J. Functional Programming 3 (3): 365–387, July 1993 © 1993 Cambridge University Press

Specifying the correctness of binding-time analysis

MITCHELL WAND[1]

College of Computer Science, Northeastern University, 360 Huntington Avenue, 161CN, Boston, MA 02115, USA

(*e-mail*:wand@ccs.neu.edu)

Abstract

Mogensen has exhibited a very compact partial evaluator for the pure lambda calculus, using binding-time analysis followed by specialization. We give a correctness criterion for this partial evaluator and prove its correctness relative to this specification. We show that the conventional properties of partial evaluators, such as the Futamura projections, are consequences of this specification. By considering both a flow analysis and the transformation it justifies together, this proof suggests a framework for incorporating flow analyses into verified compilers.

Capsule review

A self-applicable partial evaluator for a higher-order language was published by Gomard and Jones in 1991, using a binding-time analysis based on type inference. Mogensen presented a compactly coded self-applicable partial evaluator for the pure lambda calculus in 1992, extending the earlier binding-time analysis with recursive types.

A traditional input-output specification of a partial evaluator is insufficiently specific to allow a rigourous correctness proof, a problem which this paper addresses. Working with Mogensen's partial evaluator, Wand defines a correctness criterion and proves the partial evaluator correct.

The method of Gomard and Jones plays a key role in the correctness proof: binding-time analysis assigns simple typings (plus *rec*) to the subexpressions of the partial evaluator's code. The typings drive both the partial evaluator's operation and the correctness proof, where in the latter, the correctness criterion is phrased as a logical relation. The usual proof techniques then apply.

A pleasant corollary of the correctness proof is that the three Futamura projections hold. This result is independent of the partial evaluator's code, and suggests that Wand's correctness criterion is a fundamental property of all well designed partial evaluators.

1 Introduction

Compilers for modern programming languages such as Scheme, ML, or Haskell typically perform a variety of optimizations in order to produce good code. Typically, an optimization consists of an analysis to collect information about a portion of the

[1] Work supported by the National Science Foundation under grant numbers CCR-9002253 and CCR-9014603. A preliminary version of this paper appeared in the Proceedings of the 20th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (January, 1993).

program, followed by a transformation based on the results of the analysis. While program analyses of various sorts have been studied intensively for many years, it has proven remarkably difficult to specify the correctness of an analysis in a way that actually justifies the resulting transformation.

Here we study one such analysis: binding-time analysis. Binding-time analysis annotates a program to indicate which portions of the program can be executed at compile-time and which cannot be executed until run-time. Because it is temporal rather than value-based, binding-time analysis does not seem to fit comfortably into the framework of abstract interpretation. Therefore it poses a challenge to theories of analysis and transformation.

Binding-time analysis is intended to support program specialization or staging (Jorring and Scherlis, 1986). Here we show the correctness of a binding-time analysis by showing that it justifies a specializer.

Our framework is based on the off-line partial evaluator of Mogensen (1992b), a very compact self-applicable partial evaluator for the pure λ-calculus. Mogensen's partial evaluator formulates binding-time analysis as a solution of a set of constraints on the annotation of a program (that is, as an annotated type-inference tree for a certain type system). The constraints are based on those in Gomard (1990). The annotated tree is fed to a program-specializer P, which produces the residual (run-time) term.

Our main theorem is that if the specializer is given any annotation of the input program that satisfies the binding-time constraints, its output is a suitably specialized version of the input program.

In order to do this, we must formalize the notion of a 'suitably specialized version of the input program.' Our version of this notion is similar to, but simpler than, the specification given by Gomard (1992). We then show that the typical properties of partial evaluators, such as the Futamura projections, follow from the specification.

This paper is organized as follows: in Section 2 we review the relevant fundamentals of partial evaluation and consider how an arbitrary programming language can be modelled inside the λ-calculus. In Section 3 we show how the λ-calculus itself can be modelled. In Section 4 we present Mogensen's partial evaluator. In Section 5 we state and prove our main theorem relating binding-time analysis and partial evaluation, and in Section 6 we show how the expected specialization behavior of partial evaluators follows from the main theorem. In Section 7 we consider self-application of the partial evaluator, and show how the Futamura projections can be derived from the main theorem. In Section 8 we present a brief account of the operational semantics of the language of annotated terms. Section 9 compares our work to other approaches to binding-time analysis, and Section 10 gives some conclusions.

2 Partial evaluation and representation

We begin by recapitulating the usual terminology of partial evaluation.

Definition 1
A *programming language* L is specified by a set Π of L-programs, a set Δ of L-data, and for every $n \geq 0$, an $(n+2)$-ary relation

$$L_n(\pi, d_1, \ldots d_n) \equiv d$$

such that each L_n is a partial function. The relation $L_n(\pi, d_1, \ldots d_n) \equiv d$ represents the assertion that program π, given inputs $d_1, \ldots, d_n$, produces output d.

We will almost always omit the subscript on L. We shall ambiguously regard this as an $(n+2)$-place relation or as a partial function from programs and inputs to outputs. In the latter interpretation, $\equiv$ is the equality on elements of the set Δ. The functional interpretation is used, for example, in the definition of a partial evaluator:

Definition 2
An L-program P is a *partial evaluator* iff for all L-programs π and L-data d_1 and d_2,

$$L_1(L_1(P, \pi, d_1), d_2) = L_2(\pi, d_1, d_2)$$

The partial evaluators we will consider are *off-line*. In an off-line partial evaluator, the program π is first transformed in some way independent of the static input d_1:

Definition 3
An *off-line partial evaluator* is a pair (Φ, P), where Φ is a map $\Pi \to \Delta$ and P is an L-program such that for all L-programs π and L-data d_1 and d_2,

$$L_2(L_2(P, \Phi(\pi), d_1), d_2) = L_3(\pi, d_1, d_2)$$

In general, we will not be concerned with the algorithmic aspects of the transformation Φ, so we will not bother to consider the implementation of Φ as an L-program.

For these definitions to make sense, π (or $\Phi(\pi)$ in the off-line case) must also be an element of the set Δ of L-data. To prove the correctness of partial evaluators and similar program transformers, we must consider this representation explicitly. Our tool for studying this representation will be the pure λ-calculus.

The basic techniques for representing various quantities in the λ-calculus are well-known. For example, when considering computations on the integers, one first defines a set of numerals, that is, a mapping associating each integer n with a closed normal form λ-term $\lceil n \rceil$. Then we say a function f on the integers is representable iff there exists a λ-term F such that for all n, $F\lceil n \rceil =_\beta \lceil f(n) \rceil$.

To carry out a similar analysis for an arbitrary programming language L, we give a coding $\langle - \rangle$ from programs and data in the language into normal form λ-terms, and a λ-term S which is the semantics or interpreter for the language. We can then define the input-output behavior of the language by

$$L_n(\pi, d_1, \ldots d_n) \equiv d \iff S\langle \pi \rangle \langle d_1 \rangle \ldots \langle d_n \rangle = \langle d \rangle$$

where the equality on the right is equality in some λ-theory (for us, this will be $=_\beta$). Note that unlike the usual formulations of denotational semantics, in which the translation from a program to its meaning is specified informally, the semantics

S is internalized as a term in the λ-calculus. This step is crucial for dealing with self-application.

From now on we will assume that our programming languages have been specified in this way. As an example of this style of specification, we can define the notion of a translation of an L-program into the λ-calculus:

Definition 4
A *translation* of an L-program π is a λ-term T such that for all S-data d, $T\langle d\rangle = S\langle\pi\rangle\langle d\rangle$.

Thus $T\langle d\rangle =_\beta \langle d'\rangle \iff S\langle\pi\rangle\langle d\rangle = \langle d'\rangle \iff L(\pi, d) \equiv d'$.

3 Representing λ-terms in pure λ-calculus

We are interested in performing binding-time analysis and partial evaluation on λ-terms themselves. We therefore need to specify the λ-calculus as a programming language dealing with λ-terms as inputs and outputs. To do this, we must specify a representation of λ-terms and an interpreter manipulating those representations.

The representation needs to code each λ-term as a normal form λ-term. Mogensen (1992a) uses the following coding:

$$\begin{aligned}\ulcorner x\urcorner &\equiv \lambda abc.ax\\ \ulcorner MN\urcorner &\equiv \lambda abc.b\ulcorner M\urcorner\ulcorner N\urcorner\\ \ulcorner \lambda x.M\urcorner &\equiv \lambda abc.c(\lambda x.\ulcorner M\urcorner)\end{aligned}$$

This coding uses two tricks: first, it uses the coding of the sum $A+B$ as $(\forall C)((A \to C) \to (B \to C) \to C)$, so that each element of a 'sum domain' is modelled as its own case-function. Thus a case expression

$$\begin{aligned}&\textbf{case } m \textbf{ of}\\ &\quad Var(x) \Rightarrow \ldots x \ldots\\ &\quad App(M,N) \Rightarrow \ldots M \ldots N \ldots\\ &\quad Abs(M) \Rightarrow \ldots M \ldots\end{aligned}$$

would be represented as

$$m(\lambda x.\ldots x\ldots)(\lambda mn.\ldots m\ldots n\ldots)(\lambda m.\ldots m\ldots)$$

Second, it uses so-called 'higher-order abstract syntax' (Pfenning and Elliott, 1988), in which the binding operators in the represented language are modelled by binding in the lambda-calculus. Any free occurrences of x in the body of $\lambda x.M$ will be bound by the x in $\lambda x.\ulcorner M\urcorner$.

We also need to specify an interpreter. This task will be filled by a self-interpreter E. A *self-interpreter* is a λ-term E such that $E\ulcorner M\urcorner =_\beta M$ (Barendregt, 1991). E needs to have the properties that

$$\begin{aligned}E(Var\, x) &= x\\ E(App\, M N) &= (EM)(EN)\\ E(Abs\, M) &= \lambda v.E(Mv)\end{aligned}$$

Here we are using informal notation for abstract syntax of the λ-term input to E. It is easy to display a term that satisfies these properties:

$$\begin{aligned} E \equiv Y(\lambda e.\lambda m.m(\lambda x.x) \\ (\lambda mn.(em)(en)) \\ (\lambda m.\lambda v.e(mv))) \end{aligned}$$

where Y is the standard fixpoint operator.

Theorem 1 (E is a self-interpreter)
For any λ-term M, $E\lceil M \rceil =_\beta M$

Proof
By structural induction on M (Mogensen, 1992a). □

If we use E as our interpreter, then the programming language of Λ of λ-terms has an input-output relation specified by

$$\begin{aligned} &\Lambda(M, N_1, \ldots N_n) \equiv R \\ \iff &E\lceil M \rceil \lceil N_1 \rceil \ldots \lceil N_k \rceil =_\beta \lceil R \rceil \\ \iff &M\lceil N_1 \rceil \ldots \lceil N_k \rceil =_\beta \lceil R \rceil \end{aligned}$$

Since here the set Δ of Λ-data is the set of normal-form λ-terms, our use of $\equiv$ for equality on elements of Δ and its usual use for equality of λ-terms up to α-congruence coincide. Henceforth, we will use the symbol $=$ to denote β-convertibility, and $\equiv$, when necessary, to denote α-congruence of λ-terms.

4 The partial evaluator

We next present the partial evaluator for Λ given by Mogensen (1992b). This is an off-line partial evaluator, so it is divided into two phases: a binding-time analysis (the transformation Φ), and a specialization phase (a Λ-program P).

The binding-time analysis produces an annotated version of the input program by assigning each phrase of the program a type in a system due to Gomard (1990). The types are as follows:

$$t ::= d \mid v \mid t \rightarrow t \mid \mu v.t$$

where in $\mu v.t$, t must be of the form $(t_1 \rightarrow \ldots \rightarrow t_n \rightarrow d)$, and all occurrences of v in t must be positive. The intention is that the type d denotes the type of 'dynamic' quantities: i.e. representations of untyped λ-terms. The other types are built from d in the usual way, and correspond to the types one might find in an ordinary compiler. Recursive types are intended to denote their unfolding into an infinite tree. The presence of the recursive types allows loops to be formed in the static portion of the analyzed program, and the use of a single run-time type d allows unrestricted self-application in the dynamic portion.

Annotated terms are given by the following grammar:

$$W ::= v \mid WW \mid \lambda v.W \mid W\_W \mid \underline{\lambda} v.W$$

The first three productions correspond to static (compile-time) calculations, and the last two correspond to dynamic (residual, run-time) operations, which correspond at compile-time to emitting code. Annotated terms have elsewhere been called 2-level λ-terms.

The annotated terms are produced by decorating a type-inference tree. Thus, instead of considering typing judgements of the form $A \vdash M : t$, we consider judgements of the form $A \vdash M : t\ [W]$, meaning that under hypotheses A, term M can be assigned type t with annotated term W. When this judgement is true, we say that W is an annotation of M with type t. The type system can be written as follows:

$$A \vdash x : A(x)\ [x]$$

$$\frac{A \vdash M : s \to t\ [W_1] \quad A \vdash N : s\ [W_2]}{A \vdash MN : t\ [W_1 W_2]}$$

$$\frac{A[x \mapsto s] \vdash M : t\ [W]}{A \vdash (\lambda x.M) : s \to t\ [\lambda x.W]}$$

$$\frac{A \vdash M : d\ [W_1] \quad A \vdash N : d\ [W_2]}{A \vdash MN : d\ [W_1\_W_2]}$$

$$\frac{A[x \mapsto d] \vdash M : d\ [W]}{A \vdash (\lambda x.M) : d\ [\underline{\lambda} x.W]}$$

A tree of judgements in which every node satisfies these constraints is a derivation. For purposes of checking these constraints, types are regarded as equal iff their unfoldings into infinite trees are equal; thus no separate rules for recursive types are necessary. We shall not consider algorithms for performing these annotations; instead we shall assume that we have such an algorithm Φ; see Gomard (1990) and Henglein (1991).

Mogensen uses a representation scheme for annotated terms similar to that for ordinary λ-terms:

$$\begin{aligned}
\lfloor x \rfloor &\equiv \lambda abcde.ax \\
\lfloor W W' \rfloor &\equiv \lambda abcde.b\lfloor W \rfloor \lfloor W' \rfloor \\
\lfloor \lambda x.W \rfloor &\equiv \lambda abcde.c(\lambda x.\lfloor W \rfloor) \\
\lfloor W\_W' \rfloor &\equiv \lambda abcde.d\lfloor W \rfloor \lfloor W' \rfloor \\
\lfloor \underline{\lambda} x.W \rfloor &\equiv \lambda abcde.e(\lambda x.\lfloor W \rfloor)
\end{aligned}$$

To specialize a program, one must interpret the annotated term, evaluating all the static (compile-time) parts (just like E) and emitting code for all the dynamic (residual) parts. This can be expressed in terms of abstract syntax by:

$$\begin{aligned}
P(Pvar\ x) &= x \\
P(Sapp\ W\ W') &= (P\ W)(P\ W') \\
P(Sabs\ W) &= \lambda v.P(W v) \\
P(Dapp\ W\ W') &= App(P\ W)(P\ W') \\
P(Dabs\ W) &= Abs(\lambda v.P(W(Var\ v)))
\end{aligned}$$

where $Pvar$, etc., represent the abstract syntax of annotated terms. This can be realized by a λ-term P operating on the $\lfloor - \rfloor$ representation as follows:

$$\begin{aligned} P \equiv Y(\lambda p.\lambda m.m(&\lambda x.x)\\ &(\lambda ww'.(pw)(pw'))\\ &(\lambda w.\lambda v.p(wv))\\ &(\lambda ww'.\lambda abc.b(pw)(pw'))\\ &(\lambda w.\lambda abc.c(\lambda v.p(w(\lambda abc.av))))) \end{aligned}$$

We first show some simple properties of P. As usual, we will use the symbol $=$ to denote β-convertibility, and $\equiv$, when necessary, to distinguish α-congruence of λ-terms.

Lemma 1
$\lfloor W[y/x] \rfloor \equiv \lfloor W \rfloor [y/x]$

Proof
Easy induction on W. □

Lemma 2
$P(\lfloor W \rfloor [N/x]) = (P\lfloor W \rfloor)[N/x]$

Proof
By induction on the size of W. We will do two cases; the others are similar. For the base case, we have $P(\lfloor x \rfloor [N/x]) = P(\lambda abcde.aN) = N$ and $(P\lfloor x \rfloor)[N/x] = x[N/x] = N$. For static abstractions over a variable y distinct from x, we have

$$\begin{aligned} & P(\lfloor \lambda y.M \rfloor [N/x]) & \\ &= P((\lambda abcde.c(\lambda y.\lfloor M \rfloor))[N/x]) && \text{definition of } \lfloor \lambda y.M \rfloor \\ &= P(\lambda abcde.c(\lambda y'.\lfloor M \rfloor [y'/y][N/x])) && \text{substitution } (y' \text{ fresh}) \\ &= \lambda v.P((\lambda y'.\lfloor M \rfloor [y'/y][N/x])v) && \text{definition of } P \\ &= \lambda y'.P((\lambda y'.\lfloor M \rfloor [y'/y][N/x])y') && \alpha\text{-conversion} \\ &= \lambda y'.P(\lfloor M \rfloor [y'/y][N/x]) && \beta\text{-conversion} \\ &= \lambda y'.P(\lfloor M[y'/y] \rfloor [N/x]) && \text{Lemma 1} \\ &= \lambda y'.(P\lfloor M[y'/y] \rfloor)[N/x] && \text{induction hypothesis at } M[y'/y] \\ & && (M[y'/y] \text{ is smaller than } \lambda y.M) \\ &= \lambda y'.(P(\lfloor M \rfloor [y'/y]))[N/x] && \text{Lemma 1} \\ &= \lambda y'.(P\lfloor M \rfloor)[y'/y][N/x] && \text{induction hypothesis} \\ &= (\lambda y.P\lfloor M \rfloor)[N/x] && \text{substitution} \\ &= P(\lfloor \lambda y.M \rfloor)[N/x] && \text{definition of } P \end{aligned}$$

□

Corollary 1 (Substitution Lemma)
If ρ is a substitution, then $(P\lfloor W \rfloor)\rho = P(\lfloor W \rfloor \rho)$.

Proof
By repeated application of the preceding lemma for each $x \in dom(\rho)$. □

We will often write $P\lfloor W \rfloor \rho$ to mean, interchangeably, either of the terms above. Note that it would be incorrect to bring the ρ inside the $\lfloor - \rfloor$: in general $P(\lfloor W \rfloor \rho) \neq P(\lfloor W\rho \rfloor)$. We will consider this further in Section 8. The resemblance of $P\lfloor W \rfloor \rho$ to a conventional environment semantics is, of course, not coincidental, since by choosing equality to be interconvertibility, we are in effect working in the open term model, in which evaluation in an environment is the same as substitution. Our results hold *a fortiori* in any λ-model.

5 Correctness of the partial evaluator

Our goal is to show that the pair (Φ, P) is a correct off-line partial evaluator. The correctness proof must somehow involve the binding-time analysis as well: otherwise one could change the binding-time analysis arbitrarily without invalidating P, which is clearly impossible. Furthermore, the usual notion of partial evaluation only involves the programs that can be assigned relatively simple types like d or $d \rightarrow d$, while the algorithm clearly involves arbitrary types. Furthermore, the algorithm P crucially involves non-closed terms. Hence we must find an induction hypothesis that covers arbitrary types and terms with free variables.

We do this by using the technique of logical relations. For every closed type t, we will define a binary relation R_t. We begin by setting up some standard machinery. If R and S are binary relations on λ-terms, define the relation $R \rightarrow S$ in the standard way by

$$\begin{aligned}&(R \rightarrow S)(M, M') \\ &\iff (\forall N, N')(R(N, N') \implies S(MN, M'N'))\end{aligned}$$

Next, let η range over partial maps from type variables to binary relations, and let $\langle\rangle$ denote the empty map. We can define $R_{t,\eta}$ by:

$$\begin{aligned}R_{d,\eta} &= \{(M, M') \mid EM =_\beta M'\} \\ R_{v,\eta} &= \eta(v) \\ R_{(s \rightarrow t),\eta} &= R_{s,\eta} \longrightarrow S_{t,\eta} \\ R_{(\mu v.t),\eta} &= \bigcap\{S \mid R_{t,\eta[S/v]} \subseteq S\}\end{aligned}$$

We shall always assume that η is defined on all the free variables of t; it is a routine matter to check that this assumption is an invariant of all our calculations. Had our language included other base types o, R_o would be the equality relation on the other base types.

Lemma 3 (Fixed-Point Lemma)

1. If $R_{(\mu v.t),\eta} = S$, then $S = R_{t,\eta[S/v]}$.
2. If t' is a closed type, then $R_{t,\eta[t'/v]} = R_{t[t'/v],\eta}$
3. $R_{(\mu v.t),\eta} = R_{t[(\mu v.t)/v],\eta}$

Proof

(1) Since all the occurrences of v in t must be positive, the intersection in the last line must be a fixed point. (2) Routine calculation. (3) Follows from (1) and (2). □

In the remainder we will consider only closed types, for which we define $R_t = R_{t,\langle\rangle}$.

Lemma 4 (Admissibility)
If $M = M'$ and $N = N'$, then $R_t(M,N) \Longleftrightarrow R_t(M',N')$.

Proof
By induction on the length $|t|$ of t, defined by $|d| = 0$, $|s \to t| = |t| + 1$, and $|\mu v.t| = |t| + 1$. If the type is d, then

$$\begin{aligned} R_d(M,N) &\Longleftrightarrow EM = N \\ &\Longleftrightarrow EM' = N' \\ &\Longleftrightarrow R_d(M',N') \end{aligned}$$

If the type is $s \to t$, then $|t| < |s \to t|$, so we have

$$\begin{aligned} R_{s\to t}(M,N) &\Longleftrightarrow [R_s(P,Q) \Longrightarrow R_t(MP,NQ)] \\ &\Longleftrightarrow [R_s(P,Q) \Longrightarrow R_t(M'P,N'Q)] \\ &\Longleftrightarrow R_{s\to t}(M',N') \end{aligned}$$

The last case is that the type is $\mu v.t$. Now, since t is of the form $t_1 \to \ldots \to t_n \to d$, $t[\mu v.t/v] = t_1[\mu v.t/v] \to \ldots t_n[\mu v.t/v] \to d$. So $|t[\mu v.t/v]| = |t| < |\mu v.t|$. So

$$\begin{aligned} R_{\mu v.t}(M,N) &\Longleftrightarrow R_{t[\mu v.t/v]}(M,N) && \text{Lemma 3} \\ &\Longleftrightarrow R_{t[\mu v.t/v]}(M',N') && \text{Lemma 4} \\ &\Longleftrightarrow R_{\mu v.t}(M',N') && \text{Lemma 3} \end{aligned}$$

□

The following definition makes the main theorem easier to state.

Definition 5
For substitutions ρ and ρ' we say $(\rho,\rho') \models A$ iff for all $x \in dom(A)$,

$$R_{A(x)}(\rho(x), \rho'(x))$$

We say $A \models M : t\ [W]$ iff

$$(\rho,\rho') \models A \Longrightarrow R_t(P\lfloor W \rfloor \rho, M\rho')$$

Theorem 2 (Main Theorem)
If $A \vdash M : t\ \ [W]$, then $A \models M : t\ \ [W]$.

The theorem says roughly that if M can be assigned type t with annotated term W, then $P(\lfloor W \rfloor)$ and M are related by relation R_t. More precisely, it says that if ρ and ρ' are substitutions related by symbol table (type hypotheses) A, then the corresponding substitution instances of $P(\lfloor W \rfloor)$ and M are also related. This proposition is the generalization of the simple first-order case ($E(P\lfloor W \rfloor) = M$) to handle higher-order terms.

Proof
The proof is by induction on the structure of the proof of $A \vdash M : t\ \ [W]$. Thus the proof has five cases, one for each rule of inference for $\vdash$. For each case, we show that the rule preserves $\models$.

Case 1 (Variable). Assume $x \in dom(A)$. Then $A \vdash x : A(x) \quad [x]$. We need to show $A \models x : A(x) \quad [x]$. So assume $(\rho, \rho') \models A$. We need to prove that

$$R_{A(x)}(P\lfloor x \rfloor \rho, x\rho')$$

holds. We calculate:

$$\begin{aligned} R_{A(x)}(P\lfloor x \rfloor \rho, x\rho') &\iff R_{A(x)}(x\rho, x\rho') && \text{definition of } P \\ &\iff R_{A(x)}(\rho(x), \rho'(x)) \end{aligned}$$

which holds because $(\rho, \rho') \models A$

Case 2 (Static Application). Assume $A \models M : s \to t \quad [W]$ and $A \models N : s \quad [W']$. We must show $A \models MN : t \quad [W W']$. So assume $(\rho, \rho') \models A$. We want to show

$$R_t(P\lfloor W W' \rfloor \rho, (MN)\rho')$$

By the induction hypotheses, we have $R_{s\to t}(P\lfloor W \rfloor \rho, M\rho')$ and $R_s(P\lfloor W' \rfloor \rho, N\rho')$. So we have

$$\begin{array}{ll} R_t((P\lfloor W \rfloor \rho)(P\lfloor W' \rfloor \rho), (M\rho')(N\rho')) & \text{definition of } R_{s\to t} \\ R_t(((P\lfloor W \rfloor)(P\lfloor W' \rfloor))\rho, (MN)\rho') & \text{substitution} \\ R_t(P\lfloor W W' \rfloor \rho, (MN)\rho') & \text{definition of } P \end{array}$$

Case 3 (Static Abstraction). Assume $A[x \mapsto s] \models M : t \quad [W]$. We need to show that $A \models (\lambda x.M) : s \to t \quad [\lambda x.W]$. So assume $(\rho, \rho') \models A$. We need to show

$$R_{s\to t}(P\lfloor \lambda x.W \rfloor \rho, (\lambda x.M)\rho')$$

By the definition of $R_{s\to t}$, it suffices to assume $R_s(W_1, M_1)$ and then show that

$$R_t((P\lfloor \lambda x.W \rfloor \rho)W_1, ((\lambda x.M)\rho')M_1)$$

Now,

$$\begin{aligned} (P\lfloor \lambda x.W \rfloor \rho)W_1 &= ((\lambda x.P\lfloor W \rfloor)\rho)W_1 \\ &= (P\lfloor W \rfloor)\rho[W_1/x] \end{aligned}$$

(we write $\rho[N/x]$ for the substitution mapping x to N and any other variable y to $\rho(y)$; thus $((\lambda x.M)\rho)N =_\beta M(\rho[N/x])$.) Furthermore, $((\lambda x.M)\rho')M_1 = M\rho'[M_1/x]$, so by Lemma 4 it will suffice to show

$$R_t((P\lfloor W \rfloor)\rho[W_1/x], M\rho'[M_1/x])$$

But $(\rho[W_1/x], \rho'[M_1/x]) \models A[x \mapsto s]$, so the desired conclusion follows from $A[x \mapsto s] \models M : t \quad [W]$.

Case 4 (Dynamic Application). Assume $A \models M : d \quad [W]$ and $A \models N : d \quad [W']$. We must show $A \models MN : d[W\_W']$. So assume $(\rho, \rho') \models A$. We want to show

$$R_d(P\lfloor W\_W' \rfloor \rho, (MN)\rho')$$

We calculate:

$$\begin{array}{ll}
E(P\lfloor W\_W'\rfloor\rho) & \\
= E(P(\lambda abcde.d(\lfloor W\rfloor\rho)(\lfloor W'\rfloor\rho))) & \text{definition of } \lfloor - \rfloor \\
= E(\lambda abc.b(P\lfloor W\rfloor\rho)(P\lfloor W'\rfloor\rho)) & \text{definition of } P \\
= (E(P\lfloor W\rfloor\rho))(E(P\lfloor W'\rfloor\rho)) & \text{definition of } E \\
= (M\rho')(N\rho') & \text{induction hypotheses} \\
= (MN)\rho' & \text{substitution}
\end{array}$$

Case 5 (Dynamic Abstraction). Assume $A[x \mapsto d] \models M : d \quad [W]$. We need to show that $A \models \lambda x.M : d \quad [\underline{\lambda} x.W]$. So assume $(\rho,\rho') \models A$. We need to show that

$$R_d(P\lfloor \underline{\lambda} x.w\rfloor\rho, (\lambda x.M)\rho')$$

Now, $R_d(\lambda abc.ax, x)$ holds, so let $\rho_1 = \rho[\lambda abc.ax/x]$ and $\rho_1' = \rho'[x/x]$; then $(\rho_1,\rho_1') \models A[x \mapsto d]$. Therefore, by the induction hypothesis, we know that

$$R_d(P\lfloor W\rfloor\rho_1, M\rho')$$

that is,

$$E(P\lfloor W\rfloor\rho_1) = M\rho_1' \tag{1}$$

Now, to establish $R_d(P\lfloor\underline{\lambda} x.W\rfloor\rho, (\lambda x.M)\rho')$, we calculate:

$$\begin{array}{ll}
E(P\lfloor\underline{\lambda} x.W\rfloor\rho) & \\
= E(P(\lfloor\underline{\lambda} x.W\rfloor r)) & \text{Corollary 1} \\
= E(P(\lambda abcde.e(\lambda x.\lfloor W\rfloor))\rho) & \text{definition of } \lfloor\underline{\lambda} x.W\rfloor \\
= E(P(\lambda abcde.e((\lambda x.\lfloor W\rfloor)\rho))) & \text{definition of substitution} \\
= E(\lambda abc.c(\lambda x.P(((\lambda x.\lfloor W\rfloor)\rho)(\lambda abc.ax)))) & \text{definition of } P \\
= E(\lambda abc.c(\lambda x.P(\lfloor W\rfloor\rho[(\lambda abc.ax)/x]))) & \beta\text{-reduction} \\
= E(\lambda abc.c(\lambda x.P(\lfloor W\rfloor\rho_1))) & \text{definition of } \rho_1 \\
= \lambda x.E(P(\lfloor W\rfloor\rho_1)) & \text{definition of } E \\
= \lambda x.M\rho'[x/x] & \text{Equation 1} \\
= (\lambda x.M)\rho' &
\end{array}$$

This completes the cases for the proof. □

Most of the time we will be concerned with closed terms:

Corollary 2
If M is a closed term, and $\emptyset \vdash M : t \ [W]$, then $R_t(P\lfloor W\rfloor, M)$.

This specification for a partial evaluator looks rather different from the usual one, so it is useful to check to see that the usual properties of partial evaluators hold. Most often we will be interested in the following case:

Corollary 3 (Specialization Theorem)
Let M, N, N' be closed terms, and let W be an annotation of M with type $t \rightarrow d$. If $R_t(N, N')$, then $E(P\lfloor W\rfloor N) = MN'$

Proof

$R_{t\to d}(P\lfloor W\rfloor, M)$	Corollary 2
$R_t(N, N')$	Assumption
$R_d(P\lfloor W\rfloor N, MN')$	Definition of $R_{t\to d}$
$E(P\lfloor W\rfloor N) = MN'$	Definition of R_d

□

6 Relation to traditional partial evaluation

According to definition 3, (Φ, P) is an off-line partial evaluator iff for all L-programs π and L-data d_1 and d_2,

$$L_1(L_1(P, \Phi(\pi), d_1), d_2) = L_2(\pi, d_1, d_2)$$

In the case of the λ-calculus, programs are closed terms and data are terms. Programs are represented using the coding $\lceil - \rceil$, and the datum $\Phi(\pi)$ is an annotated term, represented using the coding $\lfloor - \rfloor$. This leaves open the question of how the other data are represented.

The easiest way to do this is to represent the datum d_1 as $\lceil d_1 \rceil$ at specialization time (that is, when it is static data), and as itself at run time (when it is dynamic data). Then at specialization time we have

$$\begin{aligned} & L_2(P, \Phi(\pi), d_1) = Q \\ \iff & P\lfloor \Phi(\pi)\rfloor \lceil d_1 \rceil = \lceil Q \rceil \\ \iff & E(P\lfloor \Phi(\pi)\rfloor \lceil d_1 \rceil) = Q \end{aligned}$$

but at run time

$$\begin{aligned} & L_1(L_1(P, \Phi(\pi), d_1), d_2) = L_2(\pi, d_1, d_2) \\ \iff & L_1(Q, d_2) = L_2(\pi, d_1, d_2) \\ \Longleftarrow & Qd_2 = \pi d_1 d_2 \\ \iff & E(P\lfloor \Phi(\pi)\rfloor \lceil d_1 \rceil) d_2 = \pi d_1 d_2 \end{aligned}$$

So it will clearly suffice to show that $E(P\lfloor \Phi(\pi)\rfloor \lceil d_1 \rceil) = \pi d_1$. This is accomplished easily:

Corollary 4 (Partial Evaluation Theorem (mixed representation))
Let M be a closed term with an annotation W of type $d \to d$. Then for any closed term N,

$$E(P\lfloor W\rfloor \lceil N \rceil) = MN$$

Hence $P\lfloor W\rfloor \lceil N \rceil$ is a closed term equivalent under E to MN: that is, it is a program for M specialized to N.

Proof
$E\lceil N \rceil = N$, so we have $R_d(\lceil N \rceil, N)$. Therefore, by the Specialization Theorem we have

$$E(P\lfloor W\rfloor \lceil N \rceil) = MN$$

□

Here W plays the role of $\Phi(\pi)$. This result also shows the correctness of the partial evaluator using a double encoding, in which d_1 is represented as $\lceil d_1 \rceil$ at run time and as $\lceil\lceil d_1 \rceil\rceil$ at specialization time (Launchbury, 1991).

It would be better to have a version of partial evaluation that used the same representation for both static and dynamic data. This can be obtained by looking at the types which the binding-time analysis assigns to the representations.

For any type β, let T_β denote the type

$$(\mu t).((\beta \to \beta) \to (t \to t \to \beta) \to ((\beta \to t) \to \beta) \to \beta)$$

This type, introduced in (Mogensen, 1992b), is the type of the representation of λ-terms interpreted as case functions with target β.

Lemma 5
If M is a closed λ-term, and β is any type, then $R_{T_\beta}(\lceil M \rceil, \lceil M \rceil)$.

Proof
The proof is by induction on M. As usual we need to consider open terms as well. The induction hypothesis is: Let ρ, ρ' be substitutions such that for all $x \in dom(\rho) = dom(\rho')$, $R_\beta(\rho(x), \rho'(x))$. Then $R_{T_\beta}(\lceil M \rceil \rho, \lceil M \rceil \rho')$.

Now, by the structure of T_β, to show $R_{T_\beta}(\lceil M \rceil \rho, \lceil M \rceil \rho')$ we must show that whenever $R_{\beta\to\beta}(N, N')$, $R_{T_\beta \to T_\beta \to \beta}(P, P')$, and $R_{(\beta\to T_\beta)\to\beta}(Q, Q')$, we have

$$R_\beta((\lceil M \rceil \rho)NPQ, (\lceil M \rceil \rho')N'P'Q')$$

We show this for each case in turn:

Case 1 (Variable). We calculate:

$$\begin{array}{ll} R_\beta((\lceil M \rceil \rho)NPQ, (\lceil M \rceil \rho')N'P'Q') & \\ \iff R_\beta(((\lambda abc.ax)\rho)NPQ, ((\lambda abc.ax)\rho')N'P'Q') & \text{definition of } \lceil x \rceil \\ \iff R_\beta(N(\rho(x)), N'(\rho'(x))) & \text{admissibility} \end{array}$$

But $R_{\beta\to b}(N, N')$ and $R_\beta(\rho(x), \rho'(x))$, so $R_\beta(N(\rho(x)), N'(\rho'(x)))$ is true.

Case 2 (Application). We calculate similarly:

$$\begin{array}{l} R_\beta((\lceil M_1 M_2 \rceil \rho)NPQ, (\lceil M_1 M_2 \rceil \rho')N'P'Q') \\ \iff R_\beta(((\lambda abc.b\lceil M_1 \rceil \lceil M_2 \rceil)\rho)NPQ, ((\lambda abc.b\lceil M_1 \rceil \lceil M_2 \rceil)\rho')N'P'Q') \\ \qquad\qquad\qquad\qquad\qquad\qquad \text{definition of } \lceil M_1 M_2 \rceil \\ \iff R_\beta((\lambda abc.b(\lceil M_1 \rceil \rho)(\lceil M_2 \rceil \rho))NPQ, (\lambda abc.b(\lceil M_1 \rceil \rho')(\lceil M_2 \rceil \rho'))N'P'Q') \\ \qquad\qquad\qquad\qquad\qquad\qquad \text{substitution} \\ \iff R_\beta(P(\lceil M_1 \rceil \rho)(\lceil M_2 \rceil \rho), P(\lceil M_1 \rceil \rho')(\lceil M_2 \rceil \rho')) \\ \qquad\qquad\qquad\qquad\qquad\qquad \text{admissibility} \end{array}$$

But by the induction hypothesis $R_{T_\beta}(\lceil M_1 \rceil \rho, \lceil M_1 \rceil \rho')$, $R_{T_\beta}(\lceil M_2 \rceil \rho, \lceil M_2 \rceil \rho')$, and by the assumption $R_{T_\beta \to T_\beta \to \beta}(P, P')$, so the last line is true.

Case 3 (Abstraction). Last, we consider the case of $\lambda x.M$. We claim that $R_{\beta\to T_\beta}((\lambda x.\lceil M \rceil)\rho, (\lambda x.\lceil M \rceil)\rho')$. To establish the claim, assume that $R_\beta(S, S')$. We must then show that $R_{T_\beta}(((\lambda x.\lceil M \rceil)\rho)S, ((\lambda x.\lceil M \rceil)\rho')S')$:

$$\begin{array}{l} R_{T_\beta}((\lambda x.\lceil M \rceil)\rho S, (\lambda x.\lceil M \rceil)\rho' S') \\ \iff R_{T_\beta}(\lceil M \rceil(\rho[S/x]), \lceil M \rceil(\rho'[S'/x])) \end{array}$$

but $(\rho[S/x], \rho'[S'/x])$ satisfies the conditions of the induction hypothesis, so the last line is true.

Now we can calculate $R_{T_\beta}(\lceil \lambda x.M \rceil \rho, \lceil \lambda x.M \rceil \rho')$ as before:

$$\begin{aligned}
&R_\beta((\lceil \lambda x.M \rceil \rho)NPQ, (\lceil \lambda x.M \rceil \rho')N'P'Q') \\
&\iff R_\beta(((\lambda abc.c(\lambda x.\lceil M \rceil))\rho)NPQ, ((\lambda abc.c(\lambda x.\lceil M \rceil))\rho')N'P'Q') \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{definition of } \lceil \lambda x.M \rceil \\
&\iff R_\beta((\lambda abc.c((\lambda x.\lceil M \rceil)\rho))NPQ, (\lambda abc.c((\lambda x.\lceil M \rceil)\rho'))N'P'Q') \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{substitution} \\
&\iff R_\beta(Q((\lambda x.\lceil M \rceil)\rho), Q'((\lambda x.\lceil M \rceil)\rho')) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{admissibility}
\end{aligned}$$

but $R_{(\beta \to T_\beta) \to \beta}(Q, Q')$ and $R_{\beta \to T_\beta}((\lambda x.\lceil M \rceil)\rho, (\lambda x.\lceil M \rceil)\rho')$, so the last line is true. □

Using this type, we can show the correctness of the partial evaluator using the $\lceil - \rceil$ coding at both specialization time and run time. Using this representation we have that at specialization time

$$\begin{aligned}
&L_2(P, \Phi(\pi), d_1) = Q \\
&\iff P \lfloor \Phi(\pi) \rfloor \lceil d_1 \rceil = \lceil Q \rceil \\
&\iff E(P \lfloor \Phi(\pi) \rfloor \lceil d_1 \rceil) = Q
\end{aligned}$$

as before, but at run time

$$\begin{aligned}
&L_2(L_2(P, \Phi(\pi), d_1), d_2) = L_3(\pi, d_1, d_2) \\
&\iff L_2(Q, \lceil d_2 \rceil) = L_3(\pi, \lceil d_1 \rceil, \lceil d_2 \rceil) \\
&\iff Q \lceil d_2 \rceil = \pi \lceil d_1 \rceil \lceil d_2 \rceil \\
&\iff E(P \lfloor \Phi(\pi) \rfloor \lceil d_1 \rceil) \lceil d_2 \rceil = \pi \lceil d_1 \rceil \lceil d_2 \rceil
\end{aligned}$$

so it suffices to show that $E(P \lfloor \Phi(\pi) \rfloor \lceil d_1 \rceil) = \pi \lceil d_1 \rceil$:

Corollary 5 (Partial Evaluation Theorem (uniform representation))
Let M be a closed term with an annotation W of type $T_d \to d$. Then for any closed term N,

$$E(P \lfloor W \rfloor \lceil N \rceil) = M \lceil N \rceil$$

Hence $P \lfloor W \rfloor \lceil N \rceil$ is a closed term equivalent under E to $M \lceil N \rceil$: that is, it is a program for M specialized to N.

Proof
By the main theorem, $R_{T_d \to d}(P \lfloor W \rfloor, M)$. By Lemma 5, $R_{T_d}(\lceil N \rceil, \lceil N \rceil)$ Therefore, $R_d(P \lfloor W \rfloor \lceil N \rceil, M \lceil N \rceil)$, that is $E(P \lfloor W \rfloor \lceil N \rceil) = MN$. □

This completes the connection with the 'classical' definition of partial evaluation in Section 2.

7 Correctness of the self-applications

We next check to see that the properties of the Futamura projections follow from our specification.

7.1 First projection

The first Futamura projection states that partially evaluating an interpreter with respect to a program gives a translation of the program into the language of the partial evaluator.

To get this projection, Mogensen exhibits the following annotation E^a of E with type $T_d \to d$:

$$\begin{aligned} E^a \equiv\ & Y\ \lambda e.\lambda m.m\ (\lambda x.x) \\ & \qquad\qquad (\lambda mn.(e\,m)\_(e\,n)) \\ & \qquad\qquad (\lambda m.\underline{\lambda} v.e\,(m\,v)) \\ \text{where} & \\ Y \equiv\ & \lambda h.(\lambda x.h\,(x\,x))\,(\lambda x.h\,(x\,x)) \end{aligned}$$

Hence for any closed term M, we have

$$\begin{array}{ll} R_{T_d \to d}(P\lfloor E^a \rfloor, E) & \text{Main Theorem} \\ R_{T_d}(\lceil M \rceil, \lceil M \rceil) & \text{Lemma 5} \\ E(P\lfloor E^a \rfloor \lceil M \rceil) = E\lceil M \rceil = M & \text{Specialization Theorem} \end{array}$$

So for any N, we have $E(P\lfloor E^a \rfloor \lceil M \rceil)N = MN$. Hence there is a closed term R such that $P\lfloor E^a \rfloor \lceil M \rceil = \lceil R \rceil$ and for all N, $L(R, N) \equiv L(M, N)$, as desired.

We can generalize this result to an arbitrary programming language L as defined in Section 2. Assume L is specified by a coding $\langle - \rangle$ and an interpreter S. If the coding $\langle - \rangle$ uses either ordinary or higher-order abstract syntax, then the same reasoning as Lemma 5 will hold, so that we get $R_{t_L}(\langle M \rangle, \langle M \rangle)$ for some type t_L that represents the grammar of the language.

If S is compositional, then we can always annotate S with type $t_L \to d$. This effectively views S as translating from the programs of the source language into λ-terms (rather than into their meanings). We can do this by marking all of S as dynamic, except for those parts that recursively call S. For example, a fragment of the semantics might be

$$S\langle e_1 + e_2 \rangle = \lambda \rho.(+(S\langle e_1 \rangle \rho))(S\langle e_2 \rangle \rho)$$

where we have explicitly curried the $+$. We would annotate this as:

$$S\langle e_1 + e_2 \rangle = \underline{\lambda} \rho.(+\_((S\langle e_1 \rangle)\_\rho))\_((S\langle e_2 \rangle)\_\rho)$$

In the homelier language of backquotes and commas, the right-hand side would be something like

```
`(lambda (r) ((+ (,(S e1) r)) (,(S e2) r)))
```

Now we can state the first projection:

Corollary 6 (First Projection)
Let S^a be any annotation of S with type $t_L \to d$. If $P\lfloor S^a \rfloor \langle \pi \rangle = \lceil T \rceil$ then T is a translation of π.

Proof

Calculating as before, we get

$$\begin{array}{ll} R_{t_L \to d}(P\lfloor S^a \rfloor, S) & \text{Main Theorem} \\ R_{t_L}(\langle \pi \rangle, \langle \pi \rangle) & \text{for any } \pi \text{ in the language} \\ E(P\lfloor S^a \rfloor \langle \pi \rangle) = S\langle \pi \rangle & \text{Specialization Theorem} \end{array}$$

So let $P\lfloor S^a \rfloor \langle \pi \rangle = \lceil T \rceil$. Then

$$T\langle d \rangle = E\lceil T \rceil \langle d \rangle = E(P\lfloor S^a \rfloor \langle \pi \rangle)\langle d \rangle = S\langle \pi \rangle \langle d \rangle$$

as desired. □

This projection is the basis for the compilers developed in (Wand, 1982; Clinger, 1984; Wand and Oliva, 1992), etc. In each case, we begin with a suitable semantics (typically a stack semantics) for the source language. The stack semantics (say S) is turned into a compiler by performing a binding-time analysis to separate compile-time and run-time operations, thus getting a compiler S^a written in the language of annotated terms. The language of annotated or 2-level terms is a natural language in which to write a compiler, and P acts as the interpreter for 2-level terms, much as E acts as the interpreter for pure λ-terms. We will explore these connections further in Section 8.

7.2 *Second projection*

The second projection states that partially evaluating the partial evaluator with respect to an interpreter gives a compiler.

Definition 6

If L is a programming language, then a compiler for L is a λ-term C such that for any L-program π, $C\langle \pi \rangle = \lceil T \rceil$, where T is a translation of π.

Recall that T is a translation of π iff for all L-data d, $T\langle d \rangle = S\langle \pi \rangle \langle d \rangle$, where S is the interpreter for L. Therefore, to show C is a compiler, it suffices to show that $E(C\langle \pi \rangle) = S\langle \pi \rangle$.

We can proceed as before for the syntax $\lfloor - \rfloor$ of annotations, getting a type

$$\begin{aligned} U_\beta = (\mu t).((\beta \to \beta) \\ &\to (t \to t \to \beta) \\ &\to ((\beta \to t) \to \beta) \\ &\to (t \to t \to \beta) \\ &\to ((\beta \to t) \to \beta) \\ &\to \beta) \end{aligned}$$

for codings of annotated terms (Mogensen, 1992b), and a lemma

Lemma 6

If W is a closed annotated λ-term, and β is any type, then $R_{U_\beta}(\lfloor W \rfloor, \lfloor W \rfloor)$.

Proof
As for Lemma 5. □

Mogensen gives the following annotation P^a of P with type $U_d \to d$:

$$\begin{aligned} P^a \equiv\ & Y\ \lambda p.\lambda m.m\ (\lambda x.x) \\ & \qquad (\lambda ww'.(p\,w)\_(p\,w')) \\ & \qquad (\lambda w.\underline{\lambda} v.p\,(m\,v)) \\ & \qquad (\lambda ww'.\underline{\lambda} abc.b\_(p\,w)\_(p\,w')) \\ & \qquad (\lambda w.\underline{\lambda} abc.c\_(\underline{\lambda} v.p\,(m\,(\underline{\lambda} abc.a\_v)))) \end{aligned}$$

where

$$Y \equiv \lambda h.(\lambda x.h\,(x\,x))\,(\lambda x.h\,(x\,x))$$

Corollary 7 (Second Projection)
Let L be a language with interpreter S and annotation S^a of type $t_L \to d$, and let P^a be any annotation of P with type $U_d \to d$. If $P\lfloor P^a\rfloor\lfloor S^a\rfloor = \lceil C\rceil$, then C is a compiler for L.

Proof
We have:

$$\begin{array}{ll} R_{U_d\to d}(P\lfloor P^a\rfloor, P) & \text{Main Theorem} \\ R_{U_d}(\lfloor S^a\rfloor, \lfloor S^a\rfloor) & \text{Lemma 2} \\ E(P\lfloor P^a\rfloor\lfloor S^a\rfloor) = P\lfloor S^a\rfloor & \text{Specialization Theorem} \end{array}$$

Let $P\lfloor P^a\rfloor\lfloor S^a\rfloor = \lceil C\rceil$. We can now check to see that C is a compiler:

$$\begin{array}{ll} E(C\langle\pi\rangle) & \\ = E(E\lceil C\rceil\langle\pi\rangle) & \\ = E(E(P\lfloor P^a\rfloor\lfloor S^a\rfloor)\langle\pi\rangle) & \text{definition of } \lceil C\rceil \\ = E(P\lfloor S^a\rfloor\langle\pi\rangle) & \text{preceding calculation} \\ = S\langle\pi\rangle & \text{first projection} \end{array}$$

□

7.3 Third Projection

Futamura's third projection states that partially evaluating the partial evaluator with respect to itself produces a compiler generator.

Definition 7
A *compiler generator* is a term G such that for any language L with interpreter S and annotation S^a of type $t_L \to d$, $G\lfloor S^a\rfloor = \lceil C\rceil$, where C is a compiler for L.

Corollary 8 (Third Projection)
Let $P\lfloor P^a\rfloor\lfloor P^a\rfloor = \lceil G\rceil$. Then G is a compiler-generator.

Proof
Let $P\lfloor P^a\rfloor\lfloor P^a\rfloor = \lceil G\rceil$. We first calculate, as before:

$$\begin{array}{ll} R_{U_d\to d}(P\lfloor P^a\rfloor, P) & \text{Main Theorem} \\ R_{U_d}(\lfloor P^a\rfloor, \lfloor P^a\rfloor) & \text{Lemma 2} \\ E(P\lfloor P^a\rfloor\lfloor P^a\rfloor) = P\lfloor P^a\rfloor & \text{Specialization Theorem} \end{array}$$

To check that G is a compiler-generator, let us define C by $G\lfloor S^a\rfloor = \lceil C\rceil$. We need to show that C is a compiler. Now

$$\begin{aligned}\lceil C\rceil = G\lfloor S^a\rfloor &= E\lceil G\rceil\lfloor S^a\rfloor \\ &= E(P\lfloor P^a\rfloor\lfloor P^a\rfloor)\lfloor S^a\rfloor \\ &= P\lfloor P^a\rfloor\lfloor S^a\rfloor \qquad \text{by the preceding calculation}\end{aligned}$$

But the second projection showed that if we have $\lceil C\rceil = P\lfloor P^a\rfloor\lfloor S^a\rfloor$, then C is a compiler. So G is a compiler generator. □

Note that these projections were derived solely from the definition of R_t and the fact that P satisfied the Main Theorem. They did not depend on the text of P. So they would continue to work for any other partial evaluator that satisfied the Main Theorem. This supports the thesis that the Main Theorem is a proper specification of a partial evaluator.

8 Operational semantics of the 2-level lambda calculus

Elsewhere in this issue, Palsberg (1993) presents a correctness result based on an operational semantics for annotated terms W. In our presentation, the term P serves as an interpreter for annotated terms, yielding a denotational (though not compositional) semantics. In order to compare his results with ours, we present a brief account of an operational semantics for annotated terms.

Recall that the grammar of annotated (or 2-level) terms is given by

$$W ::= v \mid W\,W \mid \lambda v.W \mid W\_W \mid \underline{\lambda} v.W$$

We can define substitution on these terms in the obvious way, treating both λ and $\underline{\lambda}$ as binders. Let $|W|$ denote the λ-term obtained from a 2-level term W by erasing all the binding-time information.

We can define *static reduction* on 2-level terms as follows:

Definition 8
$\rightarrow_s$ is the smallest relation on 2-level terms such that:

1. $(\lambda x.W)W' \rightarrow_s W[W'/x]$
2. If $W \rightarrow_s W'$ then $WV \rightarrow_s W'V$, $VW \rightarrow_s VW'$ $\lambda v.W \rightarrow_s \lambda v.W'$, $W\_V \rightarrow_s W'\_V$, $V\_W \rightarrow_s V\_W'$, and $\underline{\lambda} v.W \rightarrow_s \underline{\lambda} v.W'$.

$\rightarrow_s^*$ is the reflexive, transitive closure of $\rightarrow_s$.

So static reduction is ordinary β-reduction (on static redexes), closed under congruence of all term-builders. However, dynamic applications and abstractions are never reduced.

We first give a subject reduction theorem for 2-level terms under static reduction:

Theorem 3 (Subject Reduction Theorem)
If $A \vdash |W| : t\ \ [W]$ and $W \rightarrow_s^* W'$, then $A \vdash |W'| : t\ \ [W']$.

Proof
Consider a typed λ-calculus with the same types as considered above, with two constants

$$\begin{aligned} &Dapp : d \to d \to d \\ &Dabs : (d \to d) \to d \end{aligned}$$

It is clear that this calculus has the subject reduction property, even in the presence of recursive types, since the usual proof of the subject reduction theorem uses only induction on the size of terms, and not induction on the structure of types. Let us write $\vdash_1$ for the typing judgement in this calculus.

We give a translation $\overline{(-)}$ from 2-level terms to terms of this calculus as follows:

$$\begin{array}{ll} \overline{v} & \equiv v \\ \overline{W\,W'} & \equiv \overline{W}\ \overline{W'} \\ \overline{\lambda v.W} & \equiv \lambda v.\overline{W} \\ \overline{W\_W'} & \equiv Dapp\ \overline{W}\ \overline{W'} \\ \overline{\underline{\lambda} v.W} & \equiv Dabs(\lambda v.\overline{W}) \end{array}$$

Inspection of the typing rules in the two systems shows that $A \vdash |W| : t\ \ [W] \Longleftrightarrow A \vdash_1 \overline{W} : t$. Furthermore $W \to_s W'$ implies $\overline{W} \to \overline{W'}$. So if $A \vdash |W| : t\ \ [W]$ and $W \to_s W'$, we have $A \vdash_1 \overline{W} : t$. Hence $A \vdash_1 \overline{W'} : t$ by the subject reduction theorem, and therefore $A \vdash |W'| : t\ \ [W']$. The result then follows by induction on the number of reduction steps in the $\to_s^*$ reduction. $\square$

Now we can show that no static reduction of a well-typed term can go wrong.

Definition 9
A *confused redex* is a subterm of the form $(\underline{\lambda}x.W)V$ or $(\lambda x.W)\_V$.

Corollary 9 (Safety)
If $A \vdash M : t\ \ [W]$ and $W \to_s^* W'$, then W' contains no confused redex.

Proof
It is easy to see that no confused redex can have a type, and hence no well-typed term can contain a confused redex. $\square$

We can also easily show that static reduction is sound with respect to P, that is, it preserves the meaning of 2-level terms. We begin with two lemmas:

Lemma 7
For any 2-level term W, $P\lfloor \lambda x.W \rfloor = \lambda x.P\lfloor W \rfloor$ and $P\lfloor \underline{\lambda} x.W \rfloor = \underline{\lambda} x.P\lfloor W \rfloor$.

Proof
Similar to the corresponding proof for E in Mogensen (1992a). We do the case of static abstraction:

$$\begin{array}{ll} P\lfloor \lambda x.W \rfloor & \\ = P(\lambda abcde.c(\lambda x.\lfloor W \rfloor)) & \text{definition of } \lfloor - \rfloor \\ = \lambda v.P((\lambda x.\lfloor W \rfloor)v) & \text{definition of } P \\ = \lambda x.P((\lambda x.\lfloor W \rfloor)x) & \alpha\text{-conversion} \\ = \lambda x.P\lfloor W \rfloor & \beta\text{-reduction} \end{array}$$

$\square$

Lemma 8

For any 2-level terms W and W', $P\lfloor W[W'/x]\rfloor = (P\lfloor W\rfloor)[P\lfloor W'\rfloor/x]$

Proof

By induction on the size of W. We do two representative cases. For the base case, when $W \equiv x$, we have

$$\begin{aligned}
&(P\lfloor x\rfloor)[P\lfloor W'\rfloor/x] && \\
&= x[P\lfloor W'\rfloor/x] && \text{definition of } P\\
&= P\lfloor W'\rfloor && \text{substitution}\\
&= P\lfloor x[W'/x]\rfloor && \text{substitution}
\end{aligned}$$

For static abstractions over a variable y distinct from x, we calculate as follows, assuming without loss of generality that a, b, c, d, e, and v are fresh:

$$\begin{aligned}
&P(\lfloor(\lambda y.V)[W'/x]\rfloor) && \\
&= P(\lfloor\lambda y'.V[y'/y][W'/x]\rfloor) && \text{substitution}\\
&= P(\lambda abcde.c(\lambda y'.\lfloor V[y'/y][W'/x]\rfloor)) && \text{definition of } \lfloor - \rfloor\\
&= \lambda v.P((\lambda y'.\lfloor V[y'/y][W'/x]\rfloor)v) && \text{definition of } P\\
&= \lambda y'.P((\lambda y'.\lfloor V[y'/y][W'/x]\rfloor)y') && \alpha\text{-conversion}\\
&= \lambda y'.P\lfloor V[y'/y][W'/x]\rfloor && \beta\text{-reduction}\\
&= \lambda y'.((P\lfloor V[y'/y]\rfloor)[P\lfloor W'\rfloor/x]) && \text{induction hypothesis at } V[y'/y]\\
& && (V[y'/y] \text{ is smaller than } \lambda y.V)\\
&= \lambda y'.((P\lfloor V\rfloor)[P\lfloor y'\rfloor/y][P\lfloor W'\rfloor/x]) && \text{induction hypothesis at } V\\
&= \lambda y'.((P\lfloor V\rfloor)[y'/y][P\lfloor W'\rfloor/x]) && \text{definition of } P\\
&= (\lambda y.P\lfloor V\rfloor)[P\lfloor W'\rfloor/x] && \text{substitution}\\
&= (P\lfloor\lambda y.V\rfloor)[P\lfloor W'\rfloor/x] && \text{Lemma 7}
\end{aligned}$$

□

Theorem 4 (Soundness of static reduction)

If $W \rightarrow_s W'$, then $P\lfloor W\rfloor = P\lfloor W'\rfloor$.

Proof

By induction on the definition of $\rightarrow_s$. For the base case, β-reduction, we have $(\lambda x.W)W' \rightarrow_s W[W'/x]$. Then

$$\begin{aligned}
&P\lfloor(\lambda x.W)W'\rfloor && \\
&= (P\lfloor\lambda x.W\rfloor)(P\lfloor W'\rfloor) && \text{definition of } P\\
&= (\lambda x.P\lfloor W\rfloor)(P\lfloor W'\rfloor) && \text{Lemma 7}\\
&= (P\lfloor W\rfloor)[P\lfloor W'\rfloor/x] && \beta\text{-reduction}\\
&= P(\lfloor W[W'/x]\rfloor) && \text{Lemma 8}
\end{aligned}$$

For the induction step, we need to show that if $P\lfloor W\rfloor = P\lfloor W'\rfloor$, then $P\lfloor W V\rfloor = P\lfloor W'V\rfloor$, $P\lfloor\lambda x.W\rfloor = P\lfloor\lambda x.W'\rfloor$, etc. The application cases are trivial from the compositionality of P; the abstraction cases follow immediately from Lemma 7. □

9 Comparison with related work

Mogensen (1992b) is the primary source for our development. He exhibits the various terms above. He also includes the binding-time annotations that allow the partial evaluator to be self-applied. He proves the correctness of the self-interpreter (Mogensen, 1992a), but does not give either a specification or proof of the partial evaluator.

Gomard (1992) discusses a larger self-applicable partial evaluator in a two-level λ-calculus with constants. He gives a correctness criterion similar in intent to ours. Our specification is simpler because it avoids the model-theoretic details and assumptions used in Gomard (1992).

Both Mogensen and Gomard, as do we, use a two-level type system like that of Nielson and Nielson (1988), except that the run-time code is untyped. Launchbury (1991) discusses the encoding issues for a strongly-typed self-applicable partial evaluator with a full multi-level type system. However, he remarks in Section 6 of (Launchbury, 1991) that the same encoding issues arise in the untyped case. This paper is in part an attempt to sort out these issues.

Launchbury (1989) gives a characterization of binding-time analysis in terms of congruence relations: a quantity may be deferred to run-time if its value does not alter the result of the compile-time computation. This idea was neatly extended to higher-order computations and partially-static structures by Hunt and Sands (1991) using partial equivalence relations. Unfortunately, this characterization does not seem to be strong enough to justify emitting code on the basis of the binding-time analysis, which is our primary goal.

Consel (1990) presents an algorithm for binding-time analysis for a higher-order untyped language with partially static structures, based on an abstract interpretation. However, no correctness criterion is stated. It would be interesting to see how our approach might handle partially static structures. More recently, Consel and Khoo (1992) have shown the correctness of on-line and off-line partial evaluators for a first-order language using an instrumented semantics; in this framework binding-time analysis appears as an abstract interpretation of the on-line partial evaluation. They then derive the off-line specializer (corresponding to our P) from the on-line partial evaluator by coarsening the tests to depend only on the data available from the binding-time analysis. Their approach also shows the correctness of the binding-time analysis and specialization together, but the technical details seem quite different from ours; their approach is heavily dependent on model-theoretic details that we have been able to avoid. It would be interesting to see if our results could be adapted to give a simpler account in the first-order case.

Palsberg (1993) gives an operational semantics of 2-level λ-terms equivalent to the account in Section 8. His main result is analogous to our Corollary 9, which states that no static reduction of a well-typed 2-level term can go wrong. He identifies a subclass of the static reductions, which he calls the *top-down* reductions, and a different notion of acceptability, which he calls *well-annotatedness.* He then shows that no top-down reduction of a well-annotated term can go wrong. In comparison with our Corollary 9, his result allows more 2-level terms to be considered, but restricts the class of permissible reductions.

10 Conclusions

This paper is part of an investigation into the nature of program analysis and optimization. The goal of program analysis is to annotate a program with certain propositions about the behavior of that program. One can then apply optimizations to the program that are justified by those propositions. Typically, an analysis is specified by a set of local consistency conditions (the 'flow equations'). Our approach is to show that any solution of the consistency conditions justifies the resulting transformation.

We have carried through this program for binding-time analysis. Following Gomard (1990) and Mogensen (1992b), we formulated binding-time analysis as the solution of a set of local constraints (the typing rules). We then showed that any such solution was a valid input to a simple program specializer, that is, if the specializer is given any annotation of the input program that satisfies the local constraints, its output is a suitably specialized version of the input program.

In order to do this, we formalized the notion of a 'suitably specialized version of the input program.' Our version of this notion is similar to, but simpler than, the specification given by Gomard (1992). We then showed that the typical properties of partial evaluators, such as the Futamura projections, follow from the specification.

By considering both an analysis and the transformation it justifies together, this proof suggests a framework for incorporating flow analyses into verified compilers. By separating this soundness result from the algorithm for solving the flow equations, we obtain much simpler proofs, avoiding the many model-theoretic details that complicate typical proofs involving abstract interpretation.

This work suggests that the proposition associated with a program analysis can simply be that 'the optimization works.' This avoids a morass of model-theoretic details, at the expense of needing to know what it means for the proposed optimization to work for each possible result of the flow analysis. Thus the flow analysis and its enabled optimization should be proved correct together.

Acknowledgements

Thanks to Bob Muller, whose questions in our seminar got me started on this topic, and to Neil Jones, Torben Mogensen, Jens Palsberg, and Carsten Gomard for their extensive E-mail discussions and comments.

References

Barendregt, Henk. (1991) Self-interpretation in lambda calculus. *Journal of functional programming*, **1**(2), 229–234.

Clinger, William. (1984) (Aug.). The scheme 311 compiler: An exercise in denotational semantics. *Pages 356–364 of: Proc. 1984 ACM Symposium on Lisp and Functional Programming.*

Consel, Charles. (1990) Binding time analysis for higher order untyped functional languages. *Pages 264–272 of: Proc. 1990 ACM Symposium on Lisp and Functional Programming.*

Consel, Charles, & Khoo, Siau Cheng. (1992) (June). *On-line & off-line partial evaluation: Semantic specifications and correctness proofs.* Tech. rept. YALEU/DCS/RR-912. Yale University Department of Computer Science.

Gomard, Carsten K. (1990) Partial type inference for untyped functional programs. *Pages 282–287 of: Proc. 1990 ACM Symposium on Lisp and Functional Programming.*

Gomard, Carsten K. (1992) A self-applicable partial evaluator for the lambda calculus: Correctness and pragmatics. *Acm transactions on programming languages and systems,* **14**(2), 147–172.

Henglein, Fritz. (1991) Efficient type inference for higher-order binding-time analysis. *Pages 448–472 of:* Hughes, J. (ed), *Functional programming languages and computer architecture, 5th acm conference.* Lecture Notes in Computer Science, vol. 523. Berlin, Heidelberg, New York: Springer-Verlag.

Hunt, Sebastian, & Sands, David. (1991) Binding time analysis: A new perspective. *Pages 154–165 of: Proceedings of the symposium on partial evaluation and semantics-based program manipulation.* SIGPLAN Notices 26(9), September, 1991.

Jorring, U., & Scherlis, William L. (1986) Compilers and staging transformations. *Pages 86–96 of: Conf. Rec. 13th ACM Symposium on Principles of Programming Languages.*

Launchbury, John. (1989) (Nov.). *Projection factorizations in partial evaluation.* Ph.D. thesis, University of Glasgow.

Launchbury, John. (1991) A strongly-typed self-applicable partial evaluator. *In:* Hughes, John (ed), *Functional programming languages and computer architecture,* vol. 523. Berlin, Heidelberg, and New York: Springer-Verlag.

Mogensen, Torben Æ. (1992a) (June). *Efficient self-interpretation in lambda calculus.* to appear.

Mogensen, Torben Æ. (1992b) Self-applicable partial evaluation for pure lambda calculus. *Pages 116–121 of:* Consel, Charles (ed), *Acm sigplan workshop on partial evaluation and semantics-based program manipulation.*

Nielson, Flemming, & Nielson, Hanne Riis. (1988) Two-level semantics and code generation. *Theoretical computer science,* **56**, 59–133.

Palsberg, Jens. (1993) Correctness of binding time analysis. *Journal of functional programming,* **11**.

Pfenning, Frank, & Elliott, Conal. (1988) (June). Higher-order abstract syntax. *Pages 199–208 of: Proceedings sigplan '88 conference on programming language design and implementation.*

Wand, Mitchell. (1982) Deriving target code as a representation of continuation semantics. *Acm transactions on programming languages and systems,* **4**(3), 496–517.

Wand, Mitchell, & Oliva, Dino P. (1992) Proving the correctness of storage representations. *Pages 151–160 of: Proc. 1992 ACM Symposium on Lisp and Functional Programming.*