAMBDA` embedded names systematically, however, this transformation is also reversible, as is a key feature of `let`-*extraction* (as argued on earlier, p. 26).[41] Consider, for example:

```
        (BEGIN (foo bar)
               (baz quux)
               (mumble frotz)
               (grue bleen))
```

$\longleftrightarrow_{\mathscr{L}}$

```
        (BEGIN (foo bar)
               (BEGIN (baz quux)
                      (BEGIN (mumble frotz)
                             (grue bleen))))
```

$\longleftrightarrow_{\mathscr{L}}$

```
        ((NAMED-LAMBDA       (_Begun_ <>Fnord!<>)
           ((NAMED-LAMBDA    (_Begun_ <>Fnord!<>)
              ((NAMED-LAMBDA (_Begun_ <>Fnord!<>)
                 (grue bleen)) ;;; Invoked 4th
               (mumble frotz)) ;;; Invoked 3rd
            (baz quux)))       ;;; Invoked 2nd
         (foo bar))            ;;; Invoked 1st
```

This stratagem, therefore, has the advantage of totally obviating the need for a separate `BEGIN` rule. The net result, then, is that only the `IF` and `NAMED-LAMBDA` forms (and, by extension, `LET`) contain fully general ***`<lerk>`*** sub-terms in the LERKS normalized grammar (p. 31). All other grammatical elements are non-recursive. This is theoretically appealing since it simplifies the *termination* analysis of the *partial evaluation* process by limiting it to *only* those forms that *require* syntactic recursion.

Restricting grammatical recursion in this way to only the *conditional* and *procedural* cases distills the issue to its core. Moreover, careful derivation of this result from the initial naïve `BEGIN` rule helped draw attention to the subtlety that the LERKS *normal form* sub-dialect is delicately constructed to simplify the *partial evaluation* rules, which collectively are its only intended direct consumer. It is hoped, therefore, that this `BEGIN` rule digression was pedagogically justified.

In sum, one can characterize the *code specialization* and companion *specialization type inference* computation sub-tasks for `BEGIN` forms as altogether superfluous, given an appropriately extended LERKS *normal form* that re-writes `BEGIN` forms into staged `NAMED-LAMBDA` explicit applications.

This, however, is predicated on the delicate constraint that the `LET` *partial evaluation* rule *Do the Right Thing*^{TM} [Lee 89]. Which it does... as we shall see next.

---

[41] Even well-seasoned programmer's don't appreciate having their original source code transmogrified when done solely to simplify an internal run-time tool; I being among them. Designing the intermediate, internal code representation to be a trivially reversible transformation on the original syntax makes it possible to unobscure the code should it become necesssary to display warnings or errors or admit interactive debugging, for example.

0.3.8 Rule No. 5: LET Forms

```
--------------------------------------------------------------------------------
         (LET ((<identifier>_1 <lerk>_1)  : LET-Extraction (not user LET form)
               (<identifier>_2 <lerk>_2)
                ⋮)
            <lerk>_body)
--------------------------------------------------------------------------------
```

Note that these are LET forms introduced by LERKS normalization, not those appearing in the original source program. These are *not* general LET forms introduced by the programmer, since those will all have been unsyntaxed out as explicit NAMED-LAMBDA invocations (as per items 0 and 2 on page 18), as illustrated on page 31.

Consequently, these are a *very* special case of NAMED-LAMBDA calls (since LETs in MIT SCHEME desugar into manifest NAMED-LAMBDA β-redexes [Church 51]). Specifically, by careful design, each of the ***<lerk>_i*** binding forms will be referenced by their associated ***<identifier>_i*** name *once and only once* within the extracted LET body, ***<lerk>_body***. Thankfully, this makes LERKS *normal form* LET specialization a simple sub-case of the much more general ***<CALL_NAMED-LAMBDA>*** form detailed later (p. 65).

The resulting specialization rule for LERKS-normal (hence, let-*extraction* spawned) LET forms is thus mercifully straighforward. To wit, each ***<lerk>_i*** is *partial evaluated* and its *specialized code* and *specialized type* associated with its companion ***<identifier>_i*** in an extension to the embedded ***<LOCAL>*** *specialization environment* component of the in-coming *specialization bookkeep*. The general ***<lerk>_body*** is then *partial evaluated* in this extended *specialization bookkeep*, with the resulting *specialized code* and *specialized type* (and possibly updated *bookkeep*) being returned as the results of the overall LERKS-normalized LET form.

Formally:

$$
\begin{array}{ll}
\texttt{Input PE bookkeep:} & \mathtt{B} \\
\texttt{Input source code:} & \mathtt{s} ::= \texttt{(LET ((}\textit{<identifier>}\texttt{\_1 }\textit{<lerk>}\texttt{\_1)} \\
 & \qquad\qquad\quad \texttt{(}\textit{<identifier>}\texttt{\_2 }\textit{<lerk>}\texttt{\_2)} \\
 & \qquad\qquad\quad \vdots\texttt{)} \\
 & \qquad\quad\ \textit{<lerk>}\texttt{\_body)} \\
 & \\
\texttt{Specialized code:} & \mathcal{PE}[\![\mathtt{s}]\!]\,\mathtt{B} \ \triangleq\ \boxed{\mathcal{PE}[\![\textit{<lerk>}\texttt{\_body}]\!]\,\mathtt{B}^*} \\
\texttt{Specialized type:} & \ \ \mathcal{T}[\![\mathtt{s}]\!]\,\mathtt{B} \ \triangleq\ \ \ \mathcal{T}[\![\textit{<lerk>}\texttt{\_body}]\!]\,\mathtt{B}^* \\
 & \\
\qquad\qquad\texttt{where} & \\
 & \mathtt{B}^* \triangleq \big(\mathtt{B}[\textsc{code}][\texttt{<LOCAL>}]\big)\left[\textit{<identifier>\_i} \mapsto \boxed{\mathcal{PE}[\![\textit{<lerk>\_i}]\!]\,\mathtt{B}}\right]_{i=1}^{n} \\
 & \mathtt{B}^* \triangleq \big(\mathtt{B}[\textsc{type}][\texttt{<LOCAL>}]\big)\left[\textit{<identifier>\_i} \mapsto \ \ \mathcal{T}[\![\textit{<lerk>\_i}]\!]\,\mathtt{B}\ \right]_{i=1}^{n} \\
 & \\
\texttt{Output bookkeep:} & \mathtt{B}' \longleftarrow \boxed{\mathcal{PE}[\![\textit{<lerk>\_body}]\!]\,\mathtt{B}^*}
\end{array}
$$

This exploits the fact that let-*extracted identifier*s always appear *exactly once* within the body of the generated LET form, without the possibility of naming collisions, as anticipated earlier

(§ 0.2.2, p. 26). This, for example, obviates the need for an *occurrence counting analysis* [Bondorf & Danvy 90/91] to avoid code duplication and/or unintended elimination of non-dead code, since: a) *no* duplication can result from re-nesting the de-nested code back into their original program points, and b) *no* inadvertant elimination can occur given that each `let`-*extracted identifier* is guaranteed, by construction, to occur in the body.

This is the long-promised consequence of the insistence that `let`-*extraction* be trivially reversible (p. 26). In particular, no common subexpression coalescence is performed during `let`-*extraction* normalization, if only to make this argument transparent (and to make the LERKS-normalized result executable without semantic alteration).

In sum, one can characterize the *code specialization* and companion *specialization type inference* computation sub-tasks for `LET` forms as a simple forward substitution of the specialized bound forms back into their original program points, before `let`-*extraction* hoisted them out. The validity of this trick, however, hinges on the crucial invariant that `let`-*extraction* be reversible in exactly this way. This is why user-introduced `LET` forms are separated out as manifest `NAMED-LAMBDA` combination/application/invocation forms: for them, this simple rule cannot apply in general.

But, just when you thought it was safe to go back in the water. . . [42]

"The slings and arrows of outrageous code"[43]

One subtle consideration does arise, however: potential run-time side effects within the code being specialized force the *partial evaluator* to impose an ordering within the *specialized code* it generates. This is necessary so that the order presumed at code specialization time corresponds to the order actually executed at run time. Otherwise, the analytic assumptions won't match the emperical realization.

This exposes an underspecification in the SCHEME language's formal semantics.

Note that the order of evaluation of `LET` bindings is not specified in standard SCHEME [IEEE 91], nor is the order of evaluation of the constituents of the combination/application/invocation forms from which they were `let`-*extracted* (courtesy of LERKS normalization). Consequently, it is poor practice for any of the `<lerk>_i` expressions to incur any side effects at run time. This implies that they likewise should never incur *bookkeep* alterations at *partial evaluation* time. This behavior is readily verified by comparing the before-and-after *bookkeeps* upon *partial evaluating* each of the `<lerk>_i` bound forms. Note carefully, though, that this is not a problem exacerbated by `let`-*extraction*: it is already inherent in the nested subexpressions of the original source code.

That said, a properly semantics-preserving specialization must at least produce code that correctly corresponds to *some* sequential evaluation of the extracted forms at run time. Specializing them all in the same initial *bookkeep*, assuming no cross-talk, could result in specializations that violate this covenant. To that end, any bound forms found to potentially alter the *bookkeep* are queued to be re-specialized after all the other sibling forms have been specialized,

[42] Advertising poster tag line from the movie *Jaws 2* [Gottlieb, Sackler & Tristan 78].
[43] A corruption of [Shakespeare 1603, Hamlet, Act III, Scene 1, Line 58].

all in the same initial in-coming *bookkeep*. Those miscreant forms that have been queued for re-specialization are then re-processed using a "threaded" *bookkeep* passed from one through the next, analogous to the run-time processing of sequential expressions in a `BEGIN` form.[44]

Worse, no particular ordering chosen for the code analysis at *partial evaluation* time can be enforced in the resulting *specialized code* when it is later executed at run time without re-writing the source code to impose a corresponding order (say, via injection of appropriately nested `LET` forms, since these are guaranteed to be executed outermost first). Worse still, it is not even decidable, in general, which forms will have side effects since not all procedure variables' bindings may be known. Even at run time, they may be dynamically re-defined, *etc.* The sad reality, therefore, is that any form with a free or dynamic variable in operator position must be assumed to potentially mutate the *bookkeep*, and thus must be serialized in some fixed order in the generated *specialized code*. Otherwise, the program analysis on which the *partial evaluation* was based might not correspond to the subsequent actual order of execution of the *specialized code* at run time.

"Or to take arms against a sea of side-effects"[45]

Therefore, the current DESCARTES prototype gleefully ignores this complication. Source code that relies on the unspecified order of evaluation cannot be supported in general anyway. Thus, this particular complication is *not* reflected in the formal semantics shown. Addressing this shortcoming in a satisfactory fashion is left as a challenge for future work. Meanwhile, DESCARTES imposes an arbitrary order at random rather than attempt "to take arms against a sea of troubles" [Shakespeare 1603, Hamlet, Act III, Scene 1, Line 59].

This position seems fair given that this unspecified order of evaluation is viewed by many in the SCHEME community as a blemish on the otherwise-clean formal semantic specification of the language. « *C'est la guerre.* »

[44] Of course, the first such ill-behaved form need not be re-processed, so long as its resulting altered *bookkeep* is remembered to initiate the threading of the rest, if any, or just returned as the final *bookkeep*, if not. Thus, only those subsequent additional delinquents that occur, if any, need be re-specialized in threaded fashion (starting with the updated *bookkeep* from the first miscreant) to preserve sequential semantics. This is a peephole optimization, but one that always avoids at least some re-specialization effort. The alternative is to always *bookkeep* thread all sub-form specializations, but that would necessitate always imposing a specific order on the run-time execution of the resulting specialized code, which one should avoid whenever possible (for example, to permit better resource management later, like register allocation during compilation of the specialized code).

[45] A corruption of [Shakespeare 1603, Hamlet, Act III, Scene 1, Line 59].

0.3.9 Rule No. 6: There is *no* Rule 6![46] . . . only *Residualization*.

```
--------------------------------------------------------------------------------
          This rule intentionally left very nearly blank.
--------------------------------------------------------------------------------
```

This is a good intermission point to mention *residualization*, as contrasted with *specialization*.

The Gist of *Residualization*

Up to this point, for example, all *specialization* rules have iteratively descended their component subexpressions, substituting their resulting *partial evaluated* "residues" into copies of the original form in a straightforward, bottom-up substitution. The non-recursive nature of the rules so far has made it easy to argue that this process always terminates, generating a finite *specialized code* and *specialized type* (and updated *bookkeep*). That happy circumstance, however, is about to change.

The preceding `LET` rule was the first to introduce a fully recursive call to the *partial evaluator*, by virtue of its general `<lerk>` sub-forms. There, however, one can easily argue that this process terminates since, by construction, the `let`-*extracted* forms that are being specialized and forward-substituted were simply hoisted out of the original (finite) source code. So long as only these rules presented so far are the only rules in play, this termination argument is sound.

With the remaining rules, however, things are about to get far more intricate. Specifically, the remaining rules have the potential to expand sub-forms into their specialized equivalents stored in the *bookkeep*, then recursively descend those expanded forms *ad infinitum*. In order to bound this process, therefore, one must decide where and how to terminate the recursion. Where to terminate is a very important issue— one that is addressed in the subsequent rules and whose resolution is a fundamental result and original contribution of this work. Once decided *where* to terminate, however, *how* to terminate the recursion, on the other hand, is fairly simple. It's called *residualization* and it's the topic of this section.

Residualization as Über Substitution

Simply put, *residualization* is a glorified substitution operator: it stipulates to the *program specializer* that it *not* pursue any deeper the expansion/unfolding of the remaining sub-forms of the *partial evaluation* recursive descent, but that it *do* forward substitute the accumulated bindings of the locally bound variables in the *specialization bookkeep* for any free occurrences in the remaining program source (to avoid lexical escape). That sounds like a lot, but the idea is simple and easily illustrated with a brief examples.

A Simple *Residualization* Example

Consider the following random "coin flip fairness assessor" program. Conceptually, it flips a presumably-fair 23-sided coin repeatedly until it comes up "heads", then it returns how many flips it took to get there, along with the odd/even parity of the number of trials (just for kicks).

[46] From the *Bruces* sketch [Python 70,73,99].

```
--------------------------------------------------------------------------------
(define (coin-faerie coin-sides flip-count p np)
  (if (zero? (random coin-sides))                     ;;; (random N) ⇒ n ∈ [0, N − 1]
      (coin-fairness-assessment flip-count p np)
      (coin-faerie coin-sides (1+ flip-count) np p))) ;;; N.B.: p ↔ np

(define (coin-fairness-assessment flip-count p np)
  (quasiquote (,flip-count = ,p parity (not ,np))))

(define (coin-trial)
        (coin-faerie 23 1 'odd 'even))

]=> (coin-trial)
;Value: (16 = even parity (not odd))
;Value: ( 4 = even parity (not odd))
;Value: ( 9 = odd  parity (not even))
;Value: (52 = even parity (not odd))
;Value: (20 = even parity (not odd))
--------------------------------------------------------------------------------
```

Suppressing the LERKS normal form details, this might *partial evaluate* into the following, after being unfolded twice and then residualized:

```
--------------------------------------------------------------------------------
(named-lambda (coin-faerie coin-sides flip-count p np)
  (if (zero? (random coin-sides))
      (coin-fairness-assessment flip-count p np)
      ;; ---------
      ;; Unfold: (coin-faerie coin-sides (1+ flip-count) np p)
      ;; ---------
      (let ((flip-count_1 (1+ flip-count))
            (         p_1     np)
            (        np_1      p))
        (if (zero? (random coin-sides))
            (coin-fairness-assessment flip-count_1 p_1 np_1)
            ;; ---------
            ;; Unfold: (coin-faerie coin-sides (1+ flip-count) np p)
            ;; ---------
            (let ((flip-count_2 (1+ flip-count_1))
                  (         p_2            np_1 )
                  (        np_2             p_1 ))
              (if (zero? (random coin-sides))
                  (coin-fairness-assessment flip-count_2 p_2 np_2)
                  ;; ---------------
                  ;; Residualize: (coin-faerie coin-sides (1+ flip-count) np p)
                  ;; ---------------
                  (coin-faerie coin-sides (1+ flip-count_2) np_2 p_2)))))))
--------------------------------------------------------------------------------
```

The interesting thing to note is how the source code at each recursive call is unfolded and residualized.

Specifically, the first parameter (`coin-sides`) is always trivial and intransient so it needn't be wrapped via `let`-*insertion*, whereas the remaining parameters (`flip-count`, `p` and `np`) are recursively passed non-trivial and/or transient arguments, so they are `let`-*insertion* wrapped.

Lexical Disambiguation upon Unfold `Let`-*Insertion*

Moreover, note carefully how a fresh name is generated at each new lexical *unfold* level— hence the "_1" and "_2" suffixes— to avoid inadvertent *capture* upon unfolding. These fresh names must be forward substituted into the *residualized* call as well, since simply returning the source code to be residualized— *viz.*, `(coin-faerie coin-sides (1+ flip-count) np n)`— without substitution would result in inadvertent *escape* to the outermost lexical binding.

To be very clear: were the specializer simply to `let`-*insertion*-wrap these without subscripting or otherwise disambiguating the bindings, it could produce ill-formed gibberish like: `(let ((p np)(np p)) ...)`. It is for this reason that the *residualization* process must be careful to propagate local renamings into the residualized source code, not simply embed the residual source directly as is.

That's All for Now... `<CALL>` Again Later

Other than that, *residualization* leaves matters much as they lie. We shall see in the ***<CALL>*** rule (Rule No. 8, § 0.3.11, p. 59) that there is a bit more flexibility in how to *residualize* calls (such as operator name specialization in lieu of unfolding/open coding), but this is enough detail for now. The key issue is one of lexical name indexing upon unfold `let`-*insertion* and the requisite subsequent forward name substitution even into residualized code, as the above example dramatized.

***Residualization*: Section Summary**

To summarize, therefore: modulo forward substitution, one might characterize *residualization* as the bottom-up complement to the top-down, recursive descent of the *specialization* process. By contrast with *specialization*, the *residualization* process, by design, *always* terminates, returning a finite resulting *residualized code* fragment and *residualized type*.[47]

"Number 8: The ear."

[47] Assuming, of course, that the source code (and its inferred type) being *residualized* are themselves finite to begin with, which they are.

0.3.10 Rule No. 7: `NAMED-LAMBDA` Forms

```
--------------------------------------------------------------------------------
        (NAMED-LAMBDA (<name> <formals>) <lerk>)  : NB:  MIT SCHEME-specific
--------------------------------------------------------------------------------
 Note:  <formals> can be null, dotted, ``#!rest'' and/or ``#!optional'' params.
```

This rule covers only the case where a `NAMED-LAMBDA` form appears in non-operator position—that is, it is passed as an argument or returned as a result, *not* directly applied to arguments. For the application case, refer to the **`<CALL_NAMED-LAMBDA>`** rule instead (Rule No. 8, p. 65).

To *partial evaluate* `NAMED-LAMBDA` forms in non-operator position, the **`<formals>`** are bound in the *bookkeep*'s **`<PARAMETER>`** binding environment to "dummy" arguments (*e.g.,* to themselves with type UNKNOWN (a.k.a. $\bot_{\mathcal{V}}$)). This is done so that residualizing them has no effect on the resulting *specialized code*.[48] The **`<lerk>`** body is then *partial evaluated* in this extended *bookkeep* environment.

The resulting *partial evaluate*d `NAMED-LAMBDA` form overall is thereafter generated by substituting the body's *specialized code* into a new `NAMED-LAMBDA` form. The *specialized type* of the `NAMED-LAMBDA` form is just that of the *partial evaluate*d body, wrapped to mark it as a (partial) type. These partial types are not currently used so no more will be said about them here: they are for debugging purposes only (for now). Nevertheless, a concrete instance appears below.

Formally:

$$
\begin{array}{lll}
\texttt{Input PE bookkeep:} & \texttt{B} & \\
\texttt{Input source code:} & \texttt{s} ::= & \texttt{(NAMED-LAMBDA (}\textit{<name>}\ \textit{<formals>}\texttt{)}\ \textit{<lerk>}\texttt{)} \\
 & & \\
\texttt{Specialized code:} & \mathcal{PE}[\![\texttt{s}]\!]\,\texttt{B} & \triangleq\ [\![\texttt{(NAMED-LAMBDA (}\textit{<name>}\ \textit{<formals>}\texttt{)} \\
 & & \qquad \boxed{\mathcal{PE}[\![\textit{<lerk>}]\!]\,\texttt{B}^*}\texttt{)}]\!]\uparrow\mathcal{S} \\
\texttt{Specialized type:} & \mathcal{T}[\![\texttt{s}]\!]\,\texttt{B} & \triangleq\ \texttt{(<PROCEDURE>}^{49}\ \texttt{(}\textit{<name>}\ \textit{<formals>}\texttt{)}\ \mathcal{T}[\![\textit{<lerk>}]\!]\,\texttt{B}^*\,\texttt{)} \\
 & & \\
\qquad\qquad\texttt{where} & & \\
 & \texttt{B}^* & \triangleq\ \big(\texttt{B[}\textsc{code}\texttt{][<PARAMETER>]}\big)\,[\textit{<formals>} \mapsto^* \prod_{\textsc{param}} \textit{<formals>}] \\
 & \texttt{B}^* & \triangleq\ \big(\texttt{B[}\textsc{type}\texttt{][<PARAMETER>]}\big)\,[\textit{<formals>} \mapsto^* \texttt{UNKNOWN}] \\
\qquad\qquad\quad\texttt{for} & & \\
 & \prod_{\textsc{param}} \textit{<formals>} & \triangleq\ \{\ [\![\texttt{(<PARAMETER>}\ f\texttt{)}\,]\!]\uparrow\mathcal{S}\ \mid f \in \textit{<formals>}\ \} \\
 & & \\
\texttt{Output bookkeep:} & \texttt{B} & (\textit{not}\ \texttt{B}^*\texttt{, of course)}
\end{array}
$$

[48] An alternative strategy would be to leave the formal parameters unbound in the *specialization environment* then complicate the *`<tref>`* rule to check each reference and, if unbound, treat it as a "dummy" self-reference of type UNKNOWN. This approach, however, ignores further complications should the parameter be shadowing a lexically enclosing local or a global variable of the same name. That too can be overcome, *e.g.,* by systematic internal renaming of *all* `NAMED-LAMBDA` parameters to avoid shadow/capture naming collisions [de Bruijn 72,95], but that piles hack atop of kludge. I therefore prefer instead to address this directly here in the `NAMED-LAMBDA` rule, at each parameter's point of introduction, rather than defer to the *identifier* (de-)reference rule, at their point of use.

[49] Although presently merely debugging/tracing scaffolding, an interesting area for future work would be to

The trick here is in binding each `NAMED-LAMBDA`-bound ***<formal>*** to a special *abstract binding* rather than directly to any specific *specialized code.* This is just a one-level indirection mechanism so that downstream lookups (*viz.,* ***<tref>*** variable references) don't directly in-line substitute the abstract bindings (but they will still have access to these bindings "behind the curtain"). In effect, this is tantamount to implicitly/in-flight *inserting identity **LET**-expressions in the source code* [Bondorf & Danvy 90/91], as mentioned earlier (§ 14, p. 20).

De Facto Identity `Let`-*Insertion* in Non-Operator `NAMED-LAMBDA` Forms

To review that idea, this is the familiar device well-known among macro writers where one captures macro arguments in a trivial `LET` wrapper immediately inside the body of the macro form, like:

```
(define-macro (square expr) (let ((a expr)) (* a a)))
```

This explicit `let`-*insertion* on call entry, in effect, generates a one-step indirection between the formal parameter reference in the body and the variable to which its evaluated value is bound at run time. For DESCARTES' *partial evaluator*, this entry-point identity `let`-*insertion* is implicitly effected by binding `NAMED-LAMBDA` formal parameters to two-level, indirect "abstract bindings" at *partial evaluation* time rather than re-writing the *specialized code* body to have the same effect.

Obviously, LERKS normalization could explicitly insert these `LET`s too, but that decouples the important constraint that the specializer treat local bindings specially, which seems a brittle design choice to my taste. The best way to enforce this "abstract binding" invariant is to encode it in the specialization rule where it is vital that it be respected.

As we shall see in the ***<CALL_NAMED-LAMBDA>*** rule, this will couple the ***<CALL>*** specialization rule(s) to the otherwise trivial ***<tref>*** variable (de)reference specialization rule. Coupling complementary specialization rules to one another rather than coupling special rules to input invariants seems a more robust choice, pragmatically, and a more grounded choice from the standpoint of trying to formally prove the correctness of the specializer (*e.g.,* by transfinite induction on the specialization rules).

Of course, LERKS normalization comes close to just this sort of brittleness, but there the idea was to establish a uniform input invariance that *all* specialization rules could appreciate. Notice, for example, how *every* non-trivial specialization rule's description somewhere contains the phrase, "Thankfully, LERKS normalization ensures that the *mumble* constituent is trivial" or words to that effect. I wasn't just being cute in repeating that template throughout. It's important. In fact, it's essential.

---

replace these ***<PROCEDURE>*** type expressions with proper *procedures.* That is, given the type expressions for their formal parameters (including abstract types UNKNOWN (a.k.a. $\perp_{\mathcal{V}}$) and ANY (a.k.a. $\top_{\mathcal{V}}$)) and a *type environment* mapping free variables to their types, these type-generating procedures would produce an appropriate type expression for the `NAMED-LAMBDA` form's return type in the given *type environment.* This would further the *categorical isomorphism* between types and values, as alluded to earlier (p. 33), where the types of pairs are represented as pairs of types, the types of vectors as vectors of types, and so on. Here, the types of procedures would become procedures that compute types. This, for instance, naturally suggests the implementation of the type inference operator, $\mathcal{T}$, as a straightforward, full and proper *abstract interpreter* of source-level expressions rather than as an *ad hoc* `fulltype` procedure, as currently implemented and presented.

Specialization of `NAMED-LAMBDA` Forms: Section Summary

In sum, one can characterize the *code specialization* and companion *specialization type inference* computation sub-tasks for `NAMED-LAMBDA` forms in non-operator position as fairly sedate. In effect, the body of the `NAMED-LAMBDA` is specialized to whatever degree possible given the bindings of its free and global variables in the *specialization context* within the *specialization bookkeep*, but with the `NAMED-LAMBDA`'s formal parameters themselves being bound, effectively, to no-op placeholders. This allows the specializer to aggressively penetrate into the `NAMED-LAMBDA` body while, in effect, treating the unbound formal parameters as dynamic unknowns.

The next rule below (*viz.,* `<CALL>`) clarifies how this situation differs for `NAMED-LAMBDA` forms that *are* in operator position in a combination/application/invocation. That's when things really start to get exciting!

0.3.11 Rule No. 8: <CALL> Forms

```
-------------------------------------------------------------------------------
         (<CALL> <triv> <triv>*)        : Combination/application/invocation
-------------------------------------------------------------------------------
```

The above rule is a half-truth. The actual detailed full Rufian-ized rule is:

```
-------------------------------------------------------------------------------
        (<CALL_LOCAL>       (<LOCAL>     <identifier>) <triv>*)
        (<CALL_GLOBAL>      (<GLOBAL>    <identifier>) <triv>*)
        (<CALL_PARAMETER>   (<PARAMETER> <identifier>) <triv>*)
        (<CALL_PRIMITIVE>   (<PRIMITIVE> <identifier>) <triv>*)

        (<CALL_NAMED-LAMBDA> (NAMED-LAMBDA ...stuff...) <triv>*)
-------------------------------------------------------------------------------

 Note:  All capitalized tags here are literal symbol constants,
         not grammatical meta-variables.

-------------------------------------------------------------------------------
```

Thankfully, courtesy of LERKS *normal form*, each constituent of a ***<CALL>*** form is at least guaranteed to be trivial, with the single modest exception of the **NAMED-LAMBDA** form. That is, except for **NAMED-LAMBDA** calls, the operator is always a trivial variable reference whose class tag matches the class tag of its ***<CALL_tag>*** wrapper. Each argument expression is likewise trivial (of the slightly more general class ***<triv>***). The manifest **NAMED-LAMBDA** calls are almost as simple, the only asymmetry with respect to the other call forms being that the operator is an explicit (manifest) **NAMED-LAMBDA** form, not a trivial variable reference.

Finally, things start to get interesting. But not *too* interesting. This is the first specialization rule in which the *partial evaluator* has a choice in how to proceed: either continue to recursively *specialize* or terminate the recursion and *residualize* instead.

Decision point: the first step is to decide whether: a) to *reduce* this call by open coding (unfolding) the *operator*'s body in line, or: b) to *specialize* the *operator* out of line (either by name or by value) but leave the ***<CALL>*** form otherwise *residualized*, or: c) to just directly *residualize* the constituents without generating any new specialized operator variants. By the way, since this is the first rule where *residualization* arises as an alternative to further (recursive) sub-component *specialization*, it's convenient to have addressed that idea directly in the earlier paraleiptic "Intermission" rule aside[50] (Rule No. 6, § 0.3.9, p. 52).

That said, ***<CALL>*** form specialization proceeds as follows. In broad strokes, if the current *in situ* statistical *relevance* weight is sufficiently high, we choose to open code/in-line the call (assuming the operator's value is known). Otherwise, if there are existing specializations for the operator that match the current invocation signature, we can still partially-specialize the

[50] "Though this be madness, yet there is method in't." [Shakespeare 1603, Polonius, Act II, Scene 2, Line 206].

invocation by at least directing it to the specialized operator out of line. If even that fails, we can still remember the missing case and arrange to dynamically back-patch the call to the appropriate specialized variant should it ever be generated downstream. Finally, if the *relevance* weight for the call is sufficiently low, or if the operator's value is not known, we simply *residualize* the call form, generating a fresh call form from the specialized constituents.

Of these choices, by far the most interesting is when we choose to defer generating a fresh variant specialization of the operator, arranging instead to adaptively back-patch an out-of-line call stub should the desired variant ever be generated in the future. To wit, we generate a stub ("trampoline") that initially just relays directly to the generic, unspecialized operator. We *register* this stub in the *bookkeep*, marking it with the variant it hopes to one day encounter. If/ when the desired specialization variant is ever generated, this registered stub will be updated (in place) to jump directly to the target variant. This is an instance of *dynamic call path* swizzling, akin to the *execution cache* internal to MIT Scheme [Miller & Rozas 91] [Adams & Hanson 93, § 4.3 Efficiency Tips]. We'll return to this in due course but, for now, let's focus instead on the less exotic cases.

Specialization of `<CALL>` Forms: Each in Turn

In light of this general plan, consider now each distinct class of ***<CALL>*** form in turn, from simplest to most intricate operator: spec., from ***<PARAMETER>*** to ***<LOCAL>*** to ***<GLOBAL>*** to ***<PRIMITIVE>*** to ***NAMED-LAMBDA***.

Parameter Calls. Parameter calls are essentially always *residualize*d since their dynamic operator values cannot be resolved at *partial evaluation* time. For example, consider:

```
------------------------------------------------------------------------------------
     (DEFINE (two-me f) (f 2))
⟶ℒ
     (DEFINE          two-me
       (NAMED-LAMBDA (two-me f) (<CALL_PARAMETER> (<PARAMETER> f) (<CONSTANT> 2))))
------------------------------------------------------------------------------------
```

Note that this case refers only to those formal parameters that are not bound to a concrete value, hence whose abstract bindings are tagged as being of class ***<PARAMETER>*** (as per the preceding **NAMED-LAMBDA** rule, § 0.3.10, p. 56). When the parameter operator *has* been bound to a concrete value, the call will ultimately reduce to one of the other call forms.

Thus, the initial ***<CALL_tag>*** is used only as a seed to initiate the call specialization. If, for example, a **<CALL_PARAMETER>**'s operator first specializes to, say, a **<PRIMITIVE>**, then the **<CALL_PARAMETER>** specialization will immediately defer to **<CALL_PRIMITIVE>** to proceed. Only in instances where the specialized operator persists as an (unbound) **<PARAMETER>** does the current case prevail, at which time the call is simply *residualized*.

In sum, then, a *parameter call* form is either *residualized* (in the unbound case) or else it reduces to one of the other call forms, depending on the value to which the operator ultimately resolves. Termination is therefore guaranteed either by virtue of *residualization* or by induction over the other call forms.

Local Calls. Local call operators are always bound to an associated *specialized code* and *specialized type*. The operator's *specialized type* is ignored (for now) but the *specialized code* will, eventually, always resolve either: a) to one of the other operator forms (including, possibly, an unbound parameter), or: b) to a residualized call form (as a consequence of operator nesting).[51] The former case is a simple deflection while the latter terminates the *specialization* recursion.

In sum, then, a *local call* form reduces either to a simple case of dereference then iterate or to a trivial case of *residualization*. Therefore, no more need be said of them as they are merely a "stutter step" intermediate *en route* to one of the other call forms or else a trivial *residualization* instance. Termination is guaranteed in each case, either by induction or by fiat, respectively.

Global Calls. Global call operators likewise always eventually resolve to an associated *specialized code* and *specialized type*. Unlike the preceding cases, however, they never resolve to an unbound parameter. Therefore, eventually they always resolve to a ***<PRIMITIVE>*** or manifest ***NAMED-LAMBDA*** operator, never a ***<LOCAL>*** or (unbound) ***<PARAMETER>*** operator nor a (higher-order) *residualized* ***<CALL_PARAMETER>*** operator.

The one special *proviso* for *global call* form specializations, however, is that any generated *specialized code* for the call always first check the run-time validity of the specialization-time global binding for the operator before dispatching to any generated specialized variant. For instance, one might define global `add` to be an alias for an instrumented version of primitive `+`, run a test suite to generate various specializations and profiling data, then re-assign `add` to directly alias primitive `+` for subsequent execution. Any specialized calls to `add` must be invalidated accordingly.

Therefore, at specialization time, the operator's ***<PRIMITIVE>*** or manifest ***NAMED-LAMBDA*** ultimately associated with the global identifier is used to generate a specialization dispatch "trampoline" to redirect the call to an appropriate variant handler (as we'll see below). At run time, however, the actual dynamic binding of the global identifier may have changed from what it was at specialization time, when this dispatcher was generated. If it has, the dispatch target is invalid.

This issue is easily resolved, nonetheless, by caching the binding of the global identifier that is discovered at specialization time then testing, at run time, that the global operator is still bound to that same value. If not, the specialization dispatch "trampoline" is invalidated (discarded) and the original non-specialized global call form is invoked instead.[52]

In sum, then, a *global call* form ultimately reduces to a ***<PRIMITIVE>*** call or to a manifest ***NAMED-LAMBDA*** call. Modulo this *en passant* caching consideration. no more need be said for

[51] As in `(lambda (finagle) (lambda (f g) (lambda (x) ((finagle f g) x))))`, where the `(finagle f g)` operator applied to `x` is *residualized* because it's operator, `finagle`, is an unbound formal parameter. Higher-order code like this arises, for example, when abstracting out procedural patterns like `compose` or `derivative-wrt` in place of `finagle`.

[52] Pragmatic Point: Note that these cached global values can be weakly held so that, if reassigned and the old value is no longer referenced anywhere else in the run-time heap, they can be garbage collected. The cache validity test would then simply test the run-time binding against the "released weak value" sentinel (*viz.,* `#f`), which will always test negative (since `#f` is not even a procedure). In this way, the system avoids bloating the heap by unnecessarily retaining pointers to discarded, re-assigned, stale, former operator bindings.

this case. Termination is assured by reduction to other cases.

Primitive Calls. All preceding cases either *residualize* the call form or directly deflect/ defer to one of the other forms.[53] Not so for *primitive call* forms. Here the specialization process finally "bottoms out in the bits" by reaching a natural terminus for semantic reduction. Primitive procedures (and base data) therefore are the foundation on which the entire system is build, along with the small number of kernel syntactic forms.[54]

In broad strokes, a primitive procedure call must be residualized: 1) if any of the arguments to the primitive procedure call are not statically known (*i.e.,* do not *partial evaluate* to literal constants), or 2) if the primitive incurs observable side effects when invoked, or 3) if the primitive invocation might not halt.[55] Still, if the primitive procedure operator can be narrowed given the inferred *specialized types* of the arguments, some partial specialization may nonetheless be possible by replacing the original primitive operator with a more-specific (perhaps machine-dependent) variant. Finally, if all arguments to a primitive procedure call *partial evaluate* to constants, and if the primitive procedure is known to always terminate without observable side effects, then the call can be fully reduced to a constant using the δ-reduction rule of the λ-calculus [Church 51]. That is, assuming the *statistical relevance* of the call site justifies specialization in the first place; if not, *operator partial specialization* or *residualization* is done.

[53] That, while possibly accumulating a bit of state in the recursion, such as a trail of global identifier operator bindings to be cache-validated at run time in the generated specialized code, or possibly other useful contextual tidbits for profiling or debugging of the specializer itself.

[54] One might, in fact, characterize the primitive procedures, base data and core syntactic forms as collectively constituting a veritable *Foundation Trilogy* from which the system *in toto* is derived. That is, if one were so inclined [Asimov 51/52/53,74/82,83/86/90/92].

[55] This first restriction is a correctness condition known as *congruence*. The second precondition is a correctness stipulation called *referential transparency*. The third and final restriction is a totality restriction against potential *divergence*. The remainder of this footnote informally defines these terms in context.

Informally. and to first approximation, *congruence* in the context of *partial evaluation* means that dynamic/ unknown constituents cannot give rise to static/known residues [Jones, Gomard & Sestoft 93]. This means, for example, that call forms with dynamic/unknown constituents must be *residualized*, not unfolded/open coded. Type-driven specialization relaxes this to a restriction against dynamic/unknown *types* (as abstract values). A further relaxation comes from *uniform congruence* [Launchbury 91], which may allow unfolding across the arms of a conditional branch despite a dynamic predicate (as we shall contemplate in the `IF` rule (§ 0.3.12, p. 70)).

The term *referential transparency* [Whitehead & Russell 10/12/13,25/27,62] means that evaluating the same expression more than once always produces exactly the same result, including side effects on the run-time store a dereferences of free variables. If these side effects (including dereferences) are always precisely the same, it is therefore not observable that any side effect has in fact occurred. This is not to say, for example, that it always reference or increment the same variables; rather, it is a much more stringent restriction that it not depend on the value of free variables for its result and that it only assign variables to their pre-existing values. Otherwise, an external observer would be able to detect that one, not two, separate invocations has occurred. A slightly more general term from abstract algebra that captures the same idea (and may be more familiar to some readers) is *idempotence*. The point is that only *referentially transparent/idempotent* call forms can be invoked at specialization time. It is also true that only such forms can be safely duplicated in the specialized code, but that's a different matter altogether (and one that was addressed earlier (p. 49)).

Finally, restriction against potential *divergence* in this context ensures the proper termination of all specialization-time invocations of primitive procedures [Nielson 87,88]. Examples of primitive procedures that might not terminate include the ever-problematic `EVAL` primitive and `length` (when invoked on a cyclic linked list, such as a ring) and other similar degenerate cases of divergent primitives.

Equipped with this broad overview of *primitive call* form specialization, we are now ready to consider each subcase in turn: i) *residualization*; ii) out-of-line, polyvariant *operator partial specialization*; and iii) in-line, polyvariant *primitive call* full *specialization*.

Primitive Call: Residualization. This is trivial. The specialized constituents are reassembled into a fresh <CALL_PRIMITIVE> form to produce the *specialized code*. The inferred *specialized type* is the result of performing type inference over the primitive operator given the *specialized type* of each argument. This, for example, can be done directly or by table lookup.

For instance, primitive `cons` has an observable allocation side effect on the store when executed, so calls to it must be residualized. The type of a `cons` call is a pair of the types of its constituent arguments. Simple.

Primitive Call: Operator Partial Specialization. Alternatively, if the primitive procedure operator can be narrowed given the inferred *specialized types* of the arguments, some partial specialization may nonetheless be possible by replacing the original primitive operator with a more-specific (perhaps machine-dependent) variant, such as replacing `+` with `flonum:+`.

In the simple case, this requires only replacing the source operator with a more narrow variant primitive. For instance, when two `flonums` are added, generic `+` can be specialized to `flo:+`, like:

(+ x y) $\longrightarrow_{\mathcal{PE}}$ (flo:+ x y)   *;;; N.B.: x and y both of type* `flonum`

In the general case, this may require generating an argument-canonicalizing redirection stub (an *en passant* “trampoline”, p. **??**). For example, when a `fixnum` is added to a `flonum`, the `fixnum` must first be coerced to `flonum` before `flo:+` can be applied. Such a code situation might give rise to a specialization variant like:

```
(define (+::fix-and-flo fix:a  flo:b)
   (flo:+ (fix:->flonum fix:a) flo:b))
```

... whence the specializer might generate the following specialization for ‘`x`’ an inferred `fixnum` and ‘`y`’ an inferred `flonum`:

(+ x y) $\longrightarrow_{\mathcal{PE}}$ (+::fix-and-flo x y)   *;;; N.B.: x::*`fixnum` *and y::*`flonum`

In addition to *type-coercion canonicalization*, the specializer’s full *trampoline menagerie* includes such exotic species as *arity-reduction* trampolines (for reducing, say, `(+ x y z)` to `(int:+ (int:+ x y) z)`, *optionals default-ification* trampolines (for handling `#!optional` and/ or `#!rest` “dotted” formal parameter list canonicalization), and a host of other such trivial interface toreadors, many of which are generated by simple pattern based code generation.

More importantly, any such generated “trampoline” stub is registered in the *specialization bookkeep* for potential re-use at future specialization call sites. When *operator partial specialization* is initiated, therefore, the *specializer* first looks for an existing “trampoline” stub with a compatible type signature, generating new ones only as needed.

Primitive Call: Full Specialization. Finally, if all arguments to a primitive procedure call *partial evaluate* to constants, and if the primitive procedure is known to always terminate without observable side effects, then the call can be fully reduced to a constant using the δ-reduction rule of the λ-calculus [Church 51]. That is, assuming the *statistical relevance* of the call site justifies specialization in the first place; if not, *operator partial specialization* or *residualization* is done.

For example:

```
(expt 2.71828182845904523536 (* 2 3.14159265358979323846 0+i))
```

$\longrightarrow_{\mathcal{PE}}$

```
1.0-2.4492127076447545e-16i
```
;;; *From* Euler: $e^{i\pi} + 1 = 0 \therefore e^{2\pi i} \approx 1$

Most SCHEME primitive procedures always halt on concrete inputs; few have side effects.[56]

For those with no observable run-time (dynamic) side effects, we can directly δ-reduce those when all arguments are wholly known (regardless of *relevance weight*). Otherwise, if even a single argument is dynamic (unknown at specialization time), the `<PRIMITIVE>` operator call form must be residualized.[57]

Issues of strictness are not violated by such local optimizations since LERKS normalization guarantees that *all* arguments will be trivial (*i.e.,* `<triv>`). So eliding a primitive call will *not* forgo the evaluation of any argument form: they will have been evaluated in the enclosing `let`-*extraction* wrapper. Note, however, that those primitives that incur observable side-effects at run time (like `vector-set!` *etc.*) must *always* be *residualized*, not directly δ-reduced, although they can be symbolically/abstractly "executed" at partial evaluation time if their result can be statically determined, thereby enabling subsequent opportunities for specialization that might otherwise go undetected.

Primitive Call: The Hidden Hand. That said, there is still the unresolved issue of exactly how to choose whether and how to specialize a given primitive call form based on the specialization context. In particular, the preceding glossed over the case where an argument's inferred *specialized type* is not a simple scalar type by, say, a heterogeneous distribution, like:

`fixnum`	80%
`flonum`	20%

[56] Except for `EVAL` (and, by extension, `LOAD`) and various I/O primitives (such as `READ` and the `port` operations and the like), most primitive procedures always halt. To be precise, call those which *do* always halt *determinate* primitives. Those which might not halt are then dubbed *indeterminate*. The *determinate* primitives may always be safely invoked at *partial evaluation* time. The *indeterminate* primitives might not be (depending on their arguments).

[57] That is, except for various peep-hole optimizations, like not bothering to residualize a call to, say, `car` if the argument is provably *known* to be, say, a pair composed of a static (known) value and a dynamic (unknown) `cdr`. Though not directly δ-reducible in the conventional sense, such calls can be abstractly eliminated by knowing that, say, `car` applied to the specialized value `(x . unknown)` is equivalent to `x`. This is a well-established technique, called *projection factorization* [Launchbury 91].

[To Do: Here we review the detailed `fib-x` example from the results chapter, including the "primitives matrix" pattern-driven specialization lookup and its commensurate type inference issues... like fiducial inference. For example, see figures 0-10, 0-11 and 0-12 below.]

```
.---.---.---.---.---.---.---.---.---.---.---.---.---.---.---.---.---.---.---.---.
 \   \   \   \ s \ t \   \   \   \   \   \   \   \   \   \   \ t \ s \   \   \   \
  \   \   \   \   \   \   \   \   \   \   \   \   \   \   \   \   \   \   \   \   \
   \ f \ b \ f \ f \ f \ f \ f \ f \ f \ f \ f \ f \ f \ f \ f \ f \ f \ f \ b \ f \
    \ l \ i \ i \ i \ i \ l \ i \ l \ i \ l \ i \ i \ l \ i \ l \ i \ i \ i \ i \ l \
     \ o \ g \ x \ x \ x \ o \ x \ o \ x \ o \ x \ x \ o \ x \ o \ x \ x \ x \ g \ o \
      \ : \ : \ : \ : \ : \ : \ : \ : \ : \ : \ : \ : \ : \ : \ : \ : \ : \ : \ : \ : \
       \ - \ - \ - \ - \ - \ - \ - \ - \ - \   \   \ + \ + \ + \ + \ + \ + \ + \ + \ + \
        \   \   \   \   \   \ 2 \ 2 \ 1 \ 1 \ 0 \ 0 \ 1 \ 1 \ 2 \ 2 \   \   \   \   \
        :---:---:---:---:---:---:---:---:---:---:---:---:---:---:---:---:---:---:---:---:
  flo:- | < | < | < | < | < | < | < | < | < | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
  big:- | < | < | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
  fix:- | < | 0 | < | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
s fix:- | < | 0 | 0 | < | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
t fix:- | < | 0 | 0 | 0 | < | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
  flo:-2 | < | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
  fix:-2 | < | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
  flo:-1 | < | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
  fix:-1 | < | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
  flo: 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
  fix: 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
  fix:+1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | < |
  flo:+1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | < |
  fix:+2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | < |
  flo:+2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | < |
t fix:+ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | < | 1 | 1 | 1 | < |
s fix:+ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | < | 1 | 1 | < |
  fix:+ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | < | 1 | < |
  big:+ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | < | < |
  flo:+ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | < | < | < | < | < | < | < | < | < |
        `---^---^---^---^---^---^---^---^---^---^---^---^---^---^---^---^---^---^---^---'
```

Figure 0-10: Primitive Arithmetic Type Matrix: Dyadic <

Manifest `NAMED-LAMBDA` Calls. The final case, manifest `NAMED-LAMBDA` calls, is the *really* interesting one. Here we fold in the global context spectrum for this procedure with the *in situ* inferred *specialized type*s of its arguments. Those, along with the *relevance* weight, determine which, if any, existing variant specializations of the procedure should be used, or which news ones to generate, if any. The resulting `NAMED-LAMBDA` *specialized code* is, in effect, a run-time variant dispatch "trampoline", analogous to the *polyvariant in-line cache*s (PICs) of Ruf [Ruf 93].

```
 .---.---.---.---.---.---.---.---.---.---.---.---.---.
  \   \   \ s \ t \   \   \   \   \   \ t \ s \   \   \
   \   \   \   \   \   \   \   \   \   \   \   \   \   \
    \ b \ f \ f \ f \ f \ f \ f \ f \ f \ f \ f \ f \ b \
     \ i \ i \ i \ i \ i \ i \ i \ i \ i \ i \ i \ i \ i \
      \ g \ x \ x \ x \ x \ x \ x \ x \ x \ x \ x \ x \ g \
       \ : \ : \ : \ : \ : \ : \ : \ : \ : \ : \ : \ : \ : \
        \ - \ - \ - \ - \ - \ - \   \ + \ + \ + \ + \ + \ + \
         \   \   \   \   \ 2 \ 1 \ 0 \ 1 \ 2 \   \   \   \
         :---:---:---:---:---:---:---:---:---:---:---:---:---:
  big:-  | < | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
  fix:-  | 0 | < | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
s fix:-  | 0 | 0 | < | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
t fix:-  | 0 | 0 | 0 | < | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
  fix:-2 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
  fix:-1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
  fix: 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 |
  fix:+1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 |
  fix:+2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
t fix:+  | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | < | 1 | 1 | 1 |
s fix:+  | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | < | 1 | 1 |
  fix:+  | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | < | 1 |
  big:+  | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | < |
         ^---^---^---^---^---^---^---^---^---^---^---^---^---'
```

Figure 0-11: Integer Arithmetic Type Matrix: Dyadic `int:<`

```
  .---.---.---.---.---.---.---.
   \ f \ f \ f \ f \ f \ f \ f \
    \ l \ l \ l \ l \ l \ l \ l \
     \ o \ o \ o \ o \ o \ o \ o \
      \ : \ : \ : \ : \ : \ : \ : \
       \ - \ - \ - \   \ + \ + \ + \
        \   \ 2 \ 1 \ 0 \ 1 \ 2 \   \
        :---:---:---:---:---:---:---:
flo:-   | < | < | < | 1 | 1 | 1 | 1 |
flo:-2  | < | 0 | 1 | 1 | 1 | 1 | 1 |
flo:-1  | < | 0 | 0 | 1 | 1 | 1 | 1 |
flo: 0  | 0 | 0 | 0 | 0 | 1 | 1 | 1 |
flo:+1  | 0 | 0 | 0 | 0 | 0 | 1 | < |
flo:+2  | 0 | 0 | 0 | 0 | 0 | 0 | < |
flo:+   | 0 | 0 | 0 | 0 | < | < | < |
        `---^---^---^---^---^---^---'
```

Figure 0-12: Floating-Point Arithmetic Type Matrix: Dyadic `flo:<`

[To Do: Here we review the detailed `fib-x` example from the results chapter.]

xxx-8: **[To Do: Finish this rule.]**

xxxzzzxxx

Formally:

Input PE bookkeep: B
Input source code: s ::= (<CALL> *<triv>* *<triv>**)

Specialized code: $\mathcal{PE}[\![\mathrm{s}]\!]\,\mathrm{B} \triangleq [\mathit{code}]$
Specialized type: $\mathcal{T}[\![\mathrm{s}]\!]\,\mathrm{B} \triangleq [\mathit{type}]$

[To Do: Ignore the following personal annotation turds. . . .
. (§ 0.2.2, p. 26), obviating the need for an *occurrence counting analysis* [Bondorf & Danvy 90/91]
***Cf.*, footnote (14), p. 20.]**

xxx-8: **[To Do: Finish this rule.]**

Specialization of <CALL> Forms: Section Summary

In sum, one can characterize the *code specialization* and companion *specialization type inference* computation sub-tasks for *<CALL>* forms as:

```
Specialize code:  []
Specialize type:  []
```

0.3.12 Rule No. 9: IF Forms

```
--------------------------------------------------------------------------------
         (IF <triv> <lerk>_t <lerk>_f)  : No ``one-armed'' IFs
--------------------------------------------------------------------------------
```

The IF form is fairly interesting too.

First, the conditional predicate is, thankfully, trivial, courtesy of LERKS *normal form*. Its *specialized type* is consulted to determine which branch(es) to recursively *partial evaluate*. Also, the current *relevance* weight is also consulted, to decide where to continue the *partial evaluation* descent or just revert to *residualization* (which is why Rule No. 6 (Intermission: *residualization*) was a good intermission point to consider that before we reached this point and need to know about it).

More interesting is how this predicate's *specialized type* is bifurcated to guide the subsequent subexpressions' *partial evaluation*(s). xxxzzzxxx

What's more interesting is the propagation of T/F assertions accumulated within the *specialization environment* of the *specialization bookkeep*. This is a weak form of truth maintenance (it's really just forward propagation of simple F/non-F constraints), but it does involve non-trivial "probabilistic conditioning" [Drake 67] along each branch. Still, DESCARTES is not nearly as aggressive in this regard as, say, Futamura's *Generalized Partial Computation* [Futamura & Nogi 88]. Still, generalizing his underlying *truth maintenance system* to operate on probabilistically weighted structural type spectra should prove an interesting area for future work.

Here, again, a quick review of the `fib-x` example from the results section will help motivate and focus the running text.

[58]

xxx-9: **[To Do: Finish this rule.]**

zzz

Formally:

[58] Nomenclature: One might be tempted to use the terms *true* and *false* when speaking of the arms of a two-way branch but this tends to be misleading in the LISP family of languages (of which SCHEME is a member). This is because, in LISP dialects, only the distinguished `false` object (`#f`) is treated as false for purposes of Boolean predication, while *all* other values (*not* just `true` (`#t`) are treated as true. One might thus be tempted to use the terms *false* and *non-false* but that is more cumbersome that *affirmative* and *contradictory*, which have the added benefit of directly addressing the semantic intent while suppressing the underlying programming language idiosyncrasies.

$$
\begin{array}{ll}
\texttt{Input PE bookkeep:} & \texttt{B} \\
\texttt{Input source code:} & \texttt{s} ::= \texttt{(IF <}\mathit{triv}\texttt{>\_test <}\mathit{lerk}\texttt{>\_then <}\mathit{lerk}\texttt{>\_else)}
\end{array}
$$

$$
\begin{array}{lllll}
\texttt{Specialized code:} & \mathcal{PE}[\![\texttt{s}]\!]\,\texttt{B} \triangleq & \text{Let} & \tau_{\text{pred}} = & \mathcal{T}[\![\texttt{<}\mathit{triv}\texttt{>\_test}]\!]\,\texttt{B} \\
 & & \text{in} & & \\
 & & \quad\text{If} & \tau_{\text{pred}} \not\sqsupseteq & \texttt{FALSE} \\
 & & \quad\text{then} & & \\
 & & & \boxed{\mathcal{PE}[\![\texttt{<}\mathit{lerk}\texttt{>\_then}]\!]\,\texttt{B}} & \\
 & & \quad\text{else} & & \\
 & & \quad\text{if} & \tau_{\text{pred}} \equiv & \texttt{FALSE} \\
 & & \quad\text{then} & & \\
 & & & \boxed{\mathcal{PE}[\![\texttt{<}\mathit{lerk}\texttt{>\_else}]\!]\,\texttt{B}} & \\
 & & \quad\text{else} & & \\
 & & \quad[\![\texttt{(IF} & \boxed{\mathcal{PE}[\![\texttt{<}\mathit{triv}\texttt{>\_test}]\!]\,\texttt{B}} & \\
 & & & \boxed{\mathcal{PE}[\![\texttt{<}\mathit{triv}\texttt{>\_then}]\!]\,\texttt{B}^{+}} & \\
 & & & \boxed{\mathcal{PE}[\![\texttt{<}\mathit{triv}\texttt{>\_else}]\!]\,\texttt{B}^{-}}\texttt{)}]\!] \uparrow \mathcal{S} & \\
 & & \quad\text{endif} & & \\
\quad\quad\texttt{where} & & & & \\
 & \texttt{B}^{+} \triangleq xxxzzzxxx & & & \\
 & \texttt{B}^{-} \triangleq xxxzzzxxx & & & \\
\quad\quad\texttt{for} & & & & \\
 & \multicolumn{4}{l}{\prod_{\text{PARAM}} \texttt{<}\mathit{formals}\texttt{>} \triangleq \left\{\ [\![\texttt{(<PARAMETER>}\ f\texttt{)}\,]\!] \uparrow \mathcal{S} \ \middle|\ f \in \texttt{<}\mathit{formals}\texttt{>} \right\}}
\end{array}
$$

$$
\begin{array}{lllll}
\texttt{Specialized type:} & \mathcal{T}[\![\texttt{s}]\!]\,\texttt{B} \triangleq & \text{Let} & \tau_{\text{pred}} = & \mathcal{T}[\![\texttt{<}\mathit{triv}\texttt{>\_test}]\!]\,\texttt{B} \\
 & & \text{in} & & \\
 & & \quad\text{If} & \tau_{\text{pred}} \not\sqsupseteq & \texttt{FALSE} \\
 & & \quad\text{then} & & \\
 & & & \mathcal{T}[\![\texttt{<}\mathit{lerk}\texttt{>\_then}]\!]\,\texttt{B} & \\
 & & \quad\text{else} & & \\
 & & \quad\text{if} & \tau_{\text{pred}} \equiv & \texttt{FALSE} \\
 & & \quad\text{then} & & \\
 & & & \mathcal{T}[\![\texttt{<}\mathit{lerk}\texttt{>\_else}]\!]\,\texttt{B} & \\
 & & \quad\text{else} & & \\
 & & \quad[\![\texttt{(IF} & \mathcal{T}[\![\texttt{<}\mathit{triv}\texttt{>\_test}]\!]\,\texttt{B} & \\
 & & & \mathcal{T}[\![\texttt{<}\mathit{triv}\texttt{>\_then}]\!]\,\texttt{B}^{+} & \\
 & & & \mathcal{T}[\![\texttt{<}\mathit{triv}\texttt{>\_else}]\!]\,\texttt{B}^{-}\texttt{)}]\!] \uparrow \mathcal{S} & \\
 & & \quad\text{endif} & &
\end{array}
$$

$$
\texttt{Output bookkeep:} \quad \texttt{B} \quad (\mathit{not}\ \texttt{B}^{+}\ \texttt{or}\ \texttt{B}^{-}\texttt{, of course) xxxzzzxxx}
$$

[To Do: Note: Apropos *assertion* propagation, mention *uniform congruence* from Launchbury [Launchbury 91]. This also plays into what I call *uniquification*.

To wit, *congruence*, especially Launchbury's *uniform congruence*, & its proximity to what I call *uniquification*, as well as its surface similarity to *probabilistic*

conditioning **of** `IF` **branches. Mention Futamura's *generalized partial computation* as a more general, formal *truth maintenance system* along these same lines, though restricted to that which can be statically proven/deduced at partial evaluation time. Dynamically generated distribution spectra would relax this to that which can be empirically observed as well, which complements statically deduced information in cases where it would have to revert to the more conservative, general case when, perhaps, a hint about which unknowns are more likely to arise than others. *Etc.***

Should define *faithful*, *congruent* and *termination* all on the same page (for convenience of page citation).]

xxx-9: [**To Do: Finish this rule.**]

In sum, one can characterize the *code specialization* and companion *specialization type inference* computation sub-tasks for `IF` forms as:

```
Specialize code:  []
Specialize type:  []
```

0.3.13 Summary of Spectral Specialization Rules for Descartes

[To Do: Finish this chapter by detailing Descartes's "Specialization over Statistical Spectra" as a modest but important generalization of Ruf [Ruf 93], including his technique whereby the partial evaluator generates not just code but also an inferred type for the specialized code generated.

Be sure to define specializer-generated run-time call dispatch stubs as "*polyvariant dispatch trampoline*"s. [59] They guide polyvariant specialized code invocation while ensuring code safety. These are an instance of *en passant* delegate verification at run-time dispatch with a failsafe fallback to the original non-specialized code as needed. These are analogous to Urs Hölzle's method dispatch PICs (*Polyvariant In-line Caches*).

]

Things to mention:

1. DONE: Section "Subexpression De-nesting via `Let`-*Extraction*".

   What I call "fully *de-nested* form" looks equivalent to *monadic normal form* [Hatcliff & Danvy 94], a.k.a. *Administrative Normal Form* (*A-normal form*) of [Flanagan *et al.* 93] [Flanagan *et al.* 2003]. It is a fairly straightforward device, really, so I'm not surprised it was independently discovered by others in the field.

2. TODO: [] Mention somewhere that constant folding, copy propagation, strength reduction, dead code elimination, *etc.* all come for free by leveraging the underlying MIT SCHEME compiler.

3. DONE: Well, I inserted the below text into Rule No. 9 (IF), anyway.

   congruence, especially Launchbury's *uniform congruence*, & its proximity to what I call *uniquification*, as well as its surface similarity to *probabilistic conditioning* of `IF` branches. Mention Futamura's *generalized partial computation* as a more general, formal *truth maintenance system* along these same lines, though restricted to that which can be statically proven/deduced at partial evaluation time. Dynamically generated distribution spectra would relax this to that which can be empirically observed as well, which complements statically deduced information in cases where it would have to revert to the more conservative, general case when, perhaps, a hint about which unknowns are more likely to arise than others. *Etc.*

[59] **This is an instance of what the (non-authoritative) English Wikipedia `"Trampoline_(computers)"` entry might distinguish as a "*delegation trampoline*" [http://en.wikipedia.org]. Other plausible descriptive adjectives might include: "deflection", "jumpgate", "redirection", "stutter-step", "toreador", *etc. ad nauseum*.**

In the traditional (semi-authoritative) sense of the ubiquitous computer science "jargon file" (published as The Hacker's Dictionary [Raymond 96]), this is an instance of a "*true trampoline*" (p. ??).

4. TODO: [] Define *strictness/termination*: Argue inductive monotonicity, *etc.* [Nielson 87,88].

5. DONE: [] Define *CPS-conversion* [in overview chapter, not here.] [Plotkin 75] [Steele 78] and cite Reynold's nice history of the concept/term.

6. TODO: [] *kathenosynthesis*— a play on "kathenotheism"— means "one at a time" creation (literally, "each, one at a time"). Use it.

7. DONE: See Rule No. 9 (IF).

   assertions apropos *Generalized Partial Computation* [Futamura & Nogi 88]

   expectations apropos *conditioned probability* [Drake 67, p. 50].

8. TODO: [] empirically observed, dynamic, opened distributions -v-

   TODO: [] axiomatically derived, static, closed partitions,

   ...their implications for `ELSE` stop-gap failsafe emergency escape hatches.

9. TODO: [] Mogensen [Mogensen 88] then

   TODO: [] Launchbury [Launchbury 91],

   TODO: [] they investigated partially static data. Compare that to spectral distribution types.

End of Chapter 0.

[This page intentionally left very nearly blank.]

[This page intentionally left very nearly blank.]

[This page intentionally left very nearly blank.]

Part O

Prologue

Part I

Examples

[This page intentionally left very nearly blank.]

Part II

Details

Part III

Conclusion

[This page intentionally left very nearly blank.]

Part IV

Appendices